mmcv Documentation

Release 1.0.0rc4

MMCV Contributors

Aug 17, 2022

GET STARTED

CHAPTER

ONE

PREREQUISITES

In this section we demonstrate how to prepare an environment with PyTorch. MMDection3D works on Linux, Windows (experimental support) and macOS and requires the following packages:

- Python 3.6+
- PyTorch 1.3+
- CUDA 9.2+ (If you build PyTorch from source, CUDA 9.0 is also compatible)
- GCC 5+
- MMCV

Note: If you are experienced with PyTorch and have already installed it, just skip this part and jump to the *next section*. Otherwise, you can follow these steps for the preparation.

Step 0. Download and install Miniconda from the official website.

Step 1. Create a conda environment and activate it.

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

Step 2. Install PyTorch following official instructions, e.g.

On GPU platforms:

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

On CPU platforms:

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

CHAPTER

TWO

INSTALLATION

We recommend that users follow our best practices to install MMDetection3D. However, the whole process is highly customizable. See *Customize Installation* section for more information.

2.1 Best Practices

Assuming that you already have CUDA 11.0 installed, here is a full script for quick installation of MMDetection3D with conda. Otherwise, you should refer to the step-by-step installation instructions in the next section.

```
pip install openmim
mim install mmcv-full
mim install mmdet
mim install mmsegmentation
git clone https://github.com/open-mmlab/mmdetection3d.git
cd mmdetection3d
pip install -e .
```

Step 0. Install MMCV using MIM.

Step 1. Install MMDetection.

```
pip install mmdet
```

Optionally, you could also build MMDetection from source in case you want to modify the code:

```
git clone https://github.com/open-mmlab/mmdetection.git
cd mmdetection
git checkout v2.24.0  # switch to v2.24.0 branch
pip install -r requirements/build.txt
pip install -v -e .  # or "python setup.py develop"
```

Step 2. Install MMSegmentation.

```
pip install mmsegmentation
```

Optionally, you could also build MMSegmentation from source in case you want to modify the code:

```
git clone https://github.com/open-mmlab/mmsegmentation.git
cd mmsegmentation
git checkout v0.20.0  # switch to v0.20.0 branch
pip install -e .  # or "python setup.py develop"
```

Step 3. Clone the MMDetection3D repository.

```
git clone https://github.com/open-mmlab/mmdetection3d.git
cd mmdetection3d
```

Step 4. Install build requirements and then install MMDetection3D.

```
pip install -v -e .  # or "python setup.py develop"
```

Note:

1. The git commit id will be written to the version number with step d, e.g. 0.6.0+2e7045c. The version will also be saved in trained models. It is recommended that you run step d each time you pull some updates from github. If C++/CUDA codes are modified, then this step is compulsory.

   Important: Be sure to remove the `./build` folder if you reinstall mmdet with a different CUDA/PyTorch version.

   ```
   pip uninstall mmdet3d
   rm -rf ./build
   find . -name "*.so" | xargs rm
   ```

2. Following the above instructions, MMDetection3D is installed on `dev` mode, any local modifications made to the code will take effect without the need to reinstall it (unless you submit some commits and want to update the version number).

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

4. 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`.

   We have supported spconv2.0. If the user has installed spconv2.0, the code will use spconv2.0 first, which will take up less GPU memory than using the default mmcv spconv. Users can use the following commands to install spconv2.0:

   ```
   pip install cumm-cuxxx
   pip install spconv-cuxxx
   ```

   Where xxx is the CUDA version in the environment.

   For example, using CUDA 10.2, the command will be `pip install cumm-cu102 && pip install spconv-cu102`.

   Supported CUDA versions include 10.2, 11.1, 11.3, and 11.4. Users can also install it by building from the source. For more details please refer to spconv v2.x.

   We also support Minkowski Engine as a sparse convolution backend. If necessary please follow original installation guide or use `pip`:

   ```
   conda install openblas-devel -c anaconda
   pip install -U git+https://github.com/NVIDIA/MinkowskiEngine -v --no-deps --install-
   ↪option="--blas_include_dirs=/opt/conda/include" --install-option="--blas=openblas"
   ```

5. The code can not be built for CPU only environment (where CUDA isn't available) for now.

2.2 Verification

2.2.1 Verify with point cloud demo

We provide several demo scripts to test a single sample. Pre-trained models can be downloaded from *model zoo*. To test a single-modality 3D detection on point cloud scenes:

```
python demo/pcd_demo.py ${PCD_FILE} ${CONFIG_FILE} ${CHECKPOINT_FILE} [--device ${GPU_ID}
↪] [--score-thr ${SCORE_THR}] [--out-dir ${OUT_DIR}]
```

Examples:

```
python demo/pcd_demo.py demo/data/kitti/kitti_000008.bin configs/second/hv_second_secfpn_
↪6x8_80e_kitti-3d-car.py checkpoints/hv_second_secfpn_6x8_80e_kitti-3d-car_20200620_
↪230238-393f000c.pth
```

If you want to input a `ply` file, you can use the following function and convert it to `bin` format. Then you can use the converted `bin` file to generate demo. Note that you need to install `pandas` and `plyfile` before using this script. This function can also be used for data preprocessing for training `ply data`.

```
import numpy as np
import pandas as pd
from plyfile import PlyData

def convert_ply(input_path, output_path):
    plydata = PlyData.read(input_path)  # read file
    data = plydata.elements[0].data  # read data
    data_pd = pd.DataFrame(data)  # convert to DataFrame
    data_np = np.zeros(data_pd.shape, dtype=np.float)  # initialize array to store data
    property_names = data[0].dtype.names  # read names of properties
    for i, name in enumerate(
            property_names):  # read data by property
        data_np[:, i] = data_pd[name]
    data_np.astype(np.float32).tofile(output_path)
```

Examples:

```
convert_ply('./test.ply', './test.bin')
```

If you have point clouds in other format (`off`, `obj`, etc.), you can use `trimesh` to convert them into `ply`.

```
import trimesh

def to_ply(input_path, output_path, original_type):
    mesh = trimesh.load(input_path, file_type=original_type)  # read file
    mesh.export(output_path, file_type='ply')  # convert to ply
```

Examples:

```
to_ply('./test.obj', './test.ply', 'obj')
```

More demos about single/multi-modality and indoor/outdoor 3D detection can be found in *demo*.

2.3 Customize Installation

2.3.1 CUDA Versions

When installing PyTorch, you need to specify the version of CUDA. If you are not clear on which to choose, follow our recommendations:

- For Ampere-based NVIDIA GPUs, such as GeForce 30 series and NVIDIA A100, CUDA 11 is a must.
- For older NVIDIA GPUs, CUDA 11 is backward compatible, but CUDA 10.2 offers better compatibility and is more lightweight.

Please make sure the GPU driver satisfies the minimum version requirements. See this table for more information.

Note: Installing CUDA runtime libraries is enough if you follow our best practices, because no CUDA code will be compiled locally. However if you hope to compile MMCV from source or develop other CUDA operators, you need to install the complete CUDA toolkit from NVIDIA's website, and its version should match the CUDA version of PyTorch. i.e., the specified version of cudatoolkit in `conda install` command.

2.3.2 Install MMCV without MIM

MMCV contains C++ and CUDA extensions, thus depending on PyTorch in a complex way. MIM solves such dependencies automatically and makes the installation easier. However, it is not a must.

To install MMCV with pip instead of MIM, please follow MMCV installation guides. This requires manually specifying a find-url based on PyTorch version and its CUDA version.

For example, the following command install mmcv-full built for PyTorch 1.10.x and CUDA 11.3.

```
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu113/torch1.10/index.
↪html
```

2.3.3 Using MMDetection3D with Docker

We provide a Dockerfile to build an image.

```
# build an image with PyTorch 1.6, CUDA 10.1
docker build -t mmdetection3d -f docker/Dockerfile .
```

Run it with

```
docker run --gpus all --shm-size=8g -it -v {DATA_DIR}:/mmdetection3d/data mmdetection3d
```

2.3.4 A from-scratch setup script

Here is a full script for setting up MMdetection3D with conda.

```
conda create -n open-mmlab python=3.7 -y
conda activate open-mmlab

# install latest PyTorch prebuilt with the default prebuilt CUDA version (usually the␣
↪latest)
conda install -c pytorch pytorch torchvision -y

# install mmcv
pip install mmcv-full

# install mmdetection
pip install git+https://github.com/open-mmlab/mmdetection.git

# install mmsegmentation
pip install git+https://github.com/open-mmlab/mmsegmentation.git

# install mmdetection3d
git clone https://github.com/open-mmlab/mmdetection3d.git
cd mmdetection3d
pip install -v -e .
```

2.4 Trouble shooting

If you have some issues during the installation, please first view the *FAQ* page. You may open an issue on GitHub if no solution is found.

CHAPTER

THREE

DEMO

3.1 Introduction

We provide scripts for multi-modality/single-modality (LiDAR-based/vision-based), indoor/outdoor 3D detection and 3D semantic segmentation demos. The pre-trained models can be downloaded from model zoo. We provide pre-processed sample data from KITTI, SUN RGB-D, nuScenes and ScanNet dataset. You can use any other data following our pre-processing steps.

3.2 Testing

3.2.1 3D Detection

Single-modality demo

To test a 3D detector on point cloud data, simply run:

```
python demo/pcd_demo.py ${PCD_FILE} ${CONFIG_FILE} ${CHECKPOINT_FILE} [--device ${GPU_ID}
↪] [--score-thr ${SCORE_THR}] [--out-dir ${OUT_DIR}] [--show]
```

The visualization results including a point cloud and predicted 3D bounding boxes will be saved in `${OUT_DIR}/PCD_NAME`, which you can open using MeshLab. Note that if you set the flag `--show`, the prediction result will be displayed online using Open3D.

Example on KITTI data using SECOND model:

```
python demo/pcd_demo.py demo/data/kitti/kitti_000008.bin configs/second/hv_second_secfpn_
↪6x8_80e_kitti-3d-car.py checkpoints/hv_second_secfpn_6x8_80e_kitti-3d-car_20200620_
↪230238-393f000c.pth
```

Example on SUN RGB-D data using VoteNet model:

```
python demo/pcd_demo.py demo/data/sunrgbd/sunrgbd_000017.bin configs/votenet/votenet_
↪16x8_sunrgbd-3d-10class.py checkpoints/votenet_16x8_sunrgbd-3d-10class_20200620_230238-
↪4483c0c0.pth
```

Remember to convert the VoteNet checkpoint if you are using mmdetection3d version >= 0.6.0. See its README for detailed instructions on how to convert the checkpoint.

Multi-modality demo

To test a 3D detector on multi-modality data (typically point cloud and image), simply run:

```
python demo/multi_modality_demo.py ${PCD_FILE} ${IMAGE_FILE} ${ANNOTATION_FILE} ${CONFIG_
↪FILE} ${CHECKPOINT_FILE} [--device ${GPU_ID}] [--score-thr ${SCORE_THR}] [--out-dir $
↪{OUT_DIR}] [--show]
```

where the ANNOTATION_FILE should provide the 3D to 2D projection matrix. The visualization results including a point cloud, an image, predicted 3D bounding boxes and their projection on the image will be saved in ${OUT_DIR}/PCD_NAME.

Example on KITTI data using MVX-Net model:

```
python demo/multi_modality_demo.py demo/data/kitti/kitti_000008.bin demo/data/kitti/
↪kitti_000008.png demo/data/kitti/kitti_000008_infos.pkl configs/mvxnet/dv_mvx-fpn_
↪second_secfpn_adamw_2x8_80e_kitti-3d-3class.py checkpoints/dv_mvx-fpn_second_secfpn_
↪adamw_2x8_80e_kitti-3d-3class_20200621_003904-10140f2d.pth
```

Example on SUN RGB-D data using ImVoteNet model:

```
python demo/multi_modality_demo.py demo/data/sunrgbd/sunrgbd_000017.bin demo/data/
↪sunrgbd/sunrgbd_000017.jpg demo/data/sunrgbd/sunrgbd_000017_infos.pkl configs/
↪imvotenet/imvotenet_stage2_16x8_sunrgbd-3d-10class.py checkpoints/imvotenet_stage2_
↪16x8_sunrgbd-3d-10class_20210323_184021-d44dcb66.pth
```

3.2.2 Monocular 3D Detection

To test a monocular 3D detector on image data, simply run:

```
python demo/mono_det_demo.py ${IMAGE_FILE} ${ANNOTATION_FILE} ${CONFIG_FILE} $
↪{CHECKPOINT_FILE} [--device ${GPU_ID}] [--out-dir ${OUT_DIR}] [--show]
```

where the ANNOTATION_FILE should provide the 3D to 2D projection matrix (camera intrinsic matrix). The visualization results including an image and its predicted 3D bounding boxes projected on the image will be saved in ${OUT_DIR}/PCD_NAME.

Example on nuScenes data using FCOS3D model:

```
python demo/mono_det_demo.py demo/data/nuscenes/n015-2018-07-24-11-22-45+0800__CAM_BACK__
↪1532402927637525.jpg demo/data/nuscenes/n015-2018-07-24-11-22-45+0800__CAM_BACK__
↪1532402927637525_mono3d.coco.json configs/fcos3d/fcos3d_r101_caffe_fpn_gn-head_dcn_2x8_
↪1x_nus-mono3d_finetune.py checkpoints/fcos3d_r101_caffe_fpn_gn-head_dcn_2x8_1x_nus-
↪mono3d_finetune_20210717_095645-8d806dc2.pth
```

Note that when visualizing results of monocular 3D detection for flipped images, the camera intrinsic matrix should also be modified accordingly. See more details and examples in PR #744.

3.2.3 3D Segmentation

To test a 3D segmentor on point cloud data, simply run:

```
python demo/pc_seg_demo.py ${PCD_FILE} ${CONFIG_FILE} ${CHECKPOINT_FILE} [--device ${GPU_
↪ID}] [--out-dir ${OUT_DIR}] [--show]
```

The visualization results including a point cloud and its predicted 3D segmentation mask will be saved in `${OUT_DIR}/PCD_NAME`.

Example on ScanNet data using PointNet++ (SSG) model:

```
python demo/pc_seg_demo.py demo/data/scannet/scene0000_00.bin configs/pointnet2/
↪pointnet2_ssg_16x2_cosine_200e_scannet_seg-3d-20class.py checkpoints/pointnet2_ssg_
↪16x2_cosine_200e_scannet_seg-3d-20class_20210514_143644-ee73704a.pth
```

CHAPTER

FOUR

MODEL ZOO

4.1 Common settings

- We use distributed training.
- For fair comparison with other codebases, we report the GPU memory as the maximum value of `torch.cuda.max_memory_allocated()` for all 8 GPUs. Note that this value is usually less than what `nvidia-smi` shows.
- We report the inference time as the total time of network forwarding and post-processing, excluding the data loading time. Results are obtained with the script benchmark.py which computes the average time on 2000 images.

4.2 Baselines

4.2.1 SECOND

Please refer to SECOND for details. We provide SECOND baselines on KITTI and Waymo datasets.

4.2.2 PointPillars

Please refer to PointPillars for details. We provide pointpillars baselines on KITTI, nuScenes, Lyft, and Waymo datasets.

4.2.3 Part-A2

Please refer to Part-A2 for details.

4.2.4 VoteNet

Please refer to VoteNet for details. We provide VoteNet baselines on ScanNet and SUNRGBD datasets.

4.2.5 Dynamic Voxelization

Please refer to Dynamic Voxelization for details.

4.2.6 MVXNet

Please refer to MVXNet for details.

4.2.7 RegNetX

Please refer to RegNet for details. We provide pointpillars baselines with RegNetX backbones on nuScenes and Lyft datasets currently.

4.2.8 nuImages

We also support baseline models on nuImages dataset. Please refer to nuImages for details. We report Mask R-CNN, Cascade Mask R-CNN and HTC results currently.

4.2.9 H3DNet

Please refer to H3DNet for details.

4.2.10 3DSSD

Please refer to 3DSSD for details.

4.2.11 CenterPoint

Please refer to CenterPoint for details.

4.2.12 SSN

Please refer to SSN for details. We provide pointpillars with shape-aware grouping heads used in SSN on the nuScenes and Lyft datasets currently.

4.2.13 ImVoteNet

Please refer to ImVoteNet for details. We provide ImVoteNet baselines on SUNRGBD dataset.

4.2.14 FCOS3D

Please refer to FCOS3D for details. We provide FCOS3D baselines on the nuScenes dataset.

4.2.15 PointNet++

Please refer to PointNet++ for details. We provide PointNet++ baselines on ScanNet and S3DIS datasets.

4.2.16 Group-Free-3D

Please refer to Group-Free-3D for details. We provide Group-Free-3D baselines on ScanNet dataset.

4.2.17 ImVoxelNet

Please refer to ImVoxelNet for details. We provide ImVoxelNet baselines on KITTI dataset.

4.2.18 PAConv

Please refer to PAConv for details. We provide PAConv baselines on S3DIS dataset.

4.2.19 DGCNN

Please refer to DGCNN for details. We provide DGCNN baselines on S3DIS dataset.

4.2.20 SMOKE

Please refer to SMOKE for details. We provide SMOKE baselines on KITTI dataset.

4.2.21 PGD

Please refer to PGD for details. We provide PGD baselines on KITTI and nuScenes dataset.

4.2.22 PointRCNN

Please refer to PointRCNN for details. We provide PointRCNN baselines on KITTI dataset.

4.2.23 MonoFlex

Please refer to MonoFlex for details. We provide MonoFlex baselines on KITTI dataset.

4.2.24 SA-SSD

Please refer to SA-SSD for details. We provide SA-SSD baselines on the KITTI dataset.

4.3 FCAF3D

Please refer to FCAF3D for details. We provide FCAF3D baselines on the ScanNet, S3DIS and SUN RGB-D dataset.

4.3.1 Mixed Precision (FP16) Training

Please refer to Mixed Precision (FP16) Training on PointPillars for details.

CHAPTER

FIVE

DATASET PREPARATION

5.1 Before Preparation

It is recommended to symlink the dataset root to `$MMDETECTION3D/data`. If your folder structure is different from the following, you may need to change the corresponding paths in config files.

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
│   ├── nuscenes
│   │   ├── maps
│   │   ├── samples
│   │   ├── sweeps
│   │   ├── v1.0-test
|   |   ├── v1.0-trainval
│   ├── kitti
│   │   ├── ImageSets
│   │   ├── testing
│   │   │   ├── calib
│   │   │   ├── image_2
│   │   │   ├── velodyne
│   │   ├── training
│   │   │   ├── calib
│   │   │   ├── image_2
│   │   │   ├── label_2
│   │   │   ├── velodyne
│   ├── waymo
│   │   ├── waymo_format
│   │   │   ├── training
│   │   │   ├── validation
│   │   │   ├── testing
│   │   │   ├── gt.bin
│   │   ├── kitti_format
│   │   │   ├── ImageSets
│   ├── lyft
│   │   ├── v1.01-train
│   │   │   ├── v1.01-train (train_data)
│   │   │   ├── lidar (train_lidar)
```

(continues on next page)

(continued from previous page)

```
│   │   │   ├── images (train_images)
│   │   │   ├── maps (train_maps)
│   │   ├── v1.01-test
│   │   │   ├── v1.01-test (test_data)
│   │   │   ├── lidar (test_lidar)
│   │   │   ├── images (test_images)
│   │   │   ├── maps (test_maps)
│   │   ├── train.txt
│   │   ├── val.txt
│   │   ├── test.txt
│   │   ├── sample_submission.csv
│   ├── s3dis
│   │   ├── meta_data
│   │   ├── Stanford3dDataset_v1.2_Aligned_Version
│   │   ├── collect_indoor3d_data.py
│   │   ├── indoor3d_util.py
│   │   ├── README.md
│   ├── scannet
│   │   ├── meta_data
│   │   ├── scans
│   │   ├── scans_test
│   │   ├── batch_load_scannet_data.py
│   │   ├── load_scannet_data.py
│   │   ├── scannet_utils.py
│   │   ├── README.md
│   ├── sunrgbd
│   │   ├── OFFICIAL_SUNRGBD
│   │   ├── matlab
│   │   ├── sunrgbd_data.py
│   │   ├── sunrgbd_utils.py
│   │   ├── README.md
```

5.2 Download and Data Preparation

5.2.1 KITTI

Download KITTI 3D detection data HERE. Prepare KITTI data splits by running

```
mkdir ./data/kitti/ && mkdir ./data/kitti/ImageSets

# Download data split
wget -c  https://raw.githubusercontent.com/traveller59/second.pytorch/master/second/data/
↪ImageSets/test.txt --no-check-certificate --content-disposition -O ./data/kitti/
↪ImageSets/test.txt
wget -c  https://raw.githubusercontent.com/traveller59/second.pytorch/master/second/data/
↪ImageSets/train.txt --no-check-certificate --content-disposition -O ./data/kitti/
↪ImageSets/train.txt
wget -c  https://raw.githubusercontent.com/traveller59/second.pytorch/master/second/data/
↪ImageSets/val.txt --no-check-certificate --content-disposition -O ./data/kitti/
↪ImageSets/val.txt
```

(continues on next page)

(continued from previous page)

```
wget -c  https://raw.githubusercontent.com/traveller59/second.pytorch/master/second/data/
↪ImageSets/trainval.txt --no-check-certificate --content-disposition -O ./data/kitti/
↪ImageSets/trainval.txt
```

Then generate info files by running

```
python tools/create_data.py kitti --root-path ./data/kitti --out-dir ./data/kitti --
↪extra-tag kitti
```

In an environment using slurm, users may run the following command instead

```
sh tools/create_data.sh <partition> kitti
```

5.2.2 Waymo

Download Waymo open dataset V1.2 HERE and its data split HERE. Then put tfrecord files into corresponding folders in `data/waymo/waymo_format/` and put the data split txt files into `data/waymo/kitti_format/ImageSets`. Download ground truth bin file for validation set HERE and put it into `data/waymo/waymo_format/`. A tip is that you can use `gsutil` to download the large-scale dataset with commands. You can take this tool as an example for more details. Subsequently, prepare waymo data by running

```
python tools/create_data.py waymo --root-path ./data/waymo/ --out-dir ./data/waymo/ --
↪workers 128 --extra-tag waymo
```

Note that if your local disk does not have enough space for saving converted data, you can change the `out-dir` to anywhere else. Just remember to create folders and prepare data there in advance and link them back to `data/waymo/kitti_format` after the data conversion.

5.2.3 NuScenes

Download nuScenes V1.0 full dataset data HERE. Prepare nuscenes data by running

```
python tools/create_data.py nuscenes --root-path ./data/nuscenes --out-dir ./data/
↪nuscenes --extra-tag nuscenes
```

5.2.4 Lyft

Download Lyft 3D detection data HERE. Prepare Lyft data by running

```
python tools/create_data.py lyft --root-path ./data/lyft --out-dir ./data/lyft --extra-
↪tag lyft --version v1.01
python tools/data_converter/lyft_data_fixer.py --version v1.01 --root-folder ./data/lyft
```

Note that we follow the original folder names for clear organization. Please rename the raw folders as shown above. Also note that the second command serves the purpose of fixing a corrupted lidar data file. Please refer to the discussion here for more details.

5.2.5 S3DIS, ScanNet and SUN RGB-D

To prepare S3DIS data, please see its README.

To prepare ScanNet data, please see its README.

To prepare SUN RGB-D data, please see its README.

5.2.6 Customized Datasets

For using custom datasets, please refer to Tutorials 2: Customize Datasets.

CHAPTER

SIX

1: INFERENCE AND TRAIN WITH EXISTING MODELS AND STANDARD DATASETS

6.1 Inference with existing models

Here we provide testing scripts to evaluate a whole dataset (SUNRGBD, ScanNet, KITTI, etc.).

For high-level apis easier to integrated into other projects and basic demos, please refer to Verification/Demo under Get Started.

6.1.1 Test existing models on standard datasets

- single GPU
- CPU
- single node multiple GPU
- multiple node

You can use the following commands to test a dataset.

```
# single-gpu testing
python tools/test.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [--out ${RESULT_FILE}] [--eval $
↪{EVAL_METRICS}] [--show] [--show-dir ${SHOW_DIR}]

# CPU: disable GPUs and run single-gpu testing script (experimental)
export CUDA_VISIBLE_DEVICES=-1
python tools/test.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [--out ${RESULT_FILE}] [--eval $
↪{EVAL_METRICS}] [--show] [--show-dir ${SHOW_DIR}]

# multi-gpu testing
./tools/dist_test.sh ${CONFIG_FILE} ${CHECKPOINT_FILE} ${GPU_NUM} [--out ${RESULT_FILE}]␣
↪[--eval ${EVAL_METRICS}]
```

Note:

For now, CPU testing is only supported for SMOKE.

Optional arguments:

- `RESULT_FILE`: Filename of the output results in pickle format. If not specified, the results will not be saved to a file.

- `EVAL_METRICS`: Items to be evaluated on the results. Allowed values depend on the dataset. Typically we default to use official metrics for evaluation on different datasets, so it can be simply set to `mAP` as a placeholder for detection tasks, which applies to nuScenes, Lyft, ScanNet and SUNRGBD. For KITTI, if we only want to evaluate the 2D detection performance, we can simply set the metric to `img_bbox` (unstable, stay tuned). For Waymo, we provide both KITTI-style evaluation (unstable) and Waymo-style official protocol, corresponding to metric `kitti` and `waymo` respectively. We recommend to use the default official metric for stable performance and fair comparison with other methods. Similarly, the metric can be set to `mIoU` for segmentation tasks, which applies to S3DIS and ScanNet.
- `--show`: If specified, detection results will be plotted in the silient mode. It is only applicable to single GPU testing and used for debugging and visualization. This should be used with `--show-dir`.
- `--show-dir`: If specified, detection results will be plotted on the `***_points.obj` and `***_pred.obj` files in the specified directory. It is only applicable to single GPU testing and used for debugging and visualization. You do NOT need a GUI available in your environment for using this option.

Examples:

Assume that you have already downloaded the checkpoints to the directory `checkpoints/`.

1. Test VoteNet on ScanNet and save the points and prediction visualization results.

```
python tools/test.py configs/votenet/votenet_8x8_scannet-3d-18class.py \
    checkpoints/votenet_8x8_scannet-3d-18class_20200620_230238-2cea9c3a.pth \
    --show --show-dir ./data/scannet/show_results
```

2. Test VoteNet on ScanNet, save the points, prediction, groundtruth visualization results, and evaluate the mAP.

```
python tools/test.py configs/votenet/votenet_8x8_scannet-3d-18class.py \
    checkpoints/votenet_8x8_scannet-3d-18class_20200620_230238-2cea9c3a.pth \
    --eval mAP
    --eval-options 'show=True' 'out_dir=./data/scannet/show_results'
```

3. Test VoteNet on ScanNet (without saving the test results) and evaluate the mAP.

```
python tools/test.py configs/votenet/votenet_8x8_scannet-3d-18class.py \
    checkpoints/votenet_8x8_scannet-3d-18class_20200620_230238-2cea9c3a.pth \
    --eval mAP
```

4. Test SECOND on KITTI with 8 GPUs, and evaluate the mAP.

```
./tools/slurm_test.sh ${PARTITION} ${JOB_NAME} configs/second/hv_second_secfpn_6x8_
↪80e_kitti-3d-3class.py \
    checkpoints/hv_second_secfpn_6x8_80e_kitti-3d-3class_20200620_230238-9208083a.
↪pth \
    --out results.pkl --eval mAP
```

5. Test PointPillars on nuScenes with 8 GPUs, and generate the json file to be submit to the official evaluation server.

```
./tools/slurm_test.sh ${PARTITION} ${JOB_NAME} configs/pointpillars/hv_pointpillars_
↪fpn_sbn-all_4x8_2x_nus-3d.py \
    checkpoints/hv_pointpillars_fpn_sbn-all_4x8_2x_nus-3d_20200620_230405-2fa62f3d.
↪pth \
    --format-only --eval-options 'jsonfile_prefix=./pointpillars_nuscenes_results'
```

   The generated results be under `./pointpillars_nuscenes_results` directory.

6. Test SECOND on KITTI with 8 GPUs, and generate the pkl files and submission data to be submit to the official evaluation server.

```
./tools/slurm_test.sh ${PARTITION} ${JOB_NAME} configs/second/hv_second_secfpn_6x8_
↪80e_kitti-3d-3class.py \
    checkpoints/hv_second_secfpn_6x8_80e_kitti-3d-3class_20200620_230238-9208083a.
↪pth \
    --format-only --eval-options 'pklfile_prefix=./second_kitti_results'
↪'submission_prefix=./second_kitti_results'
```

   The generated results be under `./second_kitti_results` directory.

7. Test PointPillars on Lyft with 8 GPUs, generate the pkl files and make a submission to the leaderboard.

```
./tools/slurm_test.sh ${PARTITION} ${JOB_NAME} configs/pointpillars/hv_pointpillars_
↪fpn_sbn-2x8_2x_lyft-3d.py \
    checkpoints/hv_pointpillars_fpn_sbn-2x8_2x_lyft-3d_latest.pth --out results/pp_
↪lyft/results_challenge.pkl \
    --format-only --eval-options 'jsonfile_prefix=results/pp_lyft/results_challenge
↪' \
    'csv_savepath=results/pp_lyft/results_challenge.csv'
```

   Notice: To generate submissions on Lyft, `csv_savepath` must be given in the `--eval-options`. After generating the csv file, you can make a submission with kaggle commands given on the website.

   Note that in the config of Lyft dataset, the value of `ann_file` keyword in `test` is `data_root + 'lyft_infos_test.pkl'`, which is the official test set of Lyft without annotation. To test on the validation set, please change this to `data_root + 'lyft_infos_val.pkl'`.

8. Test PointPillars on waymo with 8 GPUs, and evaluate the mAP with waymo metrics.

```
./tools/slurm_test.sh ${PARTITION} ${JOB_NAME} configs/pointpillars/hv_pointpillars_
↪secfpn_sbn-2x16_2x_waymo-3d-car.py \
    checkpoints/hv_pointpillars_secfpn_sbn-2x16_2x_waymo-3d-car_latest.pth --out␣
↪results/waymo-car/results_eval.pkl \
    --eval waymo --eval-options 'pklfile_prefix=results/waymo-car/kitti_results' \
    'submission_prefix=results/waymo-car/kitti_results'
```

   Notice: For evaluation on waymo, please follow the instruction to build the binary file `compute_detection_metrics_main` for metrics computation and put it into `mmdet3d/core/evaluation/waymo_utils/`.(Sometimes when using bazel to build `compute_detection_metrics_main`, an error `'round' is not a member of 'std'` may appear. We just need to remove the `std::` before `round` in that file.) `pklfile_prefix` should be given in the `--eval-options` for the bin file generation. For metrics, `waymo` is the recommended official evaluation prototype. Currently, evaluating with choice `kitti` is adapted from KITTI and the results for each difficulty are not exactly the same as the definition of KITTI. Instead, most of objects are marked with difficulty 0 currently, which will be fixed in the future. The reasons of its instability include the large computation for evaluation, the lack of occlusion and truncation in the converted data, different definition of difficulty and different methods of computing average precision.

9. Test PointPillars on waymo with 8 GPUs, generate the bin files and make a submission to the leaderboard.

```
./tools/slurm_test.sh ${PARTITION} ${JOB_NAME} configs/pointpillars/hv_pointpillars_
↪secfpn_sbn-2x16_2x_waymo-3d-car.py \
    checkpoints/hv_pointpillars_secfpn_sbn-2x16_2x_waymo-3d-car_latest.pth --out␣
↪results/waymo-car/results_eval.pkl \
    --format-only --eval-options 'pklfile_prefix=results/waymo-car/kitti_results' \
```

(continues on next page)

(continued from previous page)

```
    'submission_prefix=results/waymo-car/kitti_results'
```

Notice: After generating the bin file, you can simply build the binary file `create_submission` and use them to create a submission file by following the instruction. For evaluation on the validation set with the eval server, you can also use the same way to generate a submission.

6.2 Train predefined models on standard datasets

MMDetection3D implements distributed training and non-distributed training, which uses `MMDistributedDataParallel` and `MMDataParallel` respectively.

All outputs (log files and checkpoints) will be saved to the working directory, which is specified by `work_dir` in the config file.

By default we evaluate the model on the validation set after each epoch, you can change the evaluation interval by adding the interval argument in the training config.

```
evaluation = dict(interval=12)  # This evaluate the model per 12 epoch.
```

Important: The default learning rate in config files is for 8 GPUs and the exact batch size is marked by the config's file name, e.g. '2x8' means 2 samples per GPU using 8 GPUs. According to the Linear Scaling Rule, you need to set the learning rate proportional to the batch size if you use different GPUs or images per GPU, e.g., lr=0.01 for 4 GPUs * 2 img/gpu and lr=0.08 for 16 GPUs * 4 img/gpu. However, since most of the models in this repo use ADAM rather than SGD for optimization, the rule may not hold and users need to tune the learning rate by themselves.

6.2.1 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}`.

6.2.2 Training with CPU (experimental)

The process of training on the CPU is consistent with single GPU training. We just need to disable GPUs before the training process.

```
export CUDA_VISIBLE_DEVICES=-1
```

And then run the script of train with a single GPU.

Note:

For now, most of the point cloud related algorithms rely on 3D CUDA op, which can not be trained on CPU. Some monocular 3D object detection algorithms, like FCOS3D and SMOKE can be trained on CPU. We do not recommend users to use CPU for training because it is too slow. We support this feature to allow users to debug certain models on machines without GPU for convenience.

6.2.3 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 at every k (default value is 1, which can be modified like this) epochs 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.
- `--options 'Key=value'`: Override some settings in the used config.

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.

6.2.4 Train with multiple machines

If you run MMDetection3D 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}
```

Here is an example of using 16 GPUs to train Mask R-CNN on the dev partition.

```
GPUS=16 ./tools/slurm_train.sh dev pp_kitti_3class hv_pointpillars_secfpn_6x8_160e_kitti-
↪3d-3class.py /nfs/xxxx/pp_kitti_3class
```

You can check slurm_train.sh for full arguments and environment variables.

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 ./tools/dist_train.sh
↪$CONFIG $GPUS
```

On the second machine:

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

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

6.2.5 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, there are two ways to specify the ports.

1. Set the port through `--options`. This is more recommended since it does not change the original configs.

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

2. 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

SEVEN

2: TRAIN WITH CUSTOMIZED DATASETS

In this note, you will know how to train and test predefined models with customized datasets. We use the Waymo dataset as an example to describe the whole process.

The basic steps are as below:

1. Prepare the customized dataset
2. Prepare a config
3. Train, test, inference models on the customized dataset.

7.1 Prepare the customized dataset

There are three ways to support a new dataset in MMDetection3D:

1. reorganize the dataset into existing format.
2. reorganize the dataset into a middle format.
3. implement a new dataset.

Usually we recommend to use the first two methods which are usually easier than the third.

In this note, we give an example for converting the data into KITTI format.

Note: We take Waymo as the example here considering its format is totally different from other existing formats. For other datasets using similar methods to organize data, like Lyft compared to nuScenes, it would be easier to directly implement the new data converter (for the second approach above) instead of converting it to another format (for the first approach above).

7.1.1 KITTI dataset format

Firstly, the raw data for 3D object detection from KITTI are typically organized as follows, where `ImageSets` contains split files indicating which files belong to training/validation/testing set, `calib` contains calibration information files, `image_2` and `velodyne` include image data and point cloud data, and `label_2` includes label files for 3D detection.

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
│   ├── kitti
│   │   ├── ImageSets
```

(continues on next page)

(continued from previous page)

```
│   │   ├── testing
│   │   │   ├── calib
│   │   │   ├── image_2
│   │   │   ├── velodyne
│   │   ├── training
│   │   │   ├── calib
│   │   │   ├── image_2
│   │   │   ├── label_2
│   │   │   ├── velodyne
```

Specific annotation format is described in the official object development kit. For example, it consists of the following labels:

```
#Values    Name      Description
----------------------------------------------------------------------------
   1    type         Describes the type of object: 'Car', 'Van', 'Truck',
                     'Pedestrian', 'Person_sitting', 'Cyclist', 'Tram',
                     'Misc' or 'DontCare'
   1    truncated    Float from 0 (non-truncated) to 1 (truncated), where
                     truncated refers to the object leaving image boundaries
   1    occluded     Integer (0,1,2,3) indicating occlusion state:
                     0 = fully visible, 1 = partly occluded
                     2 = largely occluded, 3 = unknown
   1    alpha        Observation angle of object, ranging [-pi..pi]
   4    bbox         2D bounding box of object in the image (0-based index):
                     contains left, top, right, bottom pixel coordinates
   3    dimensions   3D object dimensions: height, width, length (in meters)
   3    location     3D object location x,y,z in camera coordinates (in meters)
   1    rotation_y   Rotation ry around Y-axis in camera coordinates [-pi..pi]
   1    score        Only for results: Float, indicating confidence in
                     detection, needed for p/r curves, higher is better.
```

Assume we use the Waymo dataset. After downloading the data, we need to implement a function to convert both the input data and annotation format into the KITTI style. Then we can implement WaymoDataset inherited from KittiDataset to load the data and perform training and evaluation.

Specifically, we implement a waymo converter to convert Waymo data into KITTI format and a waymo dataset class to process it. Because we preprocess the raw data and reorganize it like KITTI, the dataset class could be implemented more easily by inheriting from KittiDataset. The last thing needed to be noted is the evaluation protocol you would like to use. Because Waymo has its own evaluation approach, we further incorporate it into our dataset class. Afterwards, users can successfully convert the data format and use `WaymoDataset` to train and evaluate the model.

For more details about the intermediate results of preprocessing of Waymo dataset, please refer to its tutorial.

7.2 Prepare a config

The second step is to prepare configs such that the dataset could be successfully loaded. In addition, adjusting hyper-parameters is usually necessary to obtain decent performance in 3D detection.

Suppose we would like to train PointPillars on Waymo to achieve 3D detection for 3 classes, vehicle, cyclist and pedestrian, we need to prepare dataset config like this, model config like this and combine them like this, compared to KITTI dataset config, model config and overall.

7.3 Train a new model

To train a model with the new config, you can simply run

```
python tools/train.py configs/pointpillars/hv_pointpillars_secfpn_sbn_2x16_2x_waymoD5-3d-
↪3class.py
```

For more detailed usages, please refer to the Case 1.

7.4 Test and inference

To test the trained model, you can simply run

```
python tools/test.py configs/pointpillars/hv_pointpillars_secfpn_sbn_2x16_2x_waymoD5-3d-
↪3class.py work_dirs/hv_pointpillars_secfpn_sbn_2x16_2x_waymoD5-3d-3class/latest.pth --
↪eval waymo
```

Note: To use Waymo evaluation protocol, you need to follow the tutorial and prepare files related to metrics computation as official instructions.

For more detailed usages for test and inference, please refer to the Case 1.

CHAPTER

EIGHT

LIDAR-BASED 3D DETECTION

LiDAR-based 3D detection is one of the most basic tasks supported in MMDetection3D. It expects the given model to take any number of points with features collected by LiDAR as input, and predict the 3D bounding boxes and category labels for each object of interest. Next, taking PointPillars on the KITTI dataset as an example, we will show how to prepare data, train and test a model on a standard 3D detection benchmark, and how to visualize and validate the results.

8.1 Data Preparation

To begin with, we need to download the raw data and reorganize the data in a standard way presented in the doc for data preparation. Note that for KITTI, we need extra txt files for data splits.

Due to different ways of organizing the raw data in different datasets, we typically need to collect the useful data information with a .pkl or .json file. So after getting all the raw data ready, we need to run the scripts provided in the `create_data.py` for different datasets to generate data infos. For example, for KITTI we need to run:

```
python tools/create_data.py kitti --root-path ./data/kitti --out-dir ./data/kitti --
↪extra-tag kitti
```

Afterwards, the related folder structure should be as follows:

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
    ├── kitti
        ├── ImageSets
        ├── testing
            ├── calib
            ├── image_2
            ├── velodyne
        ├── training
            ├── calib
            ├── image_2
            ├── label_2
            ├── velodyne
        ├── kitti_gt_database
        ├── kitti_infos_train.pkl
        ├── kitti_infos_trainval.pkl
        ├── kitti_infos_val.pkl
```

(continues on next page)

(continued from previous page)

```
│   │   ├── kitti_infos_test.pkl
│   │   ├── kitti_dbinfos_train.pkl
```

8.2 Training

Then let us train a model with provided configs for PointPillars. You can basically follow this tutorial for sample scripts when training with different GPU settings. Suppose we use 8 GPUs on a single machine with distributed training:

```
./tools/dist_train.sh configs/pointpillars/hv_pointpillars_secfpn_6x8_160e_kitti-3d-
↪3class.py 8
```

Note that `6x8` in the config name refers to the training is completed with 8 GPUs and 6 samples on each GPU. If your customized setting is different from this, sometimes you need to adjust the learning rate accordingly. A basic rule can be referred to here.

8.3 Quantitative Evaluation

During training, the model checkpoints will be evaluated regularly according to the setting of `evaluation = dict(interval=xxx)` in the config. We support official evaluation protocols for different datasets. For KITTI, the model will be evaluated with mean average precision (mAP) with Intersection over Union (IoU) thresholds 0.5/0.7 for 3 categories respectively. The evaluation results will be printed in the command like:

```
Car AP@0.70, 0.70, 0.70:
bbox AP:98.1839, 89.7606, 88.7837
bev AP:89.6905, 87.4570, 85.4865
3d AP:87.4561, 76.7569, 74.1302
aos AP:97.70, 88.73, 87.34
Car AP@0.70, 0.50, 0.50:
bbox AP:98.1839, 89.7606, 88.7837
bev AP:98.4400, 90.1218, 89.6270
3d AP:98.3329, 90.0209, 89.4035
aos AP:97.70, 88.73, 87.34
```

In addition, you can also evaluate a specific model checkpoint after training is finished. Simply run scripts like the following:

```
./tools/dist_test.sh configs/pointpillars/hv_pointpillars_secfpn_6x8_160e_kitti-3d-
↪3class.py \
    work_dirs/pointpillars/latest.pth --eval mAP
```

8.4 Testing and Making a Submission

If you would like to only conduct inference or test the model performance on the online benchmark, you just need to replace the `--eval mAP` with `--format-only` in the previous evaluation script and specify the `pklfile_prefix` and `submission_prefix` if necessary, e.g., adding an option `--eval-options submission_prefix=work_dirs/pointpillars/test_submission`. Please guarantee the info for testing in the config corresponds to the test set instead of validation set. After generating the results, you can basically compress the folder and upload to the KITTI evaluation server.

8.5 Qualitative Validation

MMDetection3D also provides versatile tools for visualization such that we can have an intuitive feeling of the detection results predicted by our trained models. You can either set the `--eval-options 'show=True' 'out_dir=${SHOW_DIR}'` option to visualize the detection results online during evaluation, or using `tools/misc/visualize_results.py` for offline visualization. Besides, we also provide scripts `tools/misc/browse_dataset.py` to visualize the dataset without inference. Please refer more details in the doc for visualization.

CHAPTER

NINE

VISION-BASED 3D DETECTION

Vision-based 3D detection refers to the 3D detection solutions based on vision-only input, such as monocular, binocular, and multi-view image based 3D detection. Currently, we only support monocular and multi-view 3D detection methods. Other approaches should be also compatible with our framework and will be supported in the future.

It expects the given model to take any number of images as input, and predict the 3D bounding boxes and category labels for each object of interest. Taking FCOS3D on the nuScenes dataset as an example, we will show how to prepare data, train and test a model on a standard 3D detection benchmark, and how to visualize and validate the results.

9.1 Data Preparation

To begin with, we need to download the raw data and reorganize the data in a standard way presented in the doc for data preparation.

Due to different ways of organizing the raw data in different datasets, we typically need to collect the useful data information with a .pkl or .json file. So after getting all the raw data ready, we need to run the scripts provided in the `create_data.py` for different datasets to generate data infos. For example, for nuScenes we need to run:

```
python tools/create_data.py nuscenes --root-path ./data/nuscenes --out-dir ./data/
↪nuscenes --extra-tag nuscenes
```

Afterwards, the related folder structure should be as follows:

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
│   ├── nuscenes
│   │   ├── maps
│   │   ├── samples
│   │   ├── sweeps
│   │   ├── v1.0-test
|   |   ├── v1.0-trainval
│   │   ├── nuscenes_database
│   │   ├── nuscenes_infos_train.pkl
│   │   ├── nuscenes_infos_trainval.pkl
│   │   ├── nuscenes_infos_val.pkl
│   │   ├── nuscenes_infos_test.pkl
│   │   ├── nuscenes_dbinfos_train.pkl
│   │   ├── nuscenes_infos_train_mono3d.coco.json
```

(continues on next page)

(continued from previous page)

```
│   │   ├── nuscenes_infos_trainval_mono3d.coco.json
│   │   ├── nuscenes_infos_val_mono3d.coco.json
│   │   ├── nuscenes_infos_test_mono3d.coco.json
```

Note that the .pkl files here are mainly used for methods using LiDAR data and .json files are used for 2D detection/vision-only 3D detection. The .json files only contain infos for 2D detection before supporting monocular 3D detection in v0.13.0, so if you need the latest infos, please checkout the branches after v0.13.0.

9.2 Training

Then let us train a model with provided configs for FCOS3D. The basic script is the same as other models. You can basically follow the examples provided in this tutorial when training with different GPU settings. Suppose we use 8 GPUs on a single machine with distributed training:

```
./tools/dist_train.sh configs/fcos3d/fcos3d_r101_caffe_fpn_gn-head_dcn_2x8_1x_nus-mono3d.
↪py 8
```

Note that `2x8` in the config name refers to the training is completed with 8 GPUs and 2 data samples on each GPU. If your customized setting is different from this, sometimes you need to adjust the learning rate accordingly. A basic rule can be referred to here.

We can also achieve better performance with finetuned FCOS3D by running:

```
./tools/dist_train.sh fcos3d_r101_caffe_fpn_gn-head_dcn_2x8_1x_nus-mono3d_finetune.py 8
```

After training a baseline model with the previous script, please remember to modify the path here correspondingly.

9.3 Quantitative Evaluation

During training, the model checkpoints will be evaluated regularly according to the setting of `evaluation = dict(interval=xxx)` in the config.

We support official evaluation protocols for different datasets. Due to the output format is the same as 3D detection based on other modalities, the evaluation methods are also the same.

For nuScenes, the model will be evaluated with distance-based mean AP (mAP) and NuScenes Detection Score (NDS) for 10 categories respectively. The evaluation results will be printed in the command like:

```
mAP: 0.3197
mATE: 0.7595
mASE: 0.2700
mAOE: 0.4918
mAVE: 1.3307
mAAE: 0.1724
NDS: 0.3905
Eval time: 170.8s

Per-class results:
Object Class    AP      ATE     ASE     AOE     AVE     AAE
car     0.503   0.577   0.152   0.111   2.096   0.136
truck   0.223   0.857   0.224   0.220   1.389   0.179
```

(continues on next page)

(continued from previous page)

```
bus     0.294   0.855   0.204   0.190   2.689   0.283
trailer 0.081   1.094   0.243   0.553   0.742   0.167
construction_vehicle    0.058   1.017   0.450   1.019   0.137   0.341
pedestrian      0.392   0.687   0.284   0.694   0.876   0.158
motorcycle      0.317   0.737   0.265   0.580   2.033   0.104
bicycle 0.308   0.704   0.299   0.892   0.683   0.010
traffic_cone    0.555   0.486   0.309   nan     nan     nan
barrier 0.466   0.581   0.269   0.169   nan     nan
```

In addition, you can also evaluate a specific model checkpoint after training is finished. Simply run scripts like the following:

```
./tools/dist_test.sh configs/fcos3d/fcos3d_r101_caffe_fpn_gn-head_dcn_2x8_1x_nus-mono3d.
↪py \
    work_dirs/fcos3d/latest.pth --eval mAP
```

9.4 Testing and Making a Submission

If you would like to only conduct inference or test the model performance on the online benchmark, you just need to replace the `--eval mAP` with `--format-only` in the previous evaluation script and specify the `jsonfile_prefix` if necessary, e.g., adding an option `--eval-options jsonfile_prefix=work_dirs/fcos3d/test_submission`. Please guarantee the info for testing in the config corresponds to the test set instead of validation set.

After generating the results, you can basically compress the folder and upload to the evalAI evaluation server for nuScenes 3D detection challenge.

9.5 Qualitative Validation

MMDetection3D also provides versatile tools for visualization such that we can have an intuitive feeling of the detection results predicted by our trained models. You can either set the `--eval-options 'show=True' 'out_dir=${SHOW_DIR}'` option to visualize the detection results online during evaluation, or using `tools/misc/visualize_results.py` for offline visualization.

Besides, we also provide scripts `tools/misc/browse_dataset.py` to visualize the dataset without inference. Please refer more details in the doc for visualization.

Note that currently we only support the visualization on images for vision-only methods. The visualization in the perspective view and bird-eye-view (BEV) will be integrated in the future.

CHAPTER

TEN

LIDAR-BASED 3D SEMANTIC SEGMENTATION

LiDAR-based 3D semantic segmentation is one of the most basic tasks supported in MMDetection3D. It expects the given model to take any number of points with features collected by LiDAR as input, and predict the semantic labels for each input point. Next, taking PointNet++ (SSG) on the ScanNet dataset as an example, we will show how to prepare data, train and test a model on a standard 3D semantic segmentation benchmark, and how to visualize and validate the results.

10.1 Data Preparation

To begin with, we need to download the raw data from ScanNet's official website.

Due to different ways of organizing the raw data in different datasets, we typically need to collect the useful data information with a .pkl or .json file.

So after getting all the raw data ready, we can follow the instructions presented in ScanNet README doc to generate data infos.

Afterwards, the related folder structure should be as follows:

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
│   ├── scannet
│   │   ├── scannet_utils.py
│   │   ├── batch_load_scannet_data.py
│   │   ├── load_scannet_data.py
│   │   ├── scannet_utils.py
│   │   ├── README.md
│   │   ├── scans
│   │   ├── scans_test
│   │   ├── scannet_instance_data
│   │   ├── points
│   │   ├── instance_mask
│   │   ├── semantic_mask
│   │   ├── seg_info
│   │   │   ├── train_label_weight.npy
│   │   │   ├── train_resampled_scene_idxs.npy
│   │   │   ├── val_label_weight.npy
│   │   │   ├── val_resampled_scene_idxs.npy
```

(continues on next page)

(continued from previous page)

```
│   │   ├── scannet_infos_train.pkl
│   │   ├── scannet_infos_val.pkl
│   │   ├── scannet_infos_test.pkl
```

10.2 Training

Then let us train a model with provided configs for PointNet++ (SSG). You can basically follow this tutorial for sample scripts when training with different GPU settings. Suppose we use 2 GPUs on a single machine with distributed training:

```
./tools/dist_train.sh configs/pointnet2/pointnet2_ssg_16x2_cosine_200e_scannet_seg-3d-
↪20class.py 2
```

Note that `16x2` in the config name refers to the training is completed with 2 GPUs and 16 samples on each GPU. If your customized setting is different from this, sometimes you need to adjust the learning rate accordingly. A basic rule can be referred to here.

10.3 Quantitative Evaluation

During training, the model checkpoints will be evaluated regularly according to the setting of `evaluation = dict(interval=xxx)` in the config. We support official evaluation protocols for different datasets. For ScanNet, the model will be evaluated with mean Intersection over Union (mIoU) over all 20 categories. The evaluation results will be printed in the command like:

```
+---------+--------+--------+---------+--------+--------+--------+--------+--------+-----
↪---+----------+---------+---------+--------+---------+-------------+----------------
↪+--------+--------+---------+----------------+--------+--------+---------+
| classes | wall   | floor  | cabinet | bed    | chair  | sofa   | table  | door   |␣
↪window | bookshelf | picture | counter | desk   | curtain | refrigerator |␣
↪showercurtrain | toilet | sink   | bathtub | otherfurniture | miou   | acc    | acc_
↪cls |
+---------+--------+--------+---------+--------+--------+--------+--------+--------+-----
↪---+----------+---------+---------+--------+---------+-------------+----------------
↪+--------+--------+---------+----------------+--------+--------+---------+
| results | 0.7257 | 0.9373 | 0.4625  | 0.6613 | 0.7707 | 0.5562 | 0.5864 | 0.4010 | 0.
↪4558 | 0.7011    | 0.2500  | 0.4645  | 0.4540 | 0.5399  | 0.2802       | 0.3488       ␣
↪ | 0.7359 | 0.4971 | 0.6922  | 0.3681         | 0.5444 | 0.8118 | 0.6695  |
+---------+--------+--------+---------+--------+--------+--------+--------+--------+-----
↪---+----------+---------+---------+--------+---------+-------------+----------------
↪+--------+--------+---------+----------------+--------+--------+---------+
```

In addition, you can also evaluate a specific model checkpoint after training is finished. Simply run scripts like the following:

```
./tools/dist_test.sh configs/pointnet2/pointnet2_ssg_16x2_cosine_200e_scannet_seg-3d-
↪20class.py \
    work_dirs/pointnet2_ssg/latest.pth --eval mIoU
```

10.4 Testing and Making a Submission

If you would like to only conduct inference or test the model performance on the online benchmark, you need to replace the `--eval mIoU` with `--format-only` in the previous evaluation script and change `ann_file=data_root + 'scannet_infos_val.pkl'` to `ann_file=data_root + 'scannet_infos_test.pkl'` in the ScanNet dataset's config. Remember to specify the `txt_prefix` as the directory to save the testing results, e.g., adding an option `--eval-options txt_prefix=work_dirs/pointnet2_ssg/test_submission`. After generating the results, you can basically compress the folder and upload to the ScanNet evaluation server.

10.5 Qualitative Validation

MMDetection3D also provides versatile tools for visualization such that we can have an intuitive feeling of the segmentation results predicted by our trained models. You can either set the `--eval-options 'show=True' 'out_dir=${SHOW_DIR}'` option to visualize the segmentation results online during evaluation, or using `tools/misc/visualize_results.py` for offline visualization. Besides, we also provide scripts `tools/misc/browse_dataset.py` to visualize the dataset without inference. Please refer more details in the doc for visualization.

CHAPTER

ELEVEN

KITTI DATASET FOR 3D OBJECT DETECTION

This page provides specific tutorials about the usage of MMDetection3D for KITTI dataset.

Note: Current tutorial is only for LiDAR-based and multi-modality 3D detection methods. Contents related to monocular methods will be supplemented afterwards.

11.1 Prepare dataset

You can download KITTI 3D detection data HERE and unzip all zip files. Besides, the road planes could be downloaded from HERE, which are optional for data augmentation during training for better performance. The road planes are generated by AVOD, you can see more details HERE.

Like the general way to prepare dataset, it is recommended to symlink the dataset root to `$MMDETECTION3D/data`.

The folder structure should be organized as follows before our processing.

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
    ├── kitti
        ├── ImageSets
        ├── testing
            ├── calib
            ├── image_2
            ├── velodyne
        ├── training
            ├── calib
            ├── image_2
            ├── label_2
            ├── velodyne
            ├── planes (optional)
```

11.1.1 Create KITTI dataset

To create KITTI point cloud data, we load the raw point cloud data and generate the relevant annotations including object labels and bounding boxes. We also generate all single training objects' point cloud in KITTI dataset and save them as `.bin` files in `data/kitti/kitti_gt_database`. Meanwhile, `.pkl` info files are also generated for training or validation. Subsequently, create KITTI data by running

```
mkdir ./data/kitti/ && mkdir ./data/kitti/ImageSets

# Download data split
wget -c  https://raw.githubusercontent.com/traveller59/second.pytorch/master/second/data/
↪ImageSets/test.txt --no-check-certificate --content-disposition -O ./data/kitti/
↪ImageSets/test.txt
wget -c  https://raw.githubusercontent.com/traveller59/second.pytorch/master/second/data/
↪ImageSets/train.txt --no-check-certificate --content-disposition -O ./data/kitti/
↪ImageSets/train.txt
wget -c  https://raw.githubusercontent.com/traveller59/second.pytorch/master/second/data/
↪ImageSets/val.txt --no-check-certificate --content-disposition -O ./data/kitti/
↪ImageSets/val.txt
wget -c  https://raw.githubusercontent.com/traveller59/second.pytorch/master/second/data/
↪ImageSets/trainval.txt --no-check-certificate --content-disposition -O ./data/kitti/
↪ImageSets/trainval.txt


python tools/create_data.py kitti --root-path ./data/kitti --out-dir ./data/kitti --
↪extra-tag kitti --with-plane
```

Note that if your local disk does not have enough space for saving converted data, you can change the `out-dir` to anywhere else, and you need to remove the `--with-plane` flag if `planes` are not prepared.

The folder structure after processing should be as below

```
kitti
├── ImageSets
│   ├── test.txt
│   ├── train.txt
│   ├── trainval.txt
│   ├── val.txt
├── testing
│   ├── calib
│   ├── image_2
│   ├── velodyne
│   ├── velodyne_reduced
├── training
│   ├── calib
│   ├── image_2
│   ├── label_2
│   ├── velodyne
│   ├── velodyne_reduced
│   ├── planes (optional)
├── kitti_gt_database
│   ├── xxxxx.bin
├── kitti_infos_train.pkl
├── kitti_infos_val.pkl
```

(continues on next page)

(continued from previous page)

```
├── kitti_dbinfos_train.pkl
├── kitti_infos_test.pkl
├── kitti_infos_trainval.pkl
├── kitti_infos_train_mono3d.coco.json
├── kitti_infos_trainval_mono3d.coco.json
├── kitti_infos_test_mono3d.coco.json
├── kitti_infos_val_mono3d.coco.json
```

- `kitti_gt_database/xxxxx.bin`: point cloud data included in each 3D bounding box of the training dataset
- `kitti_infos_train.pkl`: training dataset infos, each frame info contains following details:
  - info['point_cloud']: {'num_features': 4, 'velodyne_path': velodyne_path}.
  - info['annos']: {
    - location: x,y,z are bottom center in referenced camera coordinate system (in meters), an Nx3 array
    - dimensions: height, width, length (in meters), an Nx3 array
    - rotation_y: rotation ry around Y-axis in camera coordinates [-pi..pi], an N array
    - name: ground truth name array, an N array
    - difficulty: kitti difficulty, Easy, Moderate, Hard
    - group_ids: used for multi-part object }
  - (optional) info['calib']: {
    - P0: camera0 projection matrix after rectification, an 3x4 array
    - P1: camera1 projection matrix after rectification, an 3x4 array
    - P2: camera2 projection matrix after rectification, an 3x4 array
    - P3: camera3 projection matrix after rectification, an 3x4 array
    - R0_rect: rectifying rotation matrix, an 4x4 array
    - Tr_velo_to_cam: transformation from Velodyne coordinate to camera coordinate, an 4x4 array
    - Tr_imu_to_velo: transformation from IMU coordinate to Velodyne coordinate, an 4x4 array }
  - (optional) info['image']:{'image_idx': idx, 'image_path': image_path, 'image_shape', image_shape}.

Note: the info['annos'] is in the referenced camera coordinate system. More details please refer to this

The core function to get kitti_infos_xxx.pkl and kitti_infos_xxx_mono3d.coco.json are get_kitti_image_info and get_2d_boxes. Please refer to kitti_converter.py for more details.

11.2 Train pipeline

A typical train pipeline of 3D detection on KITTI is as below.

```
train_pipeline = [
    dict(
        type='LoadPointsFromFile',
        coord_type='LIDAR',
        load_dim=4, # x, y, z, intensity
```

(continues on next page)

(continued from previous page)

```
        use_dim=4, # x, y, z, intensity
        file_client_args=file_client_args),
    dict(
        type='LoadAnnotations3D',
        with_bbox_3d=True,
        with_label_3d=True,
        file_client_args=file_client_args),
    dict(type='ObjectSample', db_sampler=db_sampler),
    dict(
        type='ObjectNoise',
        num_try=100,
        translation_std=[1.0, 1.0, 0.5],
        global_rot_range=[0.0, 0.0],
        rot_range=[-0.78539816, 0.78539816]),
    dict(type='RandomFlip3D', flip_ratio_bev_horizontal=0.5),
    dict(
        type='GlobalRotScaleTrans',
        rot_range=[-0.78539816, 0.78539816],
        scale_ratio_range=[0.95, 1.05]),
    dict(type='PointsRangeFilter', point_cloud_range=point_cloud_range),
    dict(type='ObjectRangeFilter', point_cloud_range=point_cloud_range),
    dict(type='PointShuffle'),
    dict(type='DefaultFormatBundle3D', class_names=class_names),
    dict(type='Collect3D', keys=['points', 'gt_bboxes_3d', 'gt_labels_3d'])
]
```

- Data augmentation:
  - `ObjectNoise`: apply noise to each GT objects in the scene.
  - `RandomFlip3D`: randomly flip input point cloud horizontally or vertically.
  - `GlobalRotScaleTrans`: rotate input point cloud.

11.3 Evaluation

An example to evaluate PointPillars with 8 GPUs with kitti metrics is as follows:

```
bash tools/dist_test.sh configs/pointpillars/hv_pointpillars_secfpn_6x8_160e_kitti-3d-
↪3class.py work_dirs/hv_pointpillars_secfpn_6x8_160e_kitti-3d-3class/latest.pth 8 --
↪eval bbox
```

11.4 Metrics

KITTI evaluates 3D object detection performance using mean Average Precision (mAP) and Average Orientation Similarity (AOS), Please refer to its official website and original paper for more details.

We also adopt this approach for evaluation on KITTI. An example of printed evaluation results is as follows:

```
Car AP@0.70, 0.70, 0.70:
bbox AP:97.9252, 89.6183, 88.1564
bev  AP:90.4196, 87.9491, 85.1700
3d   AP:88.3891, 77.1624, 74.4654
aos  AP:97.70, 89.11, 87.38
Car AP@0.70, 0.50, 0.50:
bbox AP:97.9252, 89.6183, 88.1564
bev  AP:98.3509, 90.2042, 89.6102
3d   AP:98.2800, 90.1480, 89.4736
aos  AP:97.70, 89.11, 87.38
```

11.5 Testing and make a submission

An example to test PointPillars on KITTI with 8 GPUs and generate a submission to the leaderboard is as follows:

```
mkdir -p results/kitti-3class

./tools/dist_test.sh configs/pointpillars/hv_pointpillars_secfpn_6x8_160e_kitti-3d-
↪3class.py work_dirs/hv_pointpillars_secfpn_6x8_160e_kitti-3d-3class/latest.pth 8 --out␣
↪results/kitti-3class/results_eval.pkl --format-only --eval-options 'pklfile_
↪prefix=results/kitti-3class/kitti_results' 'submission_prefix=results/kitti-3class/
↪kitti_results'
```

After generating `results/kitti-3class/kitti_results/xxxxx.txt` files, you can submit these files to KITTI benchmark. Please refer to the KITTI official website for more details.

CHAPTER

TWELVE

NUSCENES DATASET FOR 3D OBJECT DETECTION

This page provides specific tutorials about the usage of MMDetection3D for nuScenes dataset.

12.1 Before Preparation

You can download nuScenes 3D detection data HERE and unzip all zip files.

Like the general way to prepare dataset, it is recommended to symlink the dataset root to `$MMDETECTION3D/data`.

The folder structure should be organized as follows before our processing.

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
│   ├── nuscenes
│   │   ├── maps
│   │   ├── samples
│   │   ├── sweeps
│   │   ├── v1.0-test
|   |   ├── v1.0-trainval
```

12.2 Dataset Preparation

We typically need to organize the useful data information with a .pkl or .json file in a specific style, e.g., coco-style for organizing images and their annotations. To prepare these files for nuScenes, run the following command:

```
python tools/create_data.py nuscenes --root-path ./data/nuscenes --out-dir ./data/
↪nuscenes --extra-tag nuscenes
```

The folder structure after processing should be as below.

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
│   ├── nuscenes
```

(continues on next page)

(continued from previous page)

```
│   │   ├── maps
│   │   ├── samples
│   │   ├── sweeps
│   │   ├── v1.0-test
│   │   ├── v1.0-trainval
│   │   ├── nuscenes_database
│   │   ├── nuscenes_infos_train.pkl
│   │   ├── nuscenes_infos_val.pkl
│   │   ├── nuscenes_infos_test.pkl
│   │   ├── nuscenes_dbinfos_train.pkl
│   │   ├── nuscenes_infos_train_mono3d.coco.json
│   │   ├── nuscenes_infos_val_mono3d.coco.json
│   │   ├── nuscenes_infos_test_mono3d.coco.json
```

Here, .pkl files are generally used for methods involving point clouds and coco-style .json files are more suitable for image-based methods, such as image-based 2D and 3D detection. Next, we will elaborate on the details recorded in these info files.

- `nuscenes_database/xxxxx.bin`: point cloud data included in each 3D bounding box of the training dataset
- `nuscenes_infos_train.pkl`: training dataset info, each frame info has two keys: `metadata` and `infos`. `metadata` contains the basic information for the dataset itself, such as `{'version': 'v1.0-trainval'}`, while `infos` contains the detailed information as follows:
  - info['lidar_path']: The file path of the lidar point cloud data.
  - info['token']: Sample data token.
  - info['sweeps']: Sweeps information (`sweeps` in the nuScenes refer to the intermediate frames without annotations, while `samples` refer to those key frames with annotations).
    - info['sweeps'][i]['data_path']: The data path of i-th sweep.
    - info['sweeps'][i]['type']: The sweep data type, e.g., `'lidar'`.
    - info['sweeps'][i]['sample_data_token']: The sweep sample data token.
    - info['sweeps'][i]['sensor2ego_translation']: The translation from the current sensor (for collecting the sweep data) to ego vehicle. (1x3 list)
    - info['sweeps'][i]['sensor2ego_rotation']: The rotation from the current sensor (for collecting the sweep data) to ego vehicle. (1x4 list in the quaternion format)
    - info['sweeps'][i]['ego2global_translation']: The translation from the ego vehicle to global coordinates. (1x3 list)
    - info['sweeps'][i]['ego2global_rotation']: The rotation from the ego vehicle to global coordinates. (1x4 list in the quaternion format)
    - info['sweeps'][i]['timestamp']: Timestamp of the sweep data.
    - info['sweeps'][i]['sensor2lidar_translation']: The translation from the current sensor (for collecting the sweep data) to lidar. (1x3 list)
    - info['sweeps'][i]['sensor2lidar_rotation']: The rotation from the current sensor (for collecting the sweep data) to lidar. (1x4 list in the quaternion format)
  - info['cams']: Cameras calibration information. It contains six keys corresponding to each camera: `'CAM_FRONT'`, `'CAM_FRONT_RIGHT'`, `'CAM_FRONT_LEFT'`, `'CAM_BACK'`, `'CAM_BACK_LEFT'`, `'CAM_BACK_RIGHT'`. Each dictionary contains detailed information following the above way for each

sweep data (has the same keys for each information as above). In addition, each camera has a key `'cam_intrinsic'` for recording the intrinsic parameters when projecting 3D points to each image plane.

  - info['lidar2ego_translation']: The translation from lidar to ego vehicle. (1x3 list)
  - info['lidar2ego_rotation']: The rotation from lidar to ego vehicle. (1x4 list in the quaternion format)
  - info['ego2global_translation']: The translation from the ego vehicle to global coordinates. (1x3 list)
  - info['ego2global_rotation']: The rotation from the ego vehicle to global coordinates. (1x4 list in the quaternion format)
  - info['timestamp']: Timestamp of the sample data.
  - info['gt_boxes']: 7-DoF annotations of 3D bounding boxes, an Nx7 array.
  - info['gt_names']: Categories of 3D bounding boxes, an 1xN array.
  - info['gt_velocity']: Velocities of 3D bounding boxes (no vertical measurements due to inaccuracy), an Nx2 array.
  - info['num_lidar_pts']: Number of lidar points included in each 3D bounding box.
  - info['num_radar_pts']: Number of radar points included in each 3D bounding box.
  - info['valid_flag']: Whether each bounding box is valid. In general, we only take the 3D boxes that include at least one lidar or radar point as valid boxes.

- `nuscenes_infos_train_mono3d.coco.json`: training dataset coco-style info. This file organizes image-based data into three categories (keys): `'categories'`, `'images'`, `'annotations'`.
  - info['categories']: A list containing all the category names. Each element follows the dictionary format and consists of two keys: `'id'` and `'name'`.
  - info['images']: A list containing all the image info.
    * info['images'][i]['file_name']: The file name of the i-th image.
    * info['images'][i]['id']: Sample data token of the i-th image.
    * info['images'][i]['token']: Sample token corresponding to this frame.
    * info['images'][i]['cam2ego_rotation']: The rotation from the camera to ego vehicle. (1x4 list in the quaternion format)
    * info['images'][i]['cam2ego_translation']: The translation from the camera to ego vehicle. (1x3 list)
    * info['images'][i]['ego2global_rotation"]: The rotation from the ego vehicle to global coordinates. (1x4 list in the quaternion format)
    * info['images'][i]['ego2global_translation']: The translation from the ego vehicle to global coordinates. (1x3 list)
    * info['images'][i]['cam_intrinsic']: Camera intrinsic matrix. (3x3 list)
    * info['images'][i]['width']: Image width, 1600 by default in nuScenes.
    * info['images'][i]['height']: Image height, 900 by default in nuScenes.
  - info['annotations']: A list containing all the annotation info.
    * info['annotations'][i]['file_name']: The file name of the corresponding image.
    * info['annotations'][i]['image_id']: The image id (token) of the corresponding image.
    * info['annotations'][i]['area']: Area of the 2D bounding box.
    * info['annotations'][i]['category_name']: Category name.

- info['annotations'][i]['category_id']: Category id.
- info['annotations'][i]['bbox']: 2D bounding box annotation (exterior rectangle of the projected 3D box), 1x4 list following [x1, y1, x2-x1, y2-y1]. x1/y1 are minimum coordinates along horizontal/vertical direction of the image.
- info['annotations'][i]['iscrowd']: Whether the region is crowded. Defaults to 0.
- info['annotations'][i]['bbox_cam3d']: 3D bounding box (gravity) center location (3), size (3), (global) yaw angle (1), 1x7 list.
- info['annotations'][i]['velo_cam3d']: Velocities of 3D bounding boxes (no vertical measurements due to inaccuracy), an Nx2 array.
- info['annotations'][i]['center2d']: Projected 3D-center containing 2.5D information: projected center location on the image (2) and depth (1), 1x3 list.
- info['annotations'][i]['attribute_name']: Attribute name.
- info['annotations'][i]['attribute_id']: Attribute id. We maintain a default attribute collection and mapping for attribute classification. Please refer to here for more details.
- info['annotations'][i]['id']: Annotation id. Defaults to `i`.

Here we only explain the data recorded in the training info files. The same applies to validation and testing set.

The core function to get `nuscenes_infos_xxx.pkl` and `nuscenes_infos_xxx_mono3d.coco.json` are _fill_trainval_infos and get_2d_boxes, respectively. Please refer to nuscenes_converter.py for more details.

12.3 Training pipeline

12.3.1 LiDAR-Based Methods

A typical training pipeline of LiDAR-based 3D detection (including multi-modality methods) on nuScenes is as below.

```
train_pipeline = [
    dict(
        type='LoadPointsFromFile',
        coord_type='LIDAR',
        load_dim=5,
        use_dim=5,
        file_client_args=file_client_args),
    dict(
        type='LoadPointsFromMultiSweeps',
        sweeps_num=10,
        file_client_args=file_client_args),
    dict(type='LoadAnnotations3D', with_bbox_3d=True, with_label_3d=True),
    dict(
        type='GlobalRotScaleTrans',
        rot_range=[-0.3925, 0.3925],
        scale_ratio_range=[0.95, 1.05],
        translation_std=[0, 0, 0]),
    dict(type='RandomFlip3D', flip_ratio_bev_horizontal=0.5),
    dict(type='PointsRangeFilter', point_cloud_range=point_cloud_range),
    dict(type='ObjectRangeFilter', point_cloud_range=point_cloud_range),
    dict(type='ObjectNameFilter', classes=class_names),
```

(continues on next page)

(continued from previous page)

```
    dict(type='PointShuffle'),
    dict(type='DefaultFormatBundle3D', class_names=class_names),
    dict(type='Collect3D', keys=['points', 'gt_bboxes_3d', 'gt_labels_3d'])
]
```

Compared to general cases, nuScenes has a specific `'LoadPointsFromMultiSweeps'` pipeline to load point clouds from consecutive frames. This is a common practice used in this setting. Please refer to the nuScenes original paper for more details. The default `use_dim` in `'LoadPointsFromMultiSweeps'` is `[0, 1, 2, 4]`, where the first 3 dimensions refer to point coordinates and the last refers to timestamp differences. Intensity is not used by default due to its yielded noise when concatenating the points from different frames.

12.3.2 Vision-Based Methods

A typical training pipeline of image-based 3D detection on nuScenes is as below.

```
train_pipeline = [
    dict(type='LoadImageFromFileMono3D'),
    dict(
        type='LoadAnnotations3D',
        with_bbox=True,
        with_label=True,
        with_attr_label=True,
        with_bbox_3d=True,
        with_label_3d=True,
        with_bbox_depth=True),
    dict(type='Resize', img_scale=(1600, 900), keep_ratio=True),
    dict(type='RandomFlip3D', flip_ratio_bev_horizontal=0.5),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='Pad', size_divisor=32),
    dict(type='DefaultFormatBundle3D', class_names=class_names),
    dict(
        type='Collect3D',
        keys=[
            'img', 'gt_bboxes', 'gt_labels', 'attr_labels', 'gt_bboxes_3d',
            'gt_labels_3d', 'centers2d', 'depths'
        ]),
]
```

It follows the general pipeline of 2D detection while differs in some details:

- It uses monocular pipelines to load images, which includes additional required information like camera intrinsics.
- It needs to load 3D annotations.
- Some data augmentation techniques need to be adjusted, such as `RandomFlip3D`. Currently we do not support more augmentation methods, because how to transfer and apply other techniques is still under explored.

12.4 Evaluation

An example to evaluate PointPillars with 8 GPUs with nuScenes metrics is as follows.

```
bash ./tools/dist_test.sh configs/pointpillars/hv_pointpillars_fpn_sbn-all_4x8_2x_nus-3d.
↪py checkpoints/hv_pointpillars_fpn_sbn-all_4x8_2x_nus-3d_20200620_230405-2fa62f3d.pth␣
↪8 --eval bbox
```

12.5 Metrics

NuScenes proposes a comprehensive metric, namely nuScenes detection score (NDS), to evaluate different methods and set up the benchmark. It consists of mean Average Precision (mAP), Average Translation Error (ATE), Average Scale Error (ASE), Average Orientation Error (AOE), Average Velocity Error (AVE) and Average Attribute Error (AAE). Please refer to its official website for more details.

We also adopt this approach for evaluation on nuScenes. An example of printed evaluation results is as follows:

```
mAP: 0.3197
mATE: 0.7595
mASE: 0.2700
mAOE: 0.4918
mAVE: 1.3307
mAAE: 0.1724
NDS: 0.3905
Eval time: 170.8s

Per-class results:
Object Class    AP      ATE     ASE     AOE     AVE     AAE
car     0.503   0.577   0.152   0.111   2.096   0.136
truck   0.223   0.857   0.224   0.220   1.389   0.179
bus     0.294   0.855   0.204   0.190   2.689   0.283
trailer 0.081   1.094   0.243   0.553   0.742   0.167
construction_vehicle    0.058   1.017   0.450   1.019   0.137   0.341
pedestrian      0.392   0.687   0.284   0.694   0.876   0.158
motorcycle      0.317   0.737   0.265   0.580   2.033   0.104
bicycle 0.308   0.704   0.299   0.892   0.683   0.010
traffic_cone    0.555   0.486   0.309   nan     nan     nan
barrier 0.466   0.581   0.269   0.169   nan     nan
```

12.6 Testing and make a submission

An example to test PointPillars on nuScenes with 8 GPUs and generate a submission to the leaderboard is as follows.

```
./tools/dist_test.sh configs/pointpillars/hv_pointpillars_fpn_sbn-all_4x8_2x_nus-3d.py␣
↪work_dirs/pp-nus/latest.pth 8 --out work_dirs/pp-nus/results_eval.pkl --format-only --
↪eval-options 'jsonfile_prefix=work_dirs/pp-nus/results_eval'
```

Note that the testing info should be changed to that for testing set instead of validation set here.

After generating the `work_dirs/pp-nus/results_eval.json`, you can compress it and submit it to nuScenes benchmark. Please refer to the nuScenes official website for more information.

We can also visualize the prediction results with our developed visualization tools. Please refer to the visualization doc for more details.

12.7 Notes

12.7.1 Transformation between `NuScenesBox` and our `CameraInstanceBoxes`.

In general, the main difference of `NuScenesBox` and our `CameraInstanceBoxes` is mainly reflected in the yaw definition. `NuScenesBox` defines the rotation with a quaternion or three Euler angles while ours only defines one yaw angle due to the practical scenario. It requires us to add some additional rotations manually in the pre-processing and post-processing, such as here.

In addition, please note that the definition of corners and locations are detached in the `NuScenesBox`. For example, in monocular 3D detection, the definition of the box location is in its camera coordinate (see its official illustration for car setup), which is consistent with ours. In contrast, its corners are defined with the convention "x points forward, y to the left, z up". It results in different philosophy of dimension and rotation definitions from our `CameraInstanceBoxes`. An example to remove similar hacks is PR #744. The same problem also exists in the LiDAR system. To deal with them, we typically add some transformation in the pre-processing and post-processing to guarantee the box will be in our coordinate system during the entire training and inference procedure.

CHAPTER

THIRTEEN

LYFT DATASET FOR 3D OBJECT DETECTION

This page provides specific tutorials about the usage of MMDetection3D for Lyft dataset.

13.1 Before Preparation

You can download Lyft 3D detection data HERE and unzip all zip files.

Like the general way to prepare a dataset, it is recommended to symlink the dataset root to `$MMDETECTION3D/data`.

The folder structure should be organized as follows before our processing.

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
    ├── lyft
        ├── v1.01-train
            ├── v1.01-train (train_data)
            ├── lidar (train_lidar)
            ├── images (train_images)
            ├── maps (train_maps)
        ├── v1.01-test
            ├── v1.01-test (test_data)
            ├── lidar (test_lidar)
            ├── images (test_images)
            ├── maps (test_maps)
        ├── train.txt
        ├── val.txt
        ├── test.txt
        ├── sample_submission.csv
```

Here `v1.01-train` and `v1.01-test` contain the metafiles which are similar to those of nuScenes. `.txt` files contain the data split information. Lyft does not have an official split for training and validation set, so we provide a split considering the number of objects from different categories in different scenes. `sample_submission.csv` is the base file for submission on the Kaggle evaluation server. Note that we follow the original folder names for clear organization. Please rename the raw folders as shown above.

13.2 Dataset Preparation

The way to organize Lyft dataset is similar to nuScenes. We also generate the .pkl and .json files which share almost the same structure. Next, we will mainly focus on the difference between these two datasets. For a more detailed explanation of the info structure, please refer to nuScenes tutorial.

To prepare info files for Lyft, run the following commands:

```
python tools/create_data.py lyft --root-path ./data/lyft --out-dir ./data/lyft --extra-
↪tag lyft --version v1.01
python tools/data_converter/lyft_data_fixer.py --version v1.01 --root-folder ./data/lyft
```

Note that the second command serves the purpose of fixing a corrupted lidar data file. Please refer to the discussion here for more details.

The folder structure after processing should be as below.

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
    ├── lyft
        ├── v1.01-train
            ├── v1.01-train (train_data)
            ├── lidar (train_lidar)
            ├── images (train_images)
            ├── maps (train_maps)
        ├── v1.01-test
            ├── v1.01-test (test_data)
            ├── lidar (test_lidar)
            ├── images (test_images)
            ├── maps (test_maps)
        ├── train.txt
        ├── val.txt
        ├── test.txt
        ├── sample_submission.csv
        ├── lyft_infos_train.pkl
        ├── lyft_infos_val.pkl
        ├── lyft_infos_test.pkl
        ├── lyft_infos_train_mono3d.coco.json
        ├── lyft_infos_val_mono3d.coco.json
        ├── lyft_infos_test_mono3d.coco.json
```

Here, .pkl files are generally used for methods involving point clouds, and coco-style .json files are more suitable for image-based methods, such as image-based 2D and 3D detection. Different from nuScenes, we only support using the json files for 2D detection experiments. Image-based 3D detection may be further supported in the future.

Next, we will elaborate on the difference compared to nuScenes in terms of the details recorded in these info files.

- without `lyft_database/xxxxx.bin`: This folder and `.bin` files are not extracted on the Lyft dataset due to the negligible effect of ground-truth sampling in the experiments.
- `lyft_infos_train.pkl`: training dataset infos, each frame info has two keys: `metadata` and `infos`. `metadata` contains the basic information for the dataset itself, such as `{'version': 'v1.01-train'}`, while `infos` contains the detailed information the same as nuScenes except for the following details:

  - info['sweeps']: Sweeps information.
    * info['sweeps'][i]['type']: The sweep data type, e.g., `'lidar'`. Lyft has different LiDAR settings for some samples, but we always take only the points collected by the top LiDAR for the consistency of data distribution.
  - info['gt_names']: There are 9 categories on the Lyft dataset, and the imbalance of annotations for different categories is even more significant than nuScenes.
  - without info['gt_velocity']: There is no velocity measurement on Lyft.
  - info['num_lidar_pts']: Set to -1 by default.
  - info['num_radar_pts']: Set to 0 by default.
  - without info['valid_flag']: This flag does recorded due to invalid `num_lidar_pts` and `num_radar_pts`.
- `nuscenes_infos_train_mono3d.coco.json`: training dataset coco-style info. This file only contains 2D information, without the information required by 3D detection, such as camera intrinsics.
  - info['images']: A list containing all the image info.
    * only containing `'file_name'`, `'id'`, `'width'`, `'height'`.
  - info['annotations']: A list containing all the annotation info.
    * only containing `'file_name'`, `'image_id'`, `'area'`, `'category_name'`, `'category_id'`, `'bbox'`, `'is_crowd'`, `'segmentation'`, `'id'`, where `'is_crowd'`, `'segmentation'` are set to `0` and `[]` by default. There is no attribute annotation on Lyft.

Here we only explain the data recorded in the training info files. The same applies to the testing set.

The core function to get `lyft_infos_xxx.pkl` is _fill_trainval_infos. Please refer to lyft_converter.py for more details.

13.3 Training pipeline

13.3.1 LiDAR-Based Methods

A typical training pipeline of LiDAR-based 3D detection (including multi-modality methods) on Lyft is almost the same as nuScenes as below.

```
train_pipeline = [
    dict(
        type='LoadPointsFromFile',
        coord_type='LIDAR',
        load_dim=5,
        use_dim=5,
        file_client_args=file_client_args),
    dict(
        type='LoadPointsFromMultiSweeps',
        sweeps_num=10,
        file_client_args=file_client_args),
    dict(type='LoadAnnotations3D', with_bbox_3d=True, with_label_3d=True),
    dict(
        type='GlobalRotScaleTrans',
        rot_range=[-0.3925, 0.3925],
        scale_ratio_range=[0.95, 1.05],
```

(continues on next page)

(continued from previous page)

```
        translation_std=[0, 0, 0]),
    dict(type='RandomFlip3D', flip_ratio_bev_horizontal=0.5),
    dict(type='PointsRangeFilter', point_cloud_range=point_cloud_range),
    dict(type='ObjectRangeFilter', point_cloud_range=point_cloud_range),
    dict(type='PointShuffle'),
    dict(type='DefaultFormatBundle3D', class_names=class_names),
    dict(type='Collect3D', keys=['points', 'gt_bboxes_3d', 'gt_labels_3d'])
]
```

Similar to nuScenes, models on Lyft also need the `'LoadPointsFromMultiSweeps'` pipeline to load point clouds from consecutive frames. In addition, considering the intensity of LiDAR points collected by Lyft is invalid, we also set the `use_dim` in `'LoadPointsFromMultiSweeps'` to `[0, 1, 2, 4]` by default, where the first 3 dimensions refer to point coordinates, and the last refers to timestamp differences.

13.4 Evaluation

An example to evaluate PointPillars with 8 GPUs with Lyft metrics is as follows.

```
bash ./tools/dist_test.sh configs/pointpillars/hv_pointpillars_fpn_sbn-all_2x8_2x_lyft-
↪3d.py checkpoints/hv_pointpillars_fpn_sbn-all_2x8_2x_lyft-3d_20210517_202818-fc6904c3.
↪pth 8 --eval bbox
```

13.5 Metrics

Lyft proposes a more strict metric for evaluating the predicted 3D bounding boxes. The basic criteria to judge whether a predicted box is positive or not is the same as KITTI, i.e. the 3D Intersection over Union (IoU). However, it adopts a way similar to COCO to compute the mean average precision (mAP) – compute the average precision under different thresholds of 3D IoU from 0.5-0.95. Actually, overlap more than 0.7 3D IoU is a quite strict criterion for 3D detection methods, so the overall performance seems a little low. The imbalance of annotations for different categories is another important reason for the finally lower results compared to other datasets. Please refer to its official website for more details about the definition of this metric.

We employ this official method for evaluation on Lyft. An example of printed evaluation results is as follows:

```
+mAPs@0.5:0.95------+--------------+
| class             | mAP@0.5:0.95 |
+-------------------+--------------+
| animal            | 0.0          |
| bicycle           | 0.099        |
| bus               | 0.177        |
| car               | 0.422        |
| emergency_vehicle | 0.0          |
| motorcycle        | 0.049        |
| other_vehicle     | 0.359        |
| pedestrian        | 0.066        |
| truck             | 0.176        |
| Overall           | 0.15         |
+-------------------+--------------+
```

13.6 Testing and make a submission

An example to test PointPillars on Lyft with 8 GPUs and generate a submission to the leaderboard is as follows.

```
./tools/dist_test.sh configs/pointpillars/hv_pointpillars_fpn_sbn-all_2x8_2x_lyft-3d.py␣
↪work_dirs/pp-lyft/latest.pth 8 --out work_dirs/pp-lyft/results_challenge.pkl --format-
↪only --eval-options 'jsonfile_prefix=work_dirs/pp-lyft/results_challenge' 'csv_
↪savepath=results/pp-lyft/results_challenge.csv'
```

After generating the `work_dirs/pp-lyft/results_challenge.csv`, you can submit it to the Kaggle evaluation server. Please refer to the official website for more information.

We can also visualize the prediction results with our developed visualization tools. Please refer to the visualization doc for more details.

CHAPTER

FOURTEEN

WAYMO DATASET

This page provides specific tutorials about the usage of MMDetection3D for Waymo dataset.

14.1 Prepare dataset

Before preparing Waymo dataset, if you only installed requirements in `requirements/build.txt` and `requirements/runtime.txt` before, please install the official package for this dataset at first by running

```
# tf 2.1.0.
pip install waymo-open-dataset-tf-2-1-0==1.2.0
# tf 2.0.0
# pip install waymo-open-dataset-tf-2-0-0==1.2.0
# tf 1.15.0
# pip install waymo-open-dataset-tf-1-15-0==1.2.0
```

or

```
pip install -r requirements/optional.txt
```

Like the general way to prepare dataset, it is recommended to symlink the dataset root to `$MMDETECTION3D/data`. Due to the original Waymo data format is based on `tfrecord`, we need to preprocess the raw data for convenient usage in the training and evaluation procedure. Our approach is to convert them into KITTI format.

The folder structure should be organized as follows before our processing.

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
│   ├── waymo
│   │   ├── waymo_format
│   │   │   ├── training
│   │   │   ├── validation
│   │   │   ├── testing
│   │   │   ├── gt.bin
│   │   ├── kitti_format
│   │   │   ├── ImageSets
```

You can download Waymo open dataset V1.2 HERE and its data split HERE. Then put `tfrecord` files into corresponding folders in `data/waymo/waymo_format/` and put the data split txt files into `data/waymo/kitti_format/ImageSets`. Download ground truth bin files for validation set HERE and put it into `data/waymo/waymo_format/`. A tip is that you can use `gsutil` to download the large-scale dataset with commands. You can take this tool as an example for more details. Subsequently, prepare Waymo data by running

```
python tools/create_data.py waymo --root-path ./data/waymo/ --out-dir ./data/waymo/ --
↪workers 128 --extra-tag waymo
```

Note that if your local disk does not have enough space for saving converted data, you can change the `--out-dir` to anywhere else. Just remember to create folders and prepare data there in advance and link them back to `data/waymo/kitti_format` after the data conversion.

After the data conversion, the folder structure and info files should be organized as below.

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
    ├── waymo
        ├── waymo_format
            ├── training
            ├── validation
            ├── testing
            ├── gt.bin
        ├── kitti_format
            ├── ImageSets
            ├── training
                ├── calib
                ├── image_0
                ├── image_1
                ├── image_2
                ├── image_3
                ├── image_4
                ├── label_0
                ├── label_1
                ├── label_2
                ├── label_3
                ├── label_4
                ├── label_all
                ├── pose
                ├── velodyne
            ├── testing
                ├── (the same as training)
            ├── waymo_gt_database
            ├── waymo_infos_trainval.pkl
            ├── waymo_infos_train.pkl
            ├── waymo_infos_val.pkl
            ├── waymo_infos_test.pkl
            ├── waymo_dbinfos_train.pkl
```

Here because there are several cameras, we store the corresponding image and labels that can be projected to that camera respectively and save pose for further usage of consecutive frames point clouds. We use a coding way `{a}{bbb}{ccc}`

to name the data for each frame, where `a` is the prefix for different split (`0` for training, `1` for validation and `2` for testing), `bbb` for segment index and `ccc` for frame index. You can easily locate the required frame according to this naming rule. We gather the data for training and validation together as KITTI and store the indices for different set in the `ImageSet` files.

14.2 Training

Considering there are many similar frames in the original dataset, we can basically use a subset to train our model primarily. In our preliminary baselines, we load one frame every five frames, and thanks to our hyper parameters settings and data augmentation, we obtain a better result compared with the performance given in the original dataset paper. For more details about the configuration and performance, please refer to README.md in the `configs/pointpillars/`. A more complete benchmark based on other settings and methods is coming soon.

14.3 Evaluation

For evaluation on Waymo, please follow the instruction to build the binary file `compute_detection_metrics_main` for metrics computation and put it into `mmdet3d/core/evaluation/waymo_utils/`. Basically, you can follow the commands below to install `bazel` and build the file.

```
# download the code and enter the base directory
git clone https://github.com/waymo-research/waymo-open-dataset.git waymo-od
cd waymo-od
git checkout remotes/origin/master

# use the Bazel build system
sudo apt-get install --assume-yes pkg-config zip g++ zlib1g-dev unzip python3 python3-pip
BAZEL_VERSION=3.1.0
wget https://github.com/bazelbuild/bazel/releases/download/${BAZEL_VERSION}/bazel-$
↪{BAZEL_VERSION}-installer-linux-x86_64.sh
sudo bash bazel-${BAZEL_VERSION}-installer-linux-x86_64.sh
sudo apt install build-essential

# configure .bazelrc
./configure.sh
# delete previous bazel outputs and reset internal caches
bazel clean

bazel build waymo_open_dataset/metrics/tools/compute_detection_metrics_main
cp bazel-bin/waymo_open_dataset/metrics/tools/compute_detection_metrics_main ../
↪mmdetection3d/mmdet3d/core/evaluation/waymo_utils/
```

Then you can evaluate your models on Waymo. An example to evaluate PointPillars on Waymo with 8 GPUs with Waymo metrics is as follows.

```
./tools/slurm_test.sh ${PARTITION} ${JOB_NAME} configs/pointpillars/hv_pointpillars_
↪secfpn_sbn-2x16_2x_waymo-3d-car.py \
    checkpoints/hv_pointpillars_secfpn_sbn-2x16_2x_waymo-3d-car_latest.pth --out results/
↪waymo-car/results_eval.pkl \
    --eval waymo --eval-options 'pklfile_prefix=results/waymo-car/kitti_results' \
    'submission_prefix=results/waymo-car/kitti_results'
```

`pklfile_prefix` should be given in the `--eval-options` if the bin file is needed to be generated. For metrics, `waymo` is the recommended official evaluation prototype. Currently, evaluating with choice `kitti` is adapted from KITTI and the results for each difficulty are not exactly the same as the definition of KITTI. Instead, most of objects are marked with difficulty 0 currently, which will be fixed in the future. The reasons of its instability include the large computation for evaluation, the lack of occlusion and truncation in the converted data, different definitions of difficulty and different methods of computing Average Precision.

Notice:

1. Sometimes when using `bazel` to build `compute_detection_metrics_main`, an error `'round' is not a member of 'std'` may appear. We just need to remove the `std::` before `round` in that file.
2. Considering it takes a little long time to evaluate once, we recommend to evaluate only once at the end of model training.
3. To use TensorFlow with CUDA 9, it is recommended to compile it from source. Apart from official tutorials, you can refer to this link for possibly suitable precompiled packages and useful information for compiling it from source.

14.4 Testing and make a submission

An example to test PointPillars on Waymo with 8 GPUs, generate the bin files and make a submission to the leaderboard.

```
./tools/slurm_test.sh ${PARTITION} ${JOB_NAME} configs/pointpillars/hv_pointpillars_
↪secfpn_sbn-2x16_2x_waymo-3d-car.py \
    checkpoints/hv_pointpillars_secfpn_sbn-2x16_2x_waymo-3d-car_latest.pth --out results/
↪waymo-car/results_eval.pkl \
    --format-only --eval-options 'pklfile_prefix=results/waymo-car/kitti_results' \
    'submission_prefix=results/waymo-car/kitti_results'
```

After generating the bin file, you can simply build the binary file `create_submission` and use them to create a submission file by following the instruction. Basically, here are some example commands.

```
cd ../waymo-od/
bazel build waymo_open_dataset/metrics/tools/create_submission
cp bazel-bin/waymo_open_dataset/metrics/tools/create_submission ../mmdetection3d/mmdet3d/
↪core/evaluation/waymo_utils/
vim waymo_open_dataset/metrics/tools/submission.txtpb  # set the metadata information
cp waymo_open_dataset/metrics/tools/submission.txtpb ../mmdetection3d/mmdet3d/core/
↪evaluation/waymo_utils/

cd ../mmdetection3d
# suppose the result bin is in `results/waymo-car/submission`
mmdet3d/core/evaluation/waymo_utils/create_submission  --input_filenames='results/waymo-
↪car/kitti_results_test.bin' --output_filename='results/waymo-car/submission/model' --
↪submission_filename='mmdet3d/core/evaluation/waymo_utils/submission.txtpb'

tar cvf results/waymo-car/submission/my_model.tar results/waymo-car/submission/my_model/
gzip results/waymo-car/submission/my_model.tar
```

For evaluation on the validation set with the eval server, you can also use the same way to generate a submission. Make sure you change the fields in `submission.txtpb` before running the command above.

CHAPTER

FIFTEEN

SUN RGB-D FOR 3D OBJECT DETECTION

15.1 Dataset preparation

For the overall process, please refer to the README page for SUN RGB-D.

15.1.1 Download SUN RGB-D data and toolbox

Download SUNRGBD data HERE. Then, move `SUNRGBD.zip`, `SUNRGBDMeta2DBB_v2.mat`, `SUNRGBDMeta3DBB_v2.mat` and `SUNRGBDtoolbox.zip` to the `OFFICIAL_SUNRGBD` folder, unzip the zip files.

The directory structure before data preparation should be as below:

```
sunrgbd
├── README.md
├── matlab
│   ├── extract_rgbd_data_v1.m
│   ├── extract_rgbd_data_v2.m
│   ├── extract_split.m
├── OFFICIAL_SUNRGBD
│   ├── SUNRGBD
│   ├── SUNRGBDMeta2DBB_v2.mat
│   ├── SUNRGBDMeta3DBB_v2.mat
│   ├── SUNRGBDtoolbox
```

15.1.2 Extract data and annotations for 3D detection from raw data

Extract SUN RGB-D annotation data from raw annotation data by running (this requires MATLAB installed on your machine):

```
matlab -nosplash -nodesktop -r 'extract_split;quit;'
matlab -nosplash -nodesktop -r 'extract_rgbd_data_v2;quit;'
matlab -nosplash -nodesktop -r 'extract_rgbd_data_v1;quit;'
```

The main steps include:

- Extract train and val split.
- Extract data for 3D detection from raw data.
- Extract and format detection annotation from raw data.

The main component of `extract_rgbd_data_v2.m` which extracts point cloud data from depth map is as follows:

```
data = SUNRGBDMeta(imageId);
data.depthpath(1:16) = '';
data.depthpath = strcat('../OFFICIAL_SUNRGBD', data.depthpath);
data.rgbpath(1:16) = '';
data.rgbpath = strcat('../OFFICIAL_SUNRGBD', data.rgbpath);

% extract point cloud from depth map
[rgb,points3d,depthInpaint,imsize]=read3dPoints(data);
rgb(isnan(points3d(:,1)),:) = [];
points3d(isnan(points3d(:,1)),:) = [];
points3d_rgb = [points3d, rgb];

% MAT files are 3x smaller than TXT files. In Python we can use
% scipy.io.loadmat('xxx.mat')['points3d_rgb'] to load the data.
mat_filename = strcat(num2str(imageId,'%06d'), '.mat');
txt_filename = strcat(num2str(imageId,'%06d'), '.txt');
% save point cloud data
parsave(strcat(depth_folder, mat_filename), points3d_rgb);
```

The main component of `extract_rgbd_data_v1.m` which extracts annotation is as follows:

```
% Write 2D and 3D box label
data2d = data;
fid = fopen(strcat(det_label_folder, txt_filename), 'w');
for j = 1:length(data.groundtruth3DBB)
    centroid = data.groundtruth3DBB(j).centroid;  % 3D bbox center
    classname = data.groundtruth3DBB(j).classname;  % class name
    orientation = data.groundtruth3DBB(j).orientation;  % 3D bbox orientation
    coeffs = abs(data.groundtruth3DBB(j).coeffs);  % 3D bbox size
    box2d = data2d.groundtruth2DBB(j).gtBb2D;  % 2D bbox
    fprintf(fid, '%s %d %d %d %d %f %f %f %f %f %f %f %f\n', classname, box2d(1),␣
↪box2d(2), box2d(3), box2d(4), centroid(1), centroid(2), centroid(3), coeffs(1),␣
↪coeffs(2), coeffs(3), orientation(1), orientation(2));
end
fclose(fid);
```

The above two scripts call functions such as `read3dPoints` from the toolbox provided by SUN RGB-D.

The directory structure after extraction should be as follows.

```
sunrgbd
├── README.md
├── matlab
    ├── extract_rgbd_data_v1.m
    ├── extract_rgbd_data_v2.m
    ├── extract_split.m
├── OFFICIAL_SUNRGBD
    ├── SUNRGBD
    ├── SUNRGBDMeta2DBB_v2.mat
    ├── SUNRGBDMeta3DBB_v2.mat
    ├── SUNRGBDtoolbox
├── sunrgbd_trainval
    ├── calib
    ├── depth
```

(continues on next page)

(continued from previous page)

```
    ├── image
    ├── label
    ├── label_v1
    ├── seg_label
    ├── train_data_idx.txt
    ├── val_data_idx.txt
```

Under each following folder there are overall 5285 train files and 5050 val files:

- `calib`: Camera calibration information in `.txt`
- `depth`: Point cloud saved in `.mat` (xyz+rgb)
- `image`: Image data in `.jpg`
- `label`: Detection annotation data in `.txt` (version 2)
- `label_v1`: Detection annotation data in `.txt` (version 1)
- `seg_label`: Segmentation annotation data in `.txt`

Currently, we use v1 data for training and testing, so the version 2 labels are unused.

15.1.3 Create dataset

Please run the command below to create the dataset.

```
python tools/create_data.py sunrgbd --root-path ./data/sunrgbd \
--out-dir ./data/sunrgbd --extra-tag sunrgbd
```

or (if in a slurm environment)

```
bash tools/create_data.sh <job_name> sunrgbd
```

The above point cloud data are further saved in `.bin` format. Meanwhile `.pkl` info files are also generated for saving annotation and metadata. The core function `process_single_scene` of getting data infos is as follows.

```
def process_single_scene(sample_idx):
    print(f'{self.split} sample_idx: {sample_idx}')
    # convert depth to points
    pc_upright_depth = self.get_depth(sample_idx)
    pc_upright_depth_subsampled = random_sampling(
        pc_upright_depth, self.num_points)

    info = dict()
    pc_info = {'num_features': 6, 'lidar_idx': sample_idx}
    info['point_cloud'] = pc_info

    # save point cloud data in `.bin` format
    mmcv.mkdir_or_exist(osp.join(self.root_dir, 'points'))
    pc_upright_depth_subsampled.tofile(
        osp.join(self.root_dir, 'points', f'{sample_idx:06d}.bin'))

    # save point cloud file path
    info['pts_path'] = osp.join('points', f'{sample_idx:06d}.bin')
```

(continues on next page)

(continued from previous page)

```
    # save image file path and metainfo
    img_path = osp.join('image', f'{sample_idx:06d}.jpg')
    image_info = {
        'image_idx': sample_idx,
        'image_shape': self.get_image_shape(sample_idx),
        'image_path': img_path
    }
    info['image'] = image_info

    # save calibration information
    K, Rt = self.get_calibration(sample_idx)
    calib_info = {'K': K, 'Rt': Rt}
    info['calib'] = calib_info

    # save all annotation
    if has_label:
        obj_list = self.get_label_objects(sample_idx)
        annotations = {}
        annotations['gt_num'] = len([
            obj.classname for obj in obj_list
            if obj.classname in self.cat2label.keys()
        ])
        if annotations['gt_num'] != 0:
            # class name
            annotations['name'] = np.array([
                obj.classname for obj in obj_list
                if obj.classname in self.cat2label.keys()
            ])
            # 2D image bounding boxes
            annotations['bbox'] = np.concatenate([
                obj.box2d.reshape(1, 4) for obj in obj_list
                if obj.classname in self.cat2label.keys()
            ], axis=0)
            # 3D bounding box center location (in depth coordinate system)
            annotations['location'] = np.concatenate([
                obj.centroid.reshape(1, 3) for obj in obj_list
                if obj.classname in self.cat2label.keys()
            ], axis=0)
            # 3D bounding box dimension/size (in depth coordinate system)
            annotations['dimensions'] = 2 * np.array([
                [obj.l, obj.h, obj.w] for obj in obj_list
                if obj.classname in self.cat2label.keys()
            ])
            # 3D bounding box rotation angle/yaw angle (in depth coordinate system)
            annotations['rotation_y'] = np.array([
                obj.heading_angle for obj in obj_list
                if obj.classname in self.cat2label.keys()
            ])
            annotations['index'] = np.arange(
                len(obj_list), dtype=np.int32)
            # class label (number)
```

(continues on next page)

(continued from previous page)

```
            annotations['class'] = np.array([
                self.cat2label[obj.classname] for obj in obj_list
                if obj.classname in self.cat2label.keys()
            ])
            # 3D bounding box (in depth coordinate system)
            annotations['gt_boxes_upright_depth'] = np.stack(
                [
                    obj.box3d for obj in obj_list
                    if obj.classname in self.cat2label.keys()
                ], axis=0)  # (K,8)
        info['annos'] = annotations
    return info
```

The directory structure after processing should be as follows.

```
sunrgbd
├── README.md
├── matlab
│   ├── ...
├── OFFICIAL_SUNRGBD
│   ├── ...
├── sunrgbd_trainval
│   ├── ...
├── points
├── sunrgbd_infos_train.pkl
├── sunrgbd_infos_val.pkl
```

- `points/0xxxxx.bin`: The point cloud data after downsample.
- `sunrgbd_infos_train.pkl`: The train data infos, the detailed info of each scene is as follows:
  - info['point_cloud']: ·{'num_features': 6, 'lidar_idx': sample_idx}, `where` sample_idx` is the index of the scene.
  - info['pts_path']: The path of `points/0xxxxx.bin`.
  - info['image']: The image path and metainfo:
    - image['image_idx']: The index of the image.
    - image['image_shape']: The shape of the image tensor.
    - image['image_path']: The path of the image.
  - info['annos']: The annotations of each scene.
    - annotations['gt_num']: The number of ground truths.
    - annotations['name']: The semantic name of all ground truths, e.g. `chair`.
    - annotations['location']: The gravity center of the 3D bounding boxes in depth coordinate system. Shape: [K, 3], K is the number of ground truths.
    - annotations['dimensions']: The dimensions of the 3D bounding boxes in depth coordinate system, i.e. `(x_size, y_size, z_size)`, shape: [K, 3].
    - annotations['rotation_y']: The yaw angle of the 3D bounding boxes in depth coordinate system. Shape: [K,].

  - annotations['gt_boxes_upright_depth']: The 3D bounding boxes in depth coordinate system, each bounding box is `(x, y, z, x_size, y_size, z_size, yaw)`, shape: [K, 7].
  - annotations['bbox']: The 2D bounding boxes, each bounding box is `(x, y, x_size, y_size)`, shape: [K, 4].
  - annotations['index']: The index of all ground truths, range [0, K).
  - annotations['class']: The train class id of the bounding boxes, value range: [0, 10), shape: [K,].

- `sunrgbd_infos_val.pkl`: The val data infos, which shares the same format as `sunrgbd_infos_train.pkl`.

15.2 Train pipeline

A typical train pipeline of SUN RGB-D for point cloud only 3D detection is as follows.

```
train_pipeline = [
    dict(
        type='LoadPointsFromFile',
        coord_type='DEPTH',
        shift_height=True,
        load_dim=6,
        use_dim=[0, 1, 2]),
    dict(type='LoadAnnotations3D'),
    dict(
        type='RandomFlip3D',
        sync_2d=False,
        flip_ratio_bev_horizontal=0.5,
    ),
    dict(
        type='GlobalRotScaleTrans',
        rot_range=[-0.523599, 0.523599],
        scale_ratio_range=[0.85, 1.15],
        shift_height=True),
    dict(type='PointSample', num_points=20000),
    dict(type='DefaultFormatBundle3D', class_names=class_names),
    dict(type='Collect3D', keys=['points', 'gt_bboxes_3d', 'gt_labels_3d'])
]
```

Data augmentation for point clouds:

- `RandomFlip3D`: randomly flip the input point cloud horizontally or vertically.
- `GlobalRotScaleTrans`: rotate the input point cloud, usually in the range of [-30, 30] (degrees) for SUN RGB-D; then scale the input point cloud, usually in the range of [0.85, 1.15] for SUN RGB-D; finally translate the input point cloud, usually by 0 for SUN RGB-D (which means no translation).
- `PointSample`: downsample the input point cloud.

A typical train pipeline of SUN RGB-D for multi-modality (point cloud and image) 3D detection is as follows.

```
train_pipeline = [
    dict(
        type='LoadPointsFromFile',
        coord_type='DEPTH',
        shift_height=True,
```

(continues on next page)

(continued from previous page)

```
        load_dim=6,
        use_dim=[0, 1, 2]),
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations3D'),
    dict(type='LoadAnnotations', with_bbox=True),
    dict(type='Resize', img_scale=(1333, 600), keep_ratio=True),
    dict(type='RandomFlip', flip_ratio=0.0),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='Pad', size_divisor=32),
    dict(
        type='RandomFlip3D',
        sync_2d=False,
        flip_ratio_bev_horizontal=0.5,
    ),
    dict(
        type='GlobalRotScaleTrans',
        rot_range=[-0.523599, 0.523599],
        scale_ratio_range=[0.85, 1.15],
        shift_height=True),
    dict(type='PointSample', num_points=20000),
    dict(type='DefaultFormatBundle3D', class_names=class_names),
    dict(
        type='Collect3D',
        keys=[
            'img', 'gt_bboxes', 'gt_labels', 'points', 'gt_bboxes_3d',
            'gt_labels_3d'
        ])
]
```

Data augmentation/normalization for images:

- `Resize`: resize the input image, `keep_ratio=True` means the ratio of the image is kept unchanged.
- `Normalize`: normalize the RGB channels of the input image.
- `RandomFlip`: randomly flip the input image.
- `Pad`: pad the input image with zeros by default.

The image augmentation and normalization functions are implemented in MMDetection.

15.3 Metrics

Same as ScanNet, typically mean Average Precision (mAP) is used for evaluation on SUN RGB-D, e.g. `mAP@0.25` and `mAP@0.5`. In detail, a generic function to compute precision and recall for 3D object detection for multiple classes is called, please refer to indoor_eval.

Since SUN RGB-D consists of image data, detection on image data is also feasible. For instance, in ImVoteNet, we first train an image detector, and we also use mAP for evaluation, e.g. `mAP@0.5`. We use the `eval_map` function from MMDetection to calculate mAP.

CHAPTER

SIXTEEN

SCANNET FOR 3D OBJECT DETECTION

16.1 Dataset preparation

For the overall process, please refer to the README page for ScanNet.

16.1.1 Export ScanNet point cloud data

By exporting ScanNet data, we load the raw point cloud data and generate the relevant annotations including semantic labels, instance labels and ground truth bounding boxes.

```
python batch_load_scannet_data.py
```

The directory structure before data preparation should be as below

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
    ├── scannet
        ├── meta_data
        ├── scans
            ├── scenexxxx_xx
        ├── batch_load_scannet_data.py
        ├── load_scannet_data.py
        ├── scannet_utils.py
        ├── README.md
```

Under folder `scans` there are overall 1201 train and 312 validation folders in which raw point cloud data and relevant annotations are saved. For instance, under folder `scene0001_01` the files are as below:

- `scene0001_01_vh_clean_2.ply`: Mesh file storing coordinates and colors of each vertex. The mesh's vertices are taken as raw point cloud data.
- `scene0001_01.aggregation.json`: Aggregation file including object ID, segments ID and label.
- `scene0001_01_vh_clean_2.0.010000.segs.json`: Segmentation file including segments ID and vertex.
- `scene0001_01.txt`: Meta file including axis-aligned matrix, etc.
- `scene0001_01_vh_clean_2.labels.ply`: Annotation file containing the category of each vertex.

Export ScanNet data by running `python batch_load_scannet_data.py`. The main steps include:

- Export original files to point cloud, instance label, semantic label and bounding box file.
- Downsample raw point cloud and filter invalid classes.
- Save point cloud data and relevant annotation files.

And the core function `export` in `load_scannet_data.py` is as follows:

```
def export(mesh_file,
           agg_file,
           seg_file,
           meta_file,
           label_map_file,
           output_file=None,
           test_mode=False):

    # label map file: ./data/scannet/meta_data/scannetv2-labels.combined.tsv
    # the various label standards in the label map file, e.g. 'nyu40id'
    label_map = scannet_utils.read_label_mapping(
        label_map_file, label_from='raw_category', label_to='nyu40id')
    # load raw point cloud data, 6-dims feature: XYZRGB
    mesh_vertices = scannet_utils.read_mesh_vertices_rgb(mesh_file)

    # Load scene axis alignment matrix: a 4x4 transformation matrix
    # transform raw points in sensor coordinate system to a coordinate system
    # which is axis-aligned with the length/width of the room
    lines = open(meta_file).readlines()
    # test set data doesn't have align_matrix
    axis_align_matrix = np.eye(4)
    for line in lines:
        if 'axisAlignment' in line:
            axis_align_matrix = [
                float(x)
                for x in line.rstrip().strip('axisAlignment = ').split(' ')
            ]
            break
    axis_align_matrix = np.array(axis_align_matrix).reshape((4, 4))

    # perform global alignment of mesh vertices
    pts = np.ones((mesh_vertices.shape[0], 4))
    # raw point cloud in homogeneous coordinates, each row: [x, y, z, 1]
    pts[:, 0:3] = mesh_vertices[:, 0:3]
    # transform raw mesh vertices to aligned mesh vertices
    pts = np.dot(pts, axis_align_matrix.transpose())  # Nx4
    aligned_mesh_vertices = np.concatenate([pts[:, 0:3], mesh_vertices[:, 3:]],
                                           axis=1)

    # Load semantic and instance labels
    if not test_mode:
        # each object has one semantic label and consists of several segments
        object_id_to_segs, label_to_segs = read_aggregation(agg_file)
        # many points may belong to the same segment
        seg_to_verts, num_verts = read_segmentation(seg_file)
        label_ids = np.zeros(shape=(num_verts), dtype=np.uint32)
        object_id_to_label_id = {}
```

(continues on next page)

(continued from previous page)

```
        for label, segs in label_to_segs.items():
            label_id = label_map[label]
            for seg in segs:
                verts = seg_to_verts[seg]
                # each point has one semantic label
                label_ids[verts] = label_id
        instance_ids = np.zeros(
            shape=(num_verts), dtype=np.uint32)  # 0: unannotated
        for object_id, segs in object_id_to_segs.items():
            for seg in segs:
                verts = seg_to_verts[seg]
                # object_id is 1-indexed, i.e. 1,2,3,.,,,,.NUM_INSTANCES
                # each point belongs to one object
                instance_ids[verts] = object_id
                if object_id not in object_id_to_label_id:
                    object_id_to_label_id[object_id] = label_ids[verts][0]
        # bbox format is [x, y, z, x_size, y_size, z_size, label_id]
        # [x, y, z] is gravity center of bbox, [x_size, y_size, z_size] is axis-aligned
        # [label_id] is semantic label id in 'nyu40id' standard
        # Note: since 3D bbox is axis-aligned, the yaw is 0.
        unaligned_bboxes = extract_bbox(mesh_vertices, object_id_to_segs,
                                        object_id_to_label_id, instance_ids)
        aligned_bboxes = extract_bbox(aligned_mesh_vertices, object_id_to_segs,
                                      object_id_to_label_id, instance_ids)
    ...

    return mesh_vertices, label_ids, instance_ids, unaligned_bboxes, \
        aligned_bboxes, object_id_to_label_id, axis_align_matrix
```

After exporting each scan, the raw point cloud could be downsampled, e.g. to 50000, if the number of points is too large (the raw point cloud won't be downsampled if it's also used in 3D semantic segmentation task). In addition, invalid semantic labels outside of `nyu40id` standard or optional `DONOT CARE` classes should be filtered. Finally, the point cloud data, semantic labels, instance labels and ground truth bounding boxes should be saved in `.npy` files.

16.1.2 Export ScanNet RGB data (optional)

By exporting ScanNet RGB data, for each scene we load a set of RGB images with corresponding 4x4 pose matrices, and a single 4x4 camera intrinsic matrix. Note, that this step is optional and can be skipped if multi-view detection is not planned to use.

```
python extract_posed_images.py
```

Each of 1201 train, 312 validation and 100 test scenes contains a single `.sens` file. For instance, for scene `0001_01` we have `data/scannet/scans/scene0001_01/0001_01.sens`. For this scene all images and poses are extracted to `data/scannet/posed_images/scene0001_01`. Specifically, there will be 300 image files xxxxx.jpg, 300 camera pose files xxxxx.txt and a single `intrinsic.txt` file. Typically, single scene contains several thousand images. By default, we extract only 300 of them with resulting space occupation of <100 Gb. To extract more images, use `--max-images-per-scene` parameter.

16.1.3 Create dataset

```
python tools/create_data.py scannet --root-path ./data/scannet \
--out-dir ./data/scannet --extra-tag scannet
```

The above exported point cloud file, semantic label file and instance label file are further saved in `.bin` format. Meanwhile `.pkl` info files are also generated for train or validation. The core function `process_single_scene` of getting data infos is as follows.

```
def process_single_scene(sample_idx):

    # save point cloud, instance label and semantic label in .bin file respectively, get␣
↪info['pts_path'], info['pts_instance_mask_path'] and info['pts_semantic_mask_path']
    ...

    # get annotations
    if has_label:
        annotations = {}
        # box is of shape [k, 6 + class]
        aligned_box_label = self.get_aligned_box_label(sample_idx)
        unaligned_box_label = self.get_unaligned_box_label(sample_idx)
        annotations['gt_num'] = aligned_box_label.shape[0]
        if annotations['gt_num'] != 0:
            aligned_box = aligned_box_label[:, :-1]  # k, 6
            unaligned_box = unaligned_box_label[:, :-1]
            classes = aligned_box_label[:, -1]  # k
            annotations['name'] = np.array([
                self.label2cat[self.cat_ids2class[classes[i]]]
                for i in range(annotations['gt_num'])
            ])
            # default names are given to aligned bbox for compatibility
            # we also save unaligned bbox info with marked names
            annotations['location'] = aligned_box[:, :3]
            annotations['dimensions'] = aligned_box[:, 3:6]
            annotations['gt_boxes_upright_depth'] = aligned_box
            annotations['unaligned_location'] = unaligned_box[:, :3]
            annotations['unaligned_dimensions'] = unaligned_box[:, 3:6]
            annotations[
                'unaligned_gt_boxes_upright_depth'] = unaligned_box
            annotations['index'] = np.arange(
                annotations['gt_num'], dtype=np.int32)
            annotations['class'] = np.array([
                self.cat_ids2class[classes[i]]
                for i in range(annotations['gt_num'])
            ])
        axis_align_matrix = self.get_axis_align_matrix(sample_idx)
        annotations['axis_align_matrix'] = axis_align_matrix  # 4x4
        info['annos'] = annotations
    return info
```

The directory structure after process should be as below

```
scannet
```

(continues on next page)

(continued from previous page)

```
├── meta_data
├── batch_load_scannet_data.py
├── load_scannet_data.py
├── scannet_utils.py
├── README.md
├── scans
├── scans_test
├── scannet_instance_data
├── points
│   ├── xxxxx.bin
├── instance_mask
│   ├── xxxxx.bin
├── semantic_mask
│   ├── xxxxx.bin
├── seg_info
│   ├── train_label_weight.npy
│   ├── train_resampled_scene_idxs.npy
│   ├── val_label_weight.npy
│   ├── val_resampled_scene_idxs.npy
├── posed_images
│   ├── scenexxxx_xx
│   │   ├── xxxxxx.txt
│   │   ├── xxxxxx.jpg
│   │   ├── intrinsic.txt
├── scannet_infos_train.pkl
├── scannet_infos_val.pkl
├── scannet_infos_test.pkl
```

- `points/xxxxx.bin`: The `axis-unaligned` point cloud data after downsample. Since ScanNet 3D detection task takes axis-aligned point clouds as input, while ScanNet 3D semantic segmentation task takes unaligned points, we choose to store unaligned points and their axis-align transform matrix. Note: the points would be axis-aligned in pre-processing pipeline `GlobalAlignment` of 3D detection task.
- `instance_mask/xxxxx.bin`: The instance label for each point, value range: [0, NUM_INSTANCES], 0: unannotated.
- `semantic_mask/xxxxx.bin`: The semantic label for each point, value range: [1, 40], i.e. `nyu40id` standard. Note: the `nyu40id` ID will be mapped to train ID in train pipeline `PointSegClassMapping`.
- `posed_images/scenexxxx_xx`: The set of `.jpg` images with `.txt` 4x4 poses and the single `.txt` file with camera intrinsic matrix.
- `scannet_infos_train.pkl`: The train data infos, the detailed info of each scan is as follows:
  - info['point_cloud']: {'num_features': 6, 'lidar_idx': sample_idx}.
  - info['pts_path']: The path of `points/xxxxx.bin`.
  - info['pts_instance_mask_path']: The path of `instance_mask/xxxxx.bin`.
  - info['pts_semantic_mask_path']: The path of `semantic_mask/xxxxx.bin`.
  - info['annos']: The annotations of each scan.
    * annotations['gt_num']: The number of ground truths.
    * annotations['name'] The semantic name of all ground truths, e.g. `chair`.

  - annotations['location']: The gravity center of the axis-aligned 3D bounding boxes in depth coordinate system. Shape: [K, 3], K is the number of ground truths.
  - annotations['dimensions']: The dimensions of the axis-aligned 3D bounding boxes in depth coordinate system, i.e. (x_size, y_size, z_size), shape: [K, 3].
  - annotations['gt_boxes_upright_depth']: The axis-aligned 3D bounding boxes in depth coordinate system, each bounding box is (x, y, z, x_size, y_size, z_size), shape: [K, 6].
  - annotations['unaligned_location']: The gravity center of the axis-unaligned 3D bounding boxes in depth coordinate system.
  - annotations['unaligned_dimensions']: The dimensions of the axis-unaligned 3D bounding boxes in depth coordinate system.
  - annotations['unaligned_gt_boxes_upright_depth']: The axis-unaligned 3D bounding boxes in depth coordinate system.
  - annotations['index']: The index of all ground truths, i.e. [0, K).
  - annotations['class']: The train class ID of the bounding boxes, value range: [0, 18), shape: [K,].

- `scannet_infos_val.pkl`: The val data infos, which shares the same format as `scannet_infos_train.pkl`.
- `scannet_infos_test.pkl`: The test data infos, which almost shares the same format as `scannet_infos_train.pkl` except for the lack of annotation.

16.2 Training pipeline

A typical training pipeline of ScanNet for 3D detection is as follows.

```
train_pipeline = [
    dict(
        type='LoadPointsFromFile',
        coord_type='DEPTH',
        shift_height=True,
        load_dim=6,
        use_dim=[0, 1, 2]),
    dict(
        type='LoadAnnotations3D',
        with_bbox_3d=True,
        with_label_3d=True,
        with_mask_3d=True,
        with_seg_3d=True),
    dict(type='GlobalAlignment', rotation_axis=2),
    dict(
        type='PointSegClassMapping',
        valid_cat_ids=(3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 24, 28, 33, 34,
                       36, 39),
        max_cat_id=40),
    dict(type='PointSample', num_points=40000),
    dict(
        type='RandomFlip3D',
        sync_2d=False,
        flip_ratio_bev_horizontal=0.5,
        flip_ratio_bev_vertical=0.5),
```

(continues on next page)

(continued from previous page)

```
    dict(
        type='GlobalRotScaleTrans',
        rot_range=[-0.087266, 0.087266],
        scale_ratio_range=[1.0, 1.0],
        shift_height=True),
    dict(type='DefaultFormatBundle3D', class_names=class_names),
    dict(
        type='Collect3D',
        keys=[
            'points', 'gt_bboxes_3d', 'gt_labels_3d', 'pts_semantic_mask',
            'pts_instance_mask'
        ])
]
```

- `GlobalAlignment`: The previous point cloud would be axis-aligned using the axis-aligned matrix.
- `PointSegClassMapping`: Only the valid category IDs will be mapped to class label IDs like [0, 18) during training.
- Data augmentation:
  - `PointSample`: downsample the input point cloud.
  - `RandomFlip3D`: randomly flip the input point cloud horizontally or vertically.
  - `GlobalRotScaleTrans`: rotate the input point cloud, usually in the range of [-5, 5] (degrees) for ScanNet; then scale the input point cloud, usually by 1.0 for ScanNet (which means no scaling); finally translate the input point cloud, usually by 0 for ScanNet (which means no translation).

16.3 Metrics

Typically mean Average Precision (mAP) is used for evaluation on ScanNet, e.g. `mAP@0.25` and `mAP@0.5`. In detail, a generic function to compute precision and recall for 3D object detection for multiple classes is called, please refer to indoor_eval.

As introduced in section `Export ScanNet data`, all ground truth 3D bounding box are axis-aligned, i.e. the yaw is zero. So the yaw target of network predicted 3D bounding box is also zero and axis-aligned 3D Non-Maximum Suppression (NMS), which is regardless of rotation, is adopted during post-processing .

CHAPTER

SEVENTEEN

SCANNET FOR 3D SEMANTIC SEGMENTATION

17.1 Dataset preparation

The overall process is similar to ScanNet 3D detection task. Please refer to this section. Only a few differences and additional information about the 3D semantic segmentation data will be listed below.

17.1.1 Export ScanNet data

Since ScanNet provides online benchmark for 3D semantic segmentation evaluation on the test set, we need to also download the test scans and put it under `scannet` folder.

The directory structure before data preparation should be as below:

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
│   ├── scannet
│   │   ├── meta_data
│   │   ├── scans
│   │   │   ├── scenexxxx_xx
│   │   ├── scans_test
│   │   │   ├── scenexxxx_xx
│   │   ├── batch_load_scannet_data.py
│   │   ├── load_scannet_data.py
│   │   ├── scannet_utils.py
│   │   ├── README.md
```

Under folder `scans_test` there are 100 test folders in which only raw point cloud data and its meta file are saved. For instance, under folder `scene0707_00` the files are as below:

- `scene0707_00_vh_clean_2.ply`: Mesh file storing coordinates and colors of each vertex. The mesh's vertices are taken as raw point cloud data.
- `scene0707_00.txt`: Meta file including sensor parameters, etc. Note: different from data under `scans`, axis-aligned matrix is not provided for test scans.

Export ScanNet data by running `python batch_load_scannet_data.py`. Note: only point cloud data will be saved for test set scans because no annotations are provided.

17.1.2 Create dataset

Similar to the 3D detection task, we create dataset by running `python tools/create_data.py scannet --root-path ./data/scannet --out-dir ./data/scannet --extra-tag scannet`. The directory structure after processing should be as below:

```
scannet
├── scannet_utils.py
├── batch_load_scannet_data.py
├── load_scannet_data.py
├── scannet_utils.py
├── README.md
├── scans
├── scans_test
├── scannet_instance_data
├── points
│   ├── xxxxx.bin
├── instance_mask
│   ├── xxxxx.bin
├── semantic_mask
│   ├── xxxxx.bin
├── seg_info
│   ├── train_label_weight.npy
│   ├── train_resampled_scene_idxs.npy
│   ├── val_label_weight.npy
│   ├── val_resampled_scene_idxs.npy
├── scannet_infos_train.pkl
├── scannet_infos_val.pkl
├── scannet_infos_test.pkl
```

- `seg_info`: The generated infos to support semantic segmentation model training.
  - `train_label_weight.npy`: Weighting factor for each semantic class. Since the number of points in different classes varies greatly, it's a common practice to use label re-weighting to get a better performance.
  - `train_resampled_scene_idxs.npy`: Re-sampling index for each scene. Different rooms will be sampled multiple times according to their number of points to balance training data.

17.2 Training pipeline

A typical training pipeline of ScanNet for 3D semantic segmentation is as below:

```
train_pipeline = [
    dict(
        type='LoadPointsFromFile',
        coord_type='DEPTH',
        shift_height=False,
        use_color=True,
        load_dim=6,
        use_dim=[0, 1, 2, 3, 4, 5]),
    dict(
        type='LoadAnnotations3D',
        with_bbox_3d=False,
```

(continues on next page)

(continued from previous page)

```
        with_label_3d=False,
        with_mask_3d=False,
        with_seg_3d=True),
    dict(
        type='PointSegClassMapping',
        valid_cat_ids=(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 24, 28,
                       33, 34, 36, 39),
        max_cat_id=40),
    dict(
        type='IndoorPatchPointSample',
        num_points=num_points,
        block_size=1.5,
        ignore_index=len(class_names),
        use_normalized_coord=False,
        enlarge_size=0.2,
        min_unique_num=None),
    dict(type='NormalizePointsColor', color_mean=None),
    dict(type='DefaultFormatBundle3D', class_names=class_names),
    dict(type='Collect3D', keys=['points', 'pts_semantic_mask'])
]
```

- `PointSegClassMapping`: Only the valid category ids will be mapped to class label ids like [0, 20) during training. Other class ids will be converted to `ignore_index` which equals to `20`.
- `IndoorPatchPointSample`: Crop a patch containing a fixed number of points from input point cloud. `block_size` indicates the size of the cropped block, typically `1.5` for ScanNet.
- `NormalizePointsColor`: Normalize the RGB color values of input point cloud by dividing `255`.

17.3 Metrics

Typically mean Intersection over Union (mIoU) is used for evaluation on ScanNet. In detail, we first compute IoU for multiple classes and then average them to get mIoU, please refer to seg_eval.

17.4 Testing and Making a Submission

By default, our codebase evaluates semantic segmentation results on the validation set. If you would like to test the model performance on the online benchmark, add `--format-only` flag in the evaluation script and change `ann_file=data_root + 'scannet_infos_val.pkl'` to `ann_file=data_root + 'scannet_infos_test.pkl'` in the ScanNet dataset's config. Remember to specify the `txt_prefix` as the directory to save the testing results.

Taking PointNet++ (SSG) on ScanNet for example, the following command can be used to do inference on test set:

```
./tools/dist_test.sh configs/pointnet2/pointnet2_ssg_16x2_cosine_200e_scannet_seg-3d-
↪20class.py \
    work_dirs/pointnet2_ssg/latest.pth --format-only \
    --eval-options txt_prefix=work_dirs/pointnet2_ssg/test_submission
```

After generating the results, you can basically compress the folder and upload to the ScanNet evaluation server.

CHAPTER

EIGHTEEN

S3DIS FOR 3D SEMANTIC SEGMENTATION

18.1 Dataset preparation

For the overall process, please refer to the README page for S3DIS.

18.1.1 Export S3DIS data

By exporting S3DIS data, we load the raw point cloud data and generate the relevant annotations including semantic labels and instance labels.

The directory structure before exporting should be as below:

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
│   ├── s3dis
│   │   ├── meta_data
│   │   ├── Stanford3dDataset_v1.2_Aligned_Version
│   │   │   ├── Area_1
│   │   │   │   ├── conferenceRoom_1
│   │   │   │   ├── office_1
│   │   │   │   ├── ...
│   │   │   ├── Area_2
│   │   │   ├── Area_3
│   │   │   ├── Area_4
│   │   │   ├── Area_5
│   │   │   ├── Area_6
│   │   ├── indoor3d_util.py
│   │   ├── collect_indoor3d_data.py
│   │   ├── README.md
```

Under folder `Stanford3dDataset_v1.2_Aligned_Version`, the rooms are spilted into 6 areas. We use 5 areas for training and 1 for evaluation (typically `Area_5`). Under the directory of each area, there are folders in which raw point cloud data and relevant annotations are saved. For instance, under folder `Area_1/office_1` the files are as below:

- `office_1.txt`: A txt file storing coordinates and colors of each point in the raw point cloud data.
- `Annotations/`: This folder contains txt files for different object instances. Each txt file represents one instance, e.g.

- `chair_1.txt`: A txt file storing raw point cloud data of one chair in this room.

If we concat all the txt files under `Annotations/`, we will get the same point cloud as denoted by `office_1.txt`.

Export S3DIS data by running `python collect_indoor3d_data.py`. The main steps include:

- Export original txt files to point cloud, instance label and semantic label.
- Save point cloud data and relevant annotation files.

And the core function `export` in `indoor3d_util.py` is as follows:

```
def export(anno_path, out_filename):
    """Convert original dataset files to points, instance mask and semantic
    mask files. We aggregated all the points from each instance in the room.

    Args:
        anno_path (str): path to annotations. e.g. Area_1/office_2/Annotations/
        out_filename (str): path to save collected points and labels.
        file_format (str): txt or numpy, determines what file format to save.

    Note:
        the points are shifted before save, the most negative point is now
            at origin.
    """
    points_list = []
    ins_idx = 1  # instance ids should be indexed from 1, so 0 is unannotated

    # an example of `anno_path`: Area_1/office_1/Annotations
    # which contains all object instances in this room as txt files
    for f in glob.glob(osp.join(anno_path, '*.txt')):
        # get class name of this instance
        one_class = osp.basename(f).split('_')[0]
        if one_class not in class_names:  # some rooms have 'staris' class
            one_class = 'clutter'
        points = np.loadtxt(f)
        labels = np.ones((points.shape[0], 1)) * class2label[one_class]
        ins_labels = np.ones((points.shape[0], 1)) * ins_idx
        ins_idx += 1
        points_list.append(np.concatenate([points, labels, ins_labels], 1))

    data_label = np.concatenate(points_list, 0)  # [N, 8], (pts, rgb, sem, ins)
    # align point cloud to the origin
    xyz_min = np.amin(data_label, axis=0)[0:3]
    data_label[:, 0:3] -= xyz_min

    np.save(f'{out_filename}_point.npy', data_label[:, :6].astype(np.float32))
    np.save(f'{out_filename}_sem_label.npy', data_label[:, 6].astype(np.int))
    np.save(f'{out_filename}_ins_label.npy', data_label[:, 7].astype(np.int))
```

where we load and concatenate all the point cloud instances under `Annotations/` to form raw point cloud and generate semantic/instance labels. After exporting each room, the point cloud data, semantic labels and instance labels should be saved in `.npy` files.

18.1.2 Create dataset

```
python tools/create_data.py s3dis --root-path ./data/s3dis \
--out-dir ./data/s3dis --extra-tag s3dis
```

The above exported point cloud files, semantic label files and instance label files are further saved in `.bin` format. Meanwhile `.pkl` info files are also generated for each area.

The directory structure after process should be as below:

```
s3dis
├── meta_data
├── indoor3d_util.py
├── collect_indoor3d_data.py
├── README.md
├── Stanford3dDataset_v1.2_Aligned_Version
├── s3dis_data
├── points
│   ├── xxxxx.bin
├── instance_mask
│   ├── xxxxx.bin
├── semantic_mask
│   ├── xxxxx.bin
├── seg_info
│   ├── Area_1_label_weight.npy
│   ├── Area_1_resampled_scene_idxs.npy
│   ├── Area_2_label_weight.npy
│   ├── Area_2_resampled_scene_idxs.npy
│   ├── Area_3_label_weight.npy
│   ├── Area_3_resampled_scene_idxs.npy
│   ├── Area_4_label_weight.npy
│   ├── Area_4_resampled_scene_idxs.npy
│   ├── Area_5_label_weight.npy
│   ├── Area_5_resampled_scene_idxs.npy
│   ├── Area_6_label_weight.npy
│   ├── Area_6_resampled_scene_idxs.npy
├── s3dis_infos_Area_1.pkl
├── s3dis_infos_Area_2.pkl
├── s3dis_infos_Area_3.pkl
├── s3dis_infos_Area_4.pkl
├── s3dis_infos_Area_5.pkl
├── s3dis_infos_Area_6.pkl
```

- `points/xxxxx.bin`: The exported point cloud data.
- `instance_mask/xxxxx.bin`: The instance label for each point, value range: [0, ${NUM_INSTANCES}], 0: unannotated.
- `semantic_mask/xxxxx.bin`: The semantic label for each point, value range: [0, 12].
- `s3dis_infos_Area_1.pkl`: Area 1 data infos, the detailed info of each room is as follows:
  - info['point_cloud']: {'num_features': 6, 'lidar_idx': sample_idx}.
  - info['pts_path']: The path of `points/xxxxx.bin`.
  - info['pts_instance_mask_path']: The path of `instance_mask/xxxxx.bin`.

  - info['pts_semantic_mask_path']: The path of `semantic_mask/xxxxx.bin`.

- `seg_info`: The generated infos to support semantic segmentation model training.
  - `Area_1_label_weight.npy`: Weighting factor for each semantic class. Since the number of points in different classes varies greatly, it's a common practice to use label re-weighting to get a better performance.
  - `Area_1_resampled_scene_idxs.npy`: Re-sampling index for each scene. Different rooms will be sampled multiple times according to their number of points to balance training data.

18.2 Training pipeline

A typical training pipeline of S3DIS for 3D semantic segmentation is as below.

```
class_names = ('ceiling', 'floor', 'wall', 'beam', 'column', 'window', 'door',
               'table', 'chair', 'sofa', 'bookcase', 'board', 'clutter')
num_points = 4096
train_pipeline = [
    dict(
        type='LoadPointsFromFile',
        coord_type='DEPTH',
        shift_height=False,
        use_color=True,
        load_dim=6,
        use_dim=[0, 1, 2, 3, 4, 5]),
    dict(
        type='LoadAnnotations3D',
        with_bbox_3d=False,
        with_label_3d=False,
        with_mask_3d=False,
        with_seg_3d=True),
    dict(
        type='PointSegClassMapping',
        valid_cat_ids=tuple(range(len(class_names))),
        max_cat_id=13),
    dict(
        type='IndoorPatchPointSample',
        num_points=num_points,
        block_size=1.0,
        ignore_index=None,
        use_normalized_coord=True,
        enlarge_size=None,
        min_unique_num=num_points // 4,
        eps=0.0),
    dict(type='NormalizePointsColor', color_mean=None),
    dict(
        type='GlobalRotScaleTrans',
        rot_range=[-3.141592653589793, 3.141592653589793],  # [-pi, pi]
        scale_ratio_range=[0.8, 1.2],
        translation_std=[0, 0, 0]),
    dict(
        type='RandomJitterPoints',
        jitter_std=[0.01, 0.01, 0.01],
```

(continues on next page)

(continued from previous page)

```
        clip_range=[-0.05, 0.05]),
    dict(type='RandomDropPointsColor', drop_ratio=0.2),
    dict(type='DefaultFormatBundle3D', class_names=class_names),
    dict(type='Collect3D', keys=['points', 'pts_semantic_mask'])
]
```

- `PointSegClassMapping`: Only the valid category ids will be mapped to class label ids like [0, 13) during training. Other class ids will be converted to `ignore_index` which equals to `13`.
- `IndoorPatchPointSample`: Crop a patch containing a fixed number of points from input point cloud. `block_size` indicates the size of the cropped block, typically `1.0` for S3DIS.
- `NormalizePointsColor`: Normalize the RGB color values of input point cloud by dividing `255`.
- Data augmentation:
  - `GlobalRotScaleTrans`: randomly rotate and scale input point cloud.
  - `RandomJitterPoints`: randomly jitter point cloud by adding different noise vector to each point.
  - `RandomDropPointsColor`: set the colors of point cloud to all zeros by a probability `drop_ratio`.

18.3 Metrics

Typically mean intersection over union (mIoU) is used for evaluation on S3DIS. In detail, we first compute IoU for multiple classes and then average them to get mIoU, please refer to seg_eval.py.

As introduced in section `Export S3DIS data`, S3DIS trains on 5 areas and evaluates on the remaining 1 area. But there are also other area split schemes in different papers. To enable flexible combination of train-val splits, we use sub-dataset to represent one area, and concatenate them to form a larger training set. An example of training on area 1, 2, 3, 4, 6 and evaluating on area 5 is shown as below:

```
dataset_type = 'S3DISSegDataset'
data_root = './data/s3dis/'
class_names = ('ceiling', 'floor', 'wall', 'beam', 'column', 'window', 'door',
               'table', 'chair', 'sofa', 'bookcase', 'board', 'clutter')
train_area = [1, 2, 3, 4, 6]
test_area = 5
data = dict(
    train=dict(
        type=dataset_type,
        data_root=data_root,
        ann_files=[
            data_root + f's3dis_infos_Area_{i}.pkl' for i in train_area
        ],
        pipeline=train_pipeline,
        classes=class_names,
        test_mode=False,
        ignore_index=len(class_names),
        scene_idxs=[
            data_root + f'seg_info/Area_{i}_resampled_scene_idxs.npy'
            for i in train_area
        ]),
    val=dict(
```

(continues on next page)

(continued from previous page)

```
        type=dataset_type,
        data_root=data_root,
        ann_files=data_root + f's3dis_infos_Area_{test_area}.pkl',
        pipeline=test_pipeline,
        classes=class_names,
        test_mode=True,
        ignore_index=len(class_names),
        scene_idxs=data_root +
        f'seg_info/Area_{test_area}_resampled_scene_idxs.npy'))
```

where we specify the areas used for training/validation by setting `ann_files` and `scene_idxs` with lists that include corresponding paths. The train-val split can be simply modified via changing the `train_area` and `test_area` variables.

CHAPTER

NINETEEN

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. You may also pass `--options xxx.yyy=zzz` to see updated config.

19.1 Config File Structure

There are 4 basic component types under `config/_base_`, dataset, model, schedule, default_runtime. Many methods could be easily constructed with one of each like SECOND, PointPillars, PartA2, and VoteNet. The configs that are composed by components from `_base_` are called *primitive*.

For all configs under the same folder, it is recommended to have only **one** *primitive* config. All other configs should inherit from the *primitive* config. In this way, the maximum of inheritance level is 3.

For easy understanding, we recommend contributors to inherit from exiting methods. For example, if some modification is made based on PointPillars, user may first inherit the basic PointPillars structure by specifying `_base_ = ../pointpillars/hv_pointpillars_fpn_sbn-all_4x8_2x_nus-3d.py`, then modify the necessary fields in the config files.

If you are building an entirely new method that does not share the structure with any of the existing methods, you may create a folder `xxx_rcnn` under `configs`,

Please refer to mmcv for detailed documentation.

19.2 Config Name Style

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]_[batch_per_gpu x gpu]_
↪{schedule}_{dataset}
```

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

- `{model}`: model type like `hv_pointpillars` (Hard Voxelization PointPillars), `VoteNet`, etc.
- `[model setting]`: specific setting for some model.
- `{backbone}`: backbone type like `regnet-400mf`, `regnet-1.6gf`.
- `[neck]`: neck type like `fpn`, `secfpn`.

- `[norm_setting]`: `bn` (Batch Normalization) is used unless specified, other norm layer type could be `gn` (Group Normalization), `sbn` (Synchronized Batch Normalization). `gn-head`/`gn-neck` indicates GN is applied in head/neck only, while `gn-all` means GN is applied in the entire model, e.g. backbone, neck, head.
- `[misc]`: miscellaneous setting/plugins of model, e.g. `strong-aug` means using stronger augmentation strategies for training.
- `[batch_per_gpu x gpu]`: samples per GPU and GPUs, `4x8` is used by default.
- `{schedule}`: training schedule, options are `1x`, `2x`, `20e`, etc. `1x` and `2x` means 12 epochs and 24 epochs respectively. `20e` is adopted in cascade models, which denotes 20 epochs. For `1x`/`2x`, initial learning rate decays by a factor of 10 at the 8/16th and 11/22th epochs. For `20e`, initial learning rate decays by a factor of 10 at the 16th and 19th epochs.
- `{dataset}`: dataset like `nus-3d`, `kitti-3d`, `lyft-3d`, `scannet-3d`, `sunrgbd-3d`. We also indicate the number of classes we are using if there exist multiple settings, e.g., `kitti-3d-3class` and `kitti-3d-car` means training on KITTI dataset with 3 classes and single class, respectively.

19.3 Deprecated train_cfg/test_cfg

Following MMDetection, the `train_cfg` and `test_cfg` are deprecated in config file, please specify them in the model config. The original config structure is as below.

```
# deprecated
model = dict(
   type=...,
   ...
)
train_cfg=dict(...)
test_cfg=dict(...)
```

The migration example is as below.

```
# recommended
model = dict(
   type=...,
   ...
   train_cfg=dict(...),
   test_cfg=dict(...)
)
```

19.4 An example of VoteNet

```
model = dict(
    type='VoteNet',  # The type of detector, refer to mmdet3d.models.detectors for more␣
↪details
    backbone=dict(
        type='PointNet2SASSG',  # The type of the backbone refer to mmdet3d.models.
↪backbones for more details
        in_channels=4,  # Input channels of point cloud
        num_points=(2048, 1024, 512, 256),  # The number of points which each SA module␣
↪samples
```

(continues on next page)

(continued from previous page)

```
        radius=(0.2, 0.4, 0.8, 1.2),  # Radius for each set abstraction layer
        num_samples=(64, 32, 16, 16),  # Number of samples for each set abstraction layer
        sa_channels=((64, 64, 128), (128, 128, 256), (128, 128, 256),
                     (128, 128, 256)),  # Out channels of each mlp in SA module
        fp_channels=((256, 256), (256, 256)),  # Out channels of each mlp in FP module
        norm_cfg=dict(type='BN2d'),  # Config of normalization layer
        sa_cfg=dict(  # Config of point set abstraction (SA) module
            type='PointSAModule',  # type of SA module
            pool_mod='max',  # Pool method ('max' or 'avg') for SA modules
            use_xyz=True,  # Whether to use xyz as features during feature gathering
            normalize_xyz=True)),  # Whether to use normalized xyz as feature during␣
↪feature gathering
    bbox_head=dict(
        type='VoteHead',  # The type of bbox head, refer to mmdet3d.models.dense_heads␣
↪for more details
        num_classes=18,  # Number of classes for classification
        bbox_coder=dict(
            type='PartialBinBasedBBoxCoder',  # The type of bbox_coder, refer to mmdet3d.
↪core.bbox.coders for more details
            num_sizes=18,  # Number of size clusters
            num_dir_bins=1,   # Number of bins to encode direction angle
            with_rot=False,  # Whether the bbox is with rotation
            mean_sizes=[[0.76966727, 0.8116021, 0.92573744],
                        [1.876858, 1.8425595, 1.1931566],
                        [0.61328, 0.6148609, 0.7182701],
                        [1.3955007, 1.5121545, 0.83443564],
                        [0.97949594, 1.0675149, 0.6329687],
                        [0.531663, 0.5955577, 1.7500148],
                        [0.9624706, 0.72462326, 1.1481868],
                        [0.83221924, 1.0490936, 1.6875663],
                        [0.21132214, 0.4206159, 0.5372846],
                        [1.4440073, 1.8970833, 0.26985747],
                        [1.0294262, 1.4040797, 0.87554324],
                        [1.3766412, 0.65521795, 1.6813129],
                        [0.6650819, 0.71111923, 1.298853],
                        [0.41999173, 0.37906948, 1.7513971],
                        [0.59359556, 0.5912492, 0.73919016],
                        [0.50867593, 0.50656086, 0.30136237],
                        [1.1511526, 1.0546296, 0.49706793],
                        [0.47535285, 0.49249494, 0.5802117]]),  # Mean sizes for each␣
↪class, the order is consistent with class_names.
        vote_moudule_cfg=dict(  # Config of vote module branch, refer to mmdet3d.models.
↪model_utils for more details
            in_channels=256,  # Input channels for vote_module
            vote_per_seed=1,  # Number of votes to generate for each seed
            gt_per_seed=3,  # Number of gts for each seed
            conv_channels=(256, 256),  # Channels for convolution
            conv_cfg=dict(type='Conv1d'),  # Config of convolution
            norm_cfg=dict(type='BN1d'),  # Config of normalization
            norm_feats=True,  # Whether to normalize features
            vote_loss=dict(  # Config of the loss function for voting branch
                type='ChamferDistance',  # Type of loss for voting branch
```

(continues on next page)

(continued from previous page)

```
                mode='l1',  # Loss mode of voting branch
                reduction='none',  # Specifies the reduction to apply to the output
                loss_dst_weight=10.0)),  # Destination loss weight of the voting branch
        vote_aggregation_cfg=dict(  # Config of vote aggregation branch
            type='PointSAModule',  # type of vote aggregation module
            num_point=256,  # Number of points for the set abstraction layer in vote␣
↪aggregation branch
            radius=0.3,  # Radius for the set abstraction layer in vote aggregation␣
↪branch
            num_sample=16,  # Number of samples for the set abstraction layer in vote␣
↪aggregation branch
            mlp_channels=[256, 128, 128, 128],  # Mlp channels for the set abstraction␣
↪layer in vote aggregation branch
            use_xyz=True,  # Whether to use xyz
            normalize_xyz=True),  # Whether to normalize xyz
        feat_channels=(128, 128),  # Channels for feature convolution
        conv_cfg=dict(type='Conv1d'),  # Config of convolution
        norm_cfg=dict(type='BN1d'),  # Config of normalization
        objectness_loss=dict(  # Config of objectness loss
            type='CrossEntropyLoss',  # Type of loss
            class_weight=[0.2, 0.8],  # Class weight of the objectness loss
            reduction='sum',  # Specifies the reduction to apply to the output
            loss_weight=5.0),  # Loss weight of the objectness loss
        center_loss=dict(  # Config of center loss
            type='ChamferDistance',  # Type of loss
            mode='l2',  # Loss mode of center loss
            reduction='sum',  # Specifies the reduction to apply to the output
            loss_src_weight=10.0,  # Source loss weight of the voting branch.
            loss_dst_weight=10.0),  # Destination loss weight of the voting branch.
        dir_class_loss=dict(  # Config of direction classification loss
            type='CrossEntropyLoss',  # Type of loss
            reduction='sum',  # Specifies the reduction to apply to the output
            loss_weight=1.0),  # Loss weight of the direction classification loss
        dir_res_loss=dict(  # Config of direction residual loss
            type='SmoothL1Loss',  # Type of loss
            reduction='sum',  # Specifies the reduction to apply to the output
            loss_weight=10.0),  # Loss weight of the direction residual loss
        size_class_loss=dict(  # Config of size classification loss
            type='CrossEntropyLoss',  # Type of loss
            reduction='sum',  # Specifies the reduction to apply to the output
            loss_weight=1.0),  # Loss weight of the size classification loss
        size_res_loss=dict(  # Config of size residual loss
            type='SmoothL1Loss',  # Type of loss
            reduction='sum',  # Specifies the reduction to apply to the output
            loss_weight=3.3333333333333335),  # Loss weight of the size residual loss
        semantic_loss=dict(  # Config of semantic loss
            type='CrossEntropyLoss',  # Type of loss
            reduction='sum',  # Specifies the reduction to apply to the output
            loss_weight=1.0)),  # Loss weight of the semantic loss
    train_cfg = dict(  # Config of training hyperparameters for VoteNet
        pos_distance_thr=0.3,  # distance >= threshold 0.3 will be taken as positive␣
↪samples
```

(continues on next page)

(continued from previous page)

```
        neg_distance_thr=0.6,  # distance < threshold 0.6 will be taken as negative␣
↪samples
        sample_mod='vote'),  # Mode of the sampling method
    test_cfg = dict(  # Config of testing hyperparameters for VoteNet
        sample_mod='seed',  # Mode of the sampling method
        nms_thr=0.25,  # The threshold to be used during NMS
        score_thr=0.8,  # Threshold to filter out boxes
        per_class_proposal=False))  # Whether to use per_class_proposal
dataset_type = 'ScanNetDataset'  # Type of the dataset
data_root = './data/scannet/'  # Root path of the data
class_names = ('cabinet', 'bed', 'chair', 'sofa', 'table', 'door', 'window',
               'bookshelf', 'picture', 'counter', 'desk', 'curtain',
               'refrigerator', 'showercurtrain', 'toilet', 'sink', 'bathtub',
               'garbagebin')  # Names of classes
train_pipeline = [  # Training pipeline, refer to mmdet3d.datasets.pipelines for more␣
↪details
    dict(
        type='LoadPointsFromFile',  # First pipeline to load points, refer to mmdet3d.
↪datasets.pipelines.indoor_loading for more details
        shift_height=True,  # Whether to use shifted height
        load_dim=6,  # The dimension of the loaded points
        use_dim=[0, 1, 2]),  # Which dimensions of the points to be used
    dict(
        type='LoadAnnotations3D',  # Second pipeline to load annotations, refer to␣
↪mmdet3d.datasets.pipelines.indoor_loading for more details
        with_bbox_3d=True,  # Whether to load 3D boxes
        with_label_3d=True,  # Whether to load 3D labels corresponding to each 3D box
        with_mask_3d=True,  # Whether to load 3D instance masks
        with_seg_3d=True),  # Whether to load 3D semantic masks
    dict(
        type='PointSegClassMapping',  # Declare valid categories, refer to mmdet3d.
↪datasets.pipelines.point_seg_class_mapping for more details
        valid_cat_ids=(3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 24, 28, 33, 34,
                       36, 39),  # all valid categories ids
        max_cat_id=40),  # max possible category id in input segmentation mask
    dict(type='PointSample',  # Sample points, refer to mmdet3d.datasets.pipelines.
↪transforms_3d for more details
            num_points=40000),  # Number of points to be sampled
    dict(type='IndoorFlipData',  # Augmentation pipeline that flip points and 3d boxes
        flip_ratio_yz=0.5,  # Probability of being flipped along yz plane
        flip_ratio_xz=0.5),  # Probability of being flipped along xz plane
    dict(
        type='IndoorGlobalRotScale',  # Augmentation pipeline that rotate and scale␣
↪points and 3d boxes, refer to mmdet3d.datasets.pipelines.indoor_augment for more␣
↪details
        shift_height=True,  # Whether the loaded points use `shift_height` attribute
        rot_range=[-0.027777777777777776, 0.027777777777777776],  # Range of rotation
        scale_range=None),  # Range of scale
    dict(
        type='DefaultFormatBundle3D',  # Default format bundle to gather data in the␣
↪pipeline, refer to mmdet3d.datasets.pipelines.formatting for more details
        class_names=('cabinet', 'bed', 'chair', 'sofa', 'table', 'door',
```

(continues on next page)

(continued from previous page)

```
                    'window', 'bookshelf', 'picture', 'counter', 'desk',
                    'curtain', 'refrigerator', 'showercurtrain', 'toilet',
                    'sink', 'bathtub', 'garbagebin')),
    dict(
        type='Collect3D',  # Pipeline that decides which keys in the data should be␣
↪passed to the detector, refer to mmdet3d.datasets.pipelines.formatting for more details
        keys=[
            'points', 'gt_bboxes_3d', 'gt_labels_3d', 'pts_semantic_mask',
            'pts_instance_mask'
        ])
]
test_pipeline = [  # Testing pipeline, refer to mmdet3d.datasets.pipelines for more␣
↪details
    dict(
        type='LoadPointsFromFile',  # First pipeline to load points, refer to mmdet3d.
↪datasets.pipelines.indoor_loading for more details
        shift_height=True,  # Whether to use shifted height
        load_dim=6,  # The dimension of the loaded points
        use_dim=[0, 1, 2]),  # Which dimensions of the points to be used
    dict(type='PointSample',  # Sample points, refer to mmdet3d.datasets.pipelines.
↪transforms_3d for more details
        num_points=40000),  # Number of points to be sampled
    dict(
        type='DefaultFormatBundle3D',  # Default format bundle to gather data in the␣
↪pipeline, refer to mmdet3d.datasets.pipelines.formatting for more details
        class_names=('cabinet', 'bed', 'chair', 'sofa', 'table', 'door',
                    'window', 'bookshelf', 'picture', 'counter', 'desk',
                    'curtain', 'refrigerator', 'showercurtrain', 'toilet',
                    'sink', 'bathtub', 'garbagebin')),
    dict(type='Collect3D',  # Pipeline that decides which keys in the data should be␣
↪passed to the detector, refer to mmdet3d.datasets.pipelines.formatting for more details
        keys=['points'])
]
eval_pipeline = [  # Pipeline used for evaluation or visualization, refer to mmdet3d.
↪datasets.pipelines for more details
    dict(
        type='LoadPointsFromFile',  # First pipeline to load points, refer to mmdet3d.
↪datasets.pipelines.indoor_loading for more details
        shift_height=True,  # Whether to use shifted height
        load_dim=6,  # The dimension of the loaded points
        use_dim=[0, 1, 2]),  # Which dimensions of the points to be used
    dict(
        type='DefaultFormatBundle3D',  # Default format bundle to gather data in the␣
↪pipeline, refer to mmdet3d.datasets.pipelines.formatting for more details
        class_names=('cabinet', 'bed', 'chair', 'sofa', 'table', 'door',
                    'window', 'bookshelf', 'picture', 'counter', 'desk',
                    'curtain', 'refrigerator', 'showercurtrain', 'toilet',
                    'sink', 'bathtub', 'garbagebin')),
        with_label=False),
    dict(type='Collect3D',  # Pipeline that decides which keys in the data should be␣
↪passed to the detector, refer to mmdet3d.datasets.pipelines.formatting for more details
        keys=['points'])
```

(continues on next page)

(continued from previous page)

```
]
data = dict(
    samples_per_gpu=8,  # Batch size of a single GPU
    workers_per_gpu=4,  # Number of workers to pre-fetch data for each single GPU
    train=dict(  # Train dataset config
        type='RepeatDataset',  # Wrapper of dataset, refer to https://github.com/open-
↪mmlab/mmdetection/blob/master/mmdet/datasets/dataset_wrappers.py for details.
        times=5,  # Repeat times
        dataset=dict(
            type='ScanNetDataset',  # Type of dataset
            data_root='./data/scannet/',  # Root path of the data
            ann_file='./data/scannet/scannet_infos_train.pkl',  # Ann path of the data
            pipeline=[  # pipeline, this is passed by the train_pipeline created before.
                dict(
                    type='LoadPointsFromFile',
                    shift_height=True,
                    load_dim=6,
                    use_dim=[0, 1, 2]),
                dict(
                    type='LoadAnnotations3D',
                    with_bbox_3d=True,
                    with_label_3d=True,
                    with_mask_3d=True,
                    with_seg_3d=True),
                dict(
                    type='PointSegClassMapping',
                    valid_cat_ids=(3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 24,
                                   28, 33, 34, 36, 39),
                    max_cat_id=40),
                dict(type='PointSample', num_points=40000),
                dict(
                    type='IndoorFlipData',
                    flip_ratio_yz=0.5,
                    flip_ratio_xz=0.5),
                dict(
                    type='IndoorGlobalRotScale',
                    shift_height=True,
                    rot_range=[-0.02777777777777776, 0.02777777777777776],
                    scale_range=None),
                dict(
                    type='DefaultFormatBundle3D',
                    class_names=('cabinet', 'bed', 'chair', 'sofa', 'table',
                                 'door', 'window', 'bookshelf', 'picture',
                                 'counter', 'desk', 'curtain', 'refrigerator',
                                 'showercurtrain', 'toilet', 'sink', 'bathtub',
                                 'garbagebin')),
                dict(
                    type='Collect3D',
                    keys=[
                        'points', 'gt_bboxes_3d', 'gt_labels_3d',
                        'pts_semantic_mask', 'pts_instance_mask'
                    ])
```

(continues on next page)

(continued from previous page)

```
            ],
            filter_empty_gt=False,  # Whether to filter empty ground truth boxes
            classes=('cabinet', 'bed', 'chair', 'sofa', 'table', 'door',
                     'window', 'bookshelf', 'picture', 'counter', 'desk',
                     'curtain', 'refrigerator', 'showercurtrain', 'toilet',
                     'sink', 'bathtub', 'garbagebin'))),  # Names of classes
    val=dict(  # Validation dataset config
        type='ScanNetDataset',  # Type of dataset
        data_root='./data/scannet/',  # Root path of the data
        ann_file='./data/scannet/scannet_infos_val.pkl',  # Ann path of the data
        pipeline=[  # Pipeline is passed by test_pipeline created before
            dict(
                type='LoadPointsFromFile',
                shift_height=True,
                load_dim=6,
                use_dim=[0, 1, 2]),
            dict(type='PointSample', num_points=40000),
            dict(
                type='DefaultFormatBundle3D',
                class_names=('cabinet', 'bed', 'chair', 'sofa', 'table',
                             'door', 'window', 'bookshelf', 'picture',
                             'counter', 'desk', 'curtain', 'refrigerator',
                             'showercurtrain', 'toilet', 'sink', 'bathtub',
                             'garbagebin')),
            dict(type='Collect3D', keys=['points'])
        ],
        classes=('cabinet', 'bed', 'chair', 'sofa', 'table', 'door', 'window',
                 'bookshelf', 'picture', 'counter', 'desk', 'curtain',
                 'refrigerator', 'showercurtrain', 'toilet', 'sink', 'bathtub',
                 'garbagebin'),  # Names of classes
        test_mode=True),  # Whether to use test mode
    test=dict(  # Test dataset config
        type='ScanNetDataset',  # Type of dataset
        data_root='./data/scannet/',  # Root path of the data
        ann_file='./data/scannet/scannet_infos_val.pkl',  # Ann path of the data
        pipeline=[  # Pipeline is passed by test_pipeline created before
            dict(
                type='LoadPointsFromFile',
                shift_height=True,
                load_dim=6,
                use_dim=[0, 1, 2]),
            dict(type='PointSample', num_points=40000),
            dict(
                type='DefaultFormatBundle3D',
                class_names=('cabinet', 'bed', 'chair', 'sofa', 'table',
                             'door', 'window', 'bookshelf', 'picture',
                             'counter', 'desk', 'curtain', 'refrigerator',
                             'showercurtrain', 'toilet', 'sink', 'bathtub',
                             'garbagebin')),
            dict(type='Collect3D', keys=['points'])
        ],
        classes=('cabinet', 'bed', 'chair', 'sofa', 'table', 'door', 'window',
```

(continues on next page)

(continued from previous page)

```
                'bookshelf', 'picture', 'counter', 'desk', 'curtain',
                'refrigerator', 'showercurtrain', 'toilet', 'sink', 'bathtub',
                'garbagebin'),  # Names of classes
        test_mode=True))  # Whether to use test mode
evaluation = dict(pipeline=[  # Pipeline is passed by eval_pipeline created before
    dict(
        type='LoadPointsFromFile',
        coord_type='DEPTH',
        shift_height=False,
        load_dim=6,
        use_dim=[0, 1, 2]),
    dict(
        type='DefaultFormatBundle3D',
        class_names=('cabinet', 'bed', 'chair', 'sofa', 'table', 'door',
                     'window', 'bookshelf', 'picture', 'counter', 'desk',
                     'curtain', 'refrigerator', 'showercurtrain', 'toilet',
                     'sink', 'bathtub', 'garbagebin'),
        with_label=False),
    dict(type='Collect3D', keys=['points'])
])
lr = 0.008  # Learning rate of optimizers
optimizer = dict(  # Config used to build optimizer, support all the optimizers in␣
↪PyTorch whose arguments are also the same as those in PyTorch
    type='Adam',  # Type of optimizers, refer to https://github.com/open-mmlab/mmcv/blob/
↪v1.3.7/mmcv/runner/optimizer/default_constructor.py#L12 for more details
    lr=0.008)  # Learning rate of optimizers, see detail usages of the parameters in the␣
↪documentation of PyTorch
optimizer_config = dict(  # Config used to build the optimizer hook, refer to https://
↪github.com/open-mmlab/mmcv/blob/v1.3.7/mmcv/runner/hooks/optimizer.py#L22 for␣
↪implementation details.
    grad_clip=dict(  # Config used to grad_clip
    max_norm=10,  # max norm of the gradients
    norm_type=2))  # Type of the used p-norm. Can be 'inf' for infinity norm.
lr_config = dict(  # Learning rate scheduler config used to register LrUpdater hook
    policy='step',  # The policy of scheduler, also support CosineAnnealing, Cyclic, etc.
↪ Refer to details of supported LrUpdater from https://github.com/open-mmlab/mmcv/blob/
↪v1.3.7/mmcv/runner/hooks/lr_updater.py#L9.
    warmup=None,  # The warmup policy, also support `exp` and `constant`.
    step=[24, 32])  # Steps to decay the learning rate
checkpoint_config = dict(  # Config of set the checkpoint hook, Refer to https://github.
↪com/open-mmlab/mmcv/blob/master/mmcv/runner/hooks/checkpoint.py for implementation.
    interval=1)  # The save interval is 1
log_config = dict(  # config of register logger hook
    interval=50,  # Interval to print the log
    hooks=[dict(type='TextLoggerHook'),
           dict(type='TensorboardLoggerHook')])  # The logger used to record the␣
↪training process.
runner = dict(type='EpochBasedRunner', max_epochs=36) # Runner that runs the `workflow`␣
↪in total `max_epochs`
dist_params = dict(backend='nccl')  # Parameters to setup distributed training, the port␣
↪can also be set.
log_level = 'INFO'  # The level of logging.
```

(continues on next page)

(continued from previous page)

```
find_unused_parameters = True  # Whether to find unused parameters
work_dir = None  # Directory to save the model checkpoints and logs for the current␣
↪experiments.
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. The training state such as the epoch␣
↪number and optimizer state will be restored.
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 36 epochs according to the max_epochs.
gpu_ids = range(0, 1)  # ids of gpus
```

19.5 FAQ

19.5.1 Ignore some fields in the base configs

Sometimes, you may set `_delete_=True` to ignore some of fields in base configs. You may refer to mmcv for simple illustration.

In MMDetection3D, for example, to change the FPN neck of PointPillars with the following config.

```
model = dict(
    type='MVXFasterRCNN',
    pts_voxel_layer=dict(...),
    pts_voxel_encoder=dict(...),
    pts_middle_encoder=dict(...),
    pts_backbone=dict(...),
    pts_neck=dict(
        type='FPN',
        norm_cfg=dict(type='naiveSyncBN2d', eps=1e-3, momentum=0.01),
        act_cfg=dict(type='ReLU'),
        in_channels=[64, 128, 256],
        out_channels=256,
        start_level=0,
        num_outs=3),
    pts_bbox_head=dict(...))
```

`FPN` and `SECONDFPN` use different keywords to construct.

```
_base_ = '../_base_/models/hv_pointpillars_fpn_nus.py'
model = dict(
    pts_neck=dict(
        _delete_=True,
        type='SECONDFPN',
        norm_cfg=dict(type='naiveSyncBN2d', eps=1e-3, momentum=0.01),
        in_channels=[64, 128, 256],
        upsample_strides=[1, 2, 4],
        out_channels=[128, 128, 128]),
    pts_bbox_head=dict(...))
```

The `_delete_=True` would replace all old keys in `pts_neck` field with new keys.

19.5.2 Use intermediate variables in configs

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

```
_base_ = './nus-3d.py'
train_pipeline = [
    dict(
        type='LoadPointsFromFile',
        load_dim=5,
        use_dim=5,
        file_client_args=file_client_args),
    dict(
        type='LoadPointsFromMultiSweeps',
        sweeps_num=10,
        file_client_args=file_client_args),
    dict(type='LoadAnnotations3D', with_bbox_3d=True, with_label_3d=True),
    dict(
        type='GlobalRotScaleTrans',
        rot_range=[-0.3925, 0.3925],
        scale_ratio_range=[0.95, 1.05],
        translation_std=[0, 0, 0]),
    dict(type='RandomFlip3D', flip_ratio_bev_horizontal=0.5),
    dict(type='PointsRangeFilter', point_cloud_range=point_cloud_range),
    dict(type='ObjectRangeFilter', point_cloud_range=point_cloud_range),
    dict(type='ObjectNameFilter', classes=class_names),
    dict(type='PointShuffle'),
    dict(type='DefaultFormatBundle3D', class_names=class_names),
    dict(type='Collect3D', keys=['points', 'gt_bboxes_3d', 'gt_labels_3d'])
]
test_pipeline = [
    dict(
        type='LoadPointsFromFile',
        load_dim=5,
        use_dim=5,
        file_client_args=file_client_args),
    dict(
        type='LoadPointsFromMultiSweeps',
        sweeps_num=10,
        file_client_args=file_client_args),
    dict(
        type='MultiScaleFlipAug3D',
        img_scale=(1333, 800),
        pts_scale_ratio=[0.95, 1.0, 1.05],
        flip=False,
        transforms=[
            dict(
                type='GlobalRotScaleTrans',
                rot_range=[0, 0],
```

(continues on next page)

(continued from previous page)

```
                scale_ratio_range=[1., 1.],
                translation_std=[0, 0, 0]),
            dict(type='RandomFlip3D'),
            dict(
                type='PointsRangeFilter', point_cloud_range=point_cloud_range),
            dict(
                type='DefaultFormatBundle3D',
                class_names=class_names,
                with_label=False),
            dict(type='Collect3D', keys=['points'])
        ])
]
data = dict(
    train=dict(pipeline=train_pipeline),
    val=dict(pipeline=test_pipeline),
    test=dict(pipeline=test_pipeline))
```

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

CHAPTER

TWENTY

TUTORIAL 2: CUSTOMIZE DATASETS

20.1 Support new data format

To support a new data format, you can either convert them to existing formats or directly convert them to the middle format. You could also choose to convert them offline (before training by a script) or online (implement a new dataset and do the conversion at training). In MMDetection3D, for the data that is inconvenient to read directly online, we recommend to convert it into KITTI format and do the conversion offline, thus you only need to modify the config's data annotation paths and classes after the conversion. For data sharing similar format with existing datasets, like Lyft compared to nuScenes, we recommend to directly implement data converter and dataset class. During the procedure, inheritation could be taken into consideration to reduce the implementation workload.

20.1.1 Reorganize new data formats to existing format

For data that is inconvenient to read directly online, the simplest way is to convert your dataset to existing dataset formats.

Typically we need a data converter to reorganize the raw data and convert the annotation format into KITTI style. Then a new dataset class inherited from existing ones is sometimes necessary for dealing with some specific differences between datasets. Finally, the users need to further modify the config files to use the dataset. An example training predefined models on Waymo dataset by converting it into KITTI style can be taken for reference.

20.1.2 Reorganize new data format to middle format

It is also fine if you do not want to convert the annotation format to existing formats. Actually, we convert all the supported datasets into pickle files, which summarize useful information for model training and inference.

The annotation of a dataset is a list of dict, each dict corresponds to a frame. A basic example (used in KITTI) is as follows. A frame consists of several keys, like `image`, `point_cloud`, `calib` and `annos`. As long as we could directly read data according to these information, the organization of raw data could also be different from existing ones. With this design, we provide an alternative choice for customizing datasets.

```
[
    {'image': {'image_idx': 0, 'image_path': 'training/image_2/000000.png', 'image_shape
↪': array([ 370, 1224], dtype=int32)},
     'point_cloud': {'num_features': 4, 'velodyne_path': 'training/velodyne/000000.bin'},
     'calib': {'P0': array([[707.0493,   0.    , 604.0814,   0.    ],
       [  0.    , 707.0493, 180.5066,   0.    ],
       [  0.    ,   0.    ,   1.    ,   0.    ],
       [  0.    ,   0.    ,   0.    ,   1.    ]]),
```

(continues on next page)

(continued from previous page)

```
       'P1': array([[ 707.0493,    0.    ,  604.0814, -379.7842],
       [   0.    ,  707.0493,  180.5066,    0.    ],
       [   0.    ,    0.    ,    1.    ,    0.    ],
       [   0.    ,    0.    ,    0.    ,    1.    ]]),
       'P2': array([[ 7.070493e+02,  0.000000e+00,  6.040814e+02,  4.575831e+01],
       [ 0.000000e+00,  7.070493e+02,  1.805066e+02, -3.454157e-01],
       [ 0.000000e+00,  0.000000e+00,  1.000000e+00,  4.981016e-03],
       [ 0.000000e+00,  0.000000e+00,  0.000000e+00,  1.000000e+00]]),
       'P3': array([[ 7.070493e+02,  0.000000e+00,  6.040814e+02, -3.341081e+02],
       [ 0.000000e+00,  7.070493e+02,  1.805066e+02,  2.330660e+00],
       [ 0.000000e+00,  0.000000e+00,  1.000000e+00,  3.201153e-03],
       [ 0.000000e+00,  0.000000e+00,  0.000000e+00,  1.000000e+00]]),
       'R0_rect': array([[ 0.9999128 ,  0.01009263, -0.00851193,  0.        ],
       [-0.01012729,  0.9999406 , -0.00403767,  0.        ],
       [ 0.00847068,  0.00412352,  0.9999556 ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  1.        ]]),
       'Tr_velo_to_cam': array([[ 0.00692796, -0.9999722 , -0.00275783, -0.02457729],
       [-0.00116298,  0.00274984, -0.9999955 , -0.06127237],
       [ 0.9999753 ,  0.00693114, -0.0011439 , -0.3321029 ],
       [ 0.        ,  0.        ,  0.        ,  1.        ]]),
       'Tr_imu_to_velo': array([[ 9.999976e-01,  7.553071e-04, -2.035826e-03, -8.086759e-
↪01],
       [-7.854027e-04,  9.998898e-01, -1.482298e-02,  3.195559e-01],
       [ 2.024406e-03,  1.482454e-02,  9.998881e-01, -7.997231e-01],
       [ 0.000000e+00,  0.000000e+00,  0.000000e+00,  1.000000e+00]])},
     'annos': {'name': array(['Pedestrian'], dtype='<U10'), 'truncated': array([0.]),
↪'occluded': array([0]), 'alpha': array([-0.2]), 'bbox': array([[712.4 , 143.  , 810.73,
↪ 307.92]]), 'dimensions': array([[1.2 , 1.89, 0.48]]), 'location': array([[1.84, 1.47,␣
↪8.41]]), 'rotation_y': array([0.01]), 'score': array([0.]), 'index': array([0],␣
↪dtype=int32), 'group_ids': array([0], dtype=int32), 'difficulty': array([0],␣
↪dtype=int32), 'num_points_in_gt': array([377], dtype=int32)}}
    ...
]
```

On top of this you can write a new Dataset class inherited from `Custom3DDataset`, and overwrite related methods, like KittiDataset and ScanNetDataset.

20.1.3 An example of customized dataset

Here we provide an example of customized dataset.

Assume the annotation has been reorganized into a list of dict in pickle files like ScanNet. The bounding boxes annotations are stored in `annotation.pkl` as the following

```
{'point_cloud': {'num_features': 6, 'lidar_idx': 'scene0000_00'}, 'pts_path': 'points/
↪scene0000_00.bin',
 'pts_instance_mask_path': 'instance_mask/scene0000_00.bin', 'pts_semantic_mask_path':
↪'semantic_mask/scene0000_00.bin',
 'annos': {'gt_num': 27, 'name': array(['window', 'window', 'table', 'counter', 'curtain
↪', 'curtain',
       'desk', 'cabinet', 'sink', 'garbagebin', 'garbagebin',
       'garbagebin', 'sofa', 'refrigerator', 'table', 'table', 'toilet',
```

(continues on next page)

(continued from previous page)

```
       'bed', 'cabinet', 'cabinet', 'cabinet', 'cabinet', 'cabinet',
       'cabinet', 'door', 'door', 'door'], dtype='<U12'),
       'location': array([[ 1.48129511,  3.52074146,  1.85652947],
       [ 2.90395617, -3.48033905,  1.52682471]]),
       'dimensions': array([[1.74445975, 0.23195696, 0.57235193],
       [0.66077662, 0.17072392, 0.67153597]]),
       'gt_boxes_upright_depth': array([
       [ 1.48129511,  3.52074146,  1.85652947,  1.74445975,  0.23195696,
         0.57235193],
       [ 2.90395617, -3.48033905,  1.52682471,  0.66077662,  0.17072392,
         0.67153597]]),
       'index': array([ 0,  1 ], dtype=int32),
       'class': array([ 6,  6 ])}}
```

We can create a new dataset in `mmdet3d/datasets/my_dataset.py` to load the data.

```
import numpy as np
from os import path as osp

from mmdet3d.core import show_result
from mmdet3d.core.bbox import DepthInstance3DBoxes
from mmdet.datasets import DATASETS
from .custom_3d import Custom3DDataset


@DATASETS.register_module()
class MyDataset(Custom3DDataset):
    CLASSES = ('cabinet', 'bed', 'chair', 'sofa', 'table', 'door', 'window',
               'bookshelf', 'picture', 'counter', 'desk', 'curtain',
               'refrigerator', 'showercurtrain', 'toilet', 'sink', 'bathtub',
               'garbagebin')

    def __init__(self,
                 data_root,
                 ann_file,
                 pipeline=None,
                 classes=None,
                 modality=None,
                 box_type_3d='Depth',
                 filter_empty_gt=True,
                 test_mode=False):
        super().__init__(
            data_root=data_root,
            ann_file=ann_file,
            pipeline=pipeline,
            classes=classes,
            modality=modality,
            box_type_3d=box_type_3d,
            filter_empty_gt=filter_empty_gt,
            test_mode=test_mode)

    def get_ann_info(self, index):
```

(continues on next page)

(continued from previous page)

```
        # Use index to get the annos, thus the evalhook could also use this api
        info = self.data_infos[index]
        if info['annos']['gt_num'] != 0:
            gt_bboxes_3d = info['annos']['gt_boxes_upright_depth'].astype(
                np.float32)  # k, 6
            gt_labels_3d = info['annos']['class'].astype(np.int64)
        else:
            gt_bboxes_3d = np.zeros((0, 6), dtype=np.float32)
            gt_labels_3d = np.zeros((0, ), dtype=np.int64)

        # to target box structure
        gt_bboxes_3d = DepthInstance3DBoxes(
            gt_bboxes_3d,
            box_dim=gt_bboxes_3d.shape[-1],
            with_yaw=False,
            origin=(0.5, 0.5, 0.5)).convert_to(self.box_mode_3d)

        pts_instance_mask_path = osp.join(self.data_root,
                                          info['pts_instance_mask_path'])
        pts_semantic_mask_path = osp.join(self.data_root,
                                          info['pts_semantic_mask_path'])

        anns_results = dict(
            gt_bboxes_3d=gt_bboxes_3d,
            gt_labels_3d=gt_labels_3d,
            pts_instance_mask_path=pts_instance_mask_path,
            pts_semantic_mask_path=pts_semantic_mask_path)
        return anns_results
```

Then in the config, to use `MyDataset` you can modify the config as the following

```
dataset_A_train = dict(
    type='MyDataset',
    ann_file = 'annotation.pkl',
    pipeline=train_pipeline
)
```

20.2 Customize datasets by dataset wrappers

MMDetection3D also supports many dataset wrappers to mix the dataset or modify the dataset distribution for training like MMDetection. 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.

20.2.1 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
        )
    )
```

20.2.2 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
        )
    )
```

You may refer to source code for details.

20.2.3 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'],
   ```

(continues on next page)

(continued from previous page)

```
    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,
```

(continues on next page)

(continued from previous page)

```
    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
)
```

20.3 Modify Dataset Classes

With existing dataset types, we can modify the class names of them to train subset of the annotations. For example, if you want to train only three classes of the current dataset, you can modify the classes of dataset. The dataset will filter out the ground truth boxes of other classes automatically.

```
classes = ('person', 'bicycle', 'car')
data = dict(
    train=dict(classes=classes),
    val=dict(classes=classes),
    test=dict(classes=classes))
```

MMDetection V2.0 also supports to read the classes from a file, which is common in real applications. For example, assume the `classes.txt` contains the name of classes as the following.

```
person
bicycle
car
```

Users can set the classes as a file path, the dataset will load it and convert it to a list automatically.

```
classes = 'path/to/classes.txt'
data = dict(
    train=dict(classes=classes),
    val=dict(classes=classes),
    test=dict(classes=classes))
```

Note (related to MMDetection):

- Before MMDetection v2.5.0, the dataset will filter out the empty GT images automatically if the classes are set and there is no way to disable that through config. This is an undesirable behavior and introduces confusion because if the classes are not set, the dataset only filter the empty GT images when `filter_empty_gt=True` and `test_mode=False`. After MMDetection v2.5.0, we decouple the image filtering process and the classes modification, i.e., the dataset will only filter empty GT images when `filter_empty_gt=True` and `test_mode=False`, no matter whether the classes are set. Thus, setting the classes only influences the annotations of classes used for training and users could decide whether to filter empty GT images by themselves.
- Since the middle format only has box labels and does not contain the class names, when using `CustomDataset`, users cannot filter out the empty GT images through configs but only do this offline.
- The features for setting dataset classes and dataset filtering will be refactored to be more user-friendly in the future (depends on the progress).

CHAPTER

TWENTYONE

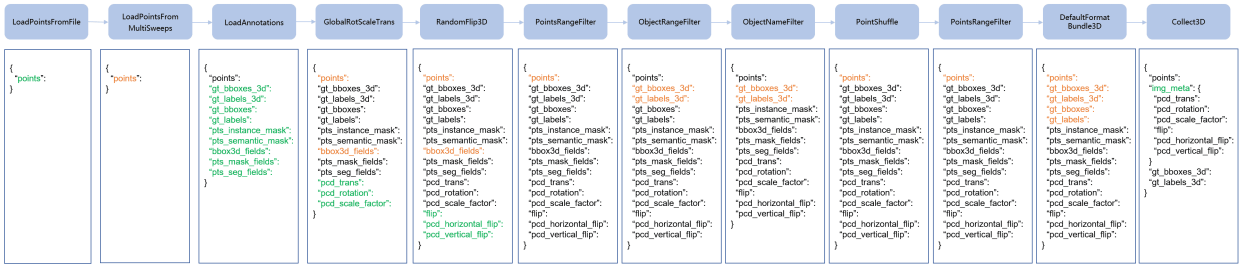
TUTORIAL 3: CUSTOMIZE DATA PIPELINES

21.1 Design of Data pipelines

Following typical conventions, we use `Dataset` and `DataLoader` for data loading with multiple workers. `Dataset` returns a dict of data items corresponding the arguments of models' forward method. Since the data in object detection may not be the same size (point number, gt bbox size, etc.), we introduce a new `DataContainer` type in MMCV to help collect and distribute data of different size. See here for more details.

The data preparation pipeline and the dataset is decomposed. Usually a dataset defines how to process the annotations and a data pipeline defines all the steps to prepare a data dict. A pipeline consists of a sequence of operations. Each operation takes a dict as input and also output a dict for the next transform.

We present a classical pipeline in the following figure. The blue blocks are pipeline operations. With the pipeline going on, each operator can add new keys (marked as green) to the result dict or update the existing keys (marked as orange).



The operations are categorized into data loading, pre-processing, formatting and test-time augmentation.

Here is an pipeline example for PointPillars.

```
train_pipeline = [
    dict(
        type='LoadPointsFromFile',
        load_dim=5,
        use_dim=5,
        file_client_args=file_client_args),
    dict(
        type='LoadPointsFromMultiSweeps',
        sweeps_num=10,
        file_client_args=file_client_args),
    dict(type='LoadAnnotations3D', with_bbox_3d=True, with_label_3d=True),
    dict(
        type='GlobalRotScaleTrans',
        rot_range=[-0.3925, 0.3925],
        scale_ratio_range=[0.95, 1.05],
```

(continues on next page)

(continued from previous page)

```
        translation_std=[0, 0, 0]),
    dict(type='RandomFlip3D', flip_ratio_bev_horizontal=0.5),
    dict(type='PointsRangeFilter', point_cloud_range=point_cloud_range),
    dict(type='ObjectRangeFilter', point_cloud_range=point_cloud_range),
    dict(type='ObjectNameFilter', classes=class_names),
    dict(type='PointShuffle'),
    dict(type='DefaultFormatBundle3D', class_names=class_names),
    dict(type='Collect3D', keys=['points', 'gt_bboxes_3d', 'gt_labels_3d'])
]
test_pipeline = [
    dict(
        type='LoadPointsFromFile',
        load_dim=5,
        use_dim=5,
        file_client_args=file_client_args),
    dict(
        type='LoadPointsFromMultiSweeps',
        sweeps_num=10,
        file_client_args=file_client_args),
    dict(
        type='MultiScaleFlipAug',
        img_scale=(1333, 800),
        pts_scale_ratio=1.0,
        flip=False,
        pcd_horizontal_flip=False,
        pcd_vertical_flip=False,
        transforms=[
            dict(
                type='GlobalRotScaleTrans',
                rot_range=[0, 0],
                scale_ratio_range=[1., 1.],
                translation_std=[0, 0, 0]),
            dict(type='RandomFlip3D'),
            dict(
                type='PointsRangeFilter', point_cloud_range=point_cloud_range),
            dict(
                type='DefaultFormatBundle3D',
                class_names=class_names,
                with_label=False),
            dict(type='Collect3D', keys=['points'])
        ])
]
```

For each operation, we list the related dict fields that are added/updated/removed.

21.1.1 Data loading

`LoadPointsFromFile`

- add: points

`LoadPointsFromMultiSweeps`

- update: points

`LoadAnnotations3D`

- add: gt_bboxes_3d, gt_labels_3d, gt_bboxes, gt_labels, pts_instance_mask, pts_semantic_mask, bbox3d_fields, pts_mask_fields, pts_seg_fields

21.1.2 Pre-processing

`GlobalRotScaleTrans`

- add: pcd_trans, pcd_rotation, pcd_scale_factor
- update: points, *bbox3d_fields

`RandomFlip3D`

- add: flip, pcd_horizontal_flip, pcd_vertical_flip
- update: points, *bbox3d_fields

`PointsRangeFilter`

- update: points

`ObjectRangeFilter`

- update: gt_bboxes_3d, gt_labels_3d

`ObjectNameFilter`

- update: gt_bboxes_3d, gt_labels_3d

`PointShuffle`

- update: points

`PointsRangeFilter`

- update: points

21.1.3 Formatting

`DefaultFormatBundle3D`

- update: points, gt_bboxes_3d, gt_labels_3d, gt_bboxes, gt_labels

`Collect3D`

- add: img_meta (the keys of img_meta is specified by `meta_keys`)
- remove: all other keys except for those specified by `keys`

21.1.4 Test time augmentation

`MultiScaleFlipAug`

- update: scale, pcd_scale_factor, flip, flip_direction, pcd_horizontal_flip, pcd_vertical_flip with list of augmented data with these specific parameters

21.2 Extend and use custom pipelines

1. Write a new pipeline in any file, e.g., `my_pipeline.py`. It takes a dict as input and return a dict.

```
from mmdet.datasets import PIPELINES

@PIPELINES.register_module()
class MyTransform:

    def __call__(self, results):
        results['dummy'] = True
        return results
```

2. Import the new class.

```
from .my_pipeline import MyTransform
```

3. Use it in config files.

```
train_pipeline = [
    dict(
        type='LoadPointsFromFile',
        load_dim=5,
        use_dim=5,
        file_client_args=file_client_args),
    dict(
        type='LoadPointsFromMultiSweeps',
        sweeps_num=10,
        file_client_args=file_client_args),
    dict(type='LoadAnnotations3D', with_bbox_3d=True, with_label_3d=True),
    dict(
        type='GlobalRotScaleTrans',
        rot_range=[-0.3925, 0.3925],
        scale_ratio_range=[0.95, 1.05],
        translation_std=[0, 0, 0]),
    dict(type='RandomFlip3D', flip_ratio_bev_horizontal=0.5),
    dict(type='PointsRangeFilter', point_cloud_range=point_cloud_range),
    dict(type='ObjectRangeFilter', point_cloud_range=point_cloud_range),
    dict(type='ObjectNameFilter', classes=class_names),
    dict(type='MyTransform'),
    dict(type='PointShuffle'),
    dict(type='DefaultFormatBundle3D', class_names=class_names),
    dict(type='Collect3D', keys=['points', 'gt_bboxes_3d', 'gt_labels_3d'])
]
```

CHAPTER

TWENTYTWO

TUTORIAL 4: CUSTOMIZE MODELS

We basically categorize model components into 6 types.

- encoder: including voxel layer, voxel encoder and middle encoder used in voxel-based methods before backbone, e.g., HardVFE and PointPillarsScatter.
- backbone: usually an FCN network to extract feature maps, e.g., ResNet, SECOND.
- neck: the component between backbones and heads, e.g., FPN, SECONDFPN.
- head: the component for specific tasks, e.g., bbox prediction and mask prediction.
- RoI extractor: the part for extracting RoI features from feature maps, e.g., H3DRoIHead and PartAggregationROIHead.
- loss: the component in heads for calculating losses, e.g., FocalLoss, L1Loss, and GHMLoss.

22.1 Develop new components

22.1.1 Add a new encoder

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

1. Define a new voxel encoder (e.g. HardVFE: Voxel feature encoder used in DV-SECOND)

Create a new file `mmdet3d/models/voxel_encoders/voxel_encoder.py`.

```
import torch.nn as nn

from ..builder import VOXEL_ENCODERS


@VOXEL_ENCODERS.register_module()
class HardVFE(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 `mmdet3d/models/voxel_encoders/__init__.py`

```
from .voxel_encoder import HardVFE
```

or alternatively add

```
custom_imports = dict(
    imports=['mmdet3d.models.voxel_encoders.HardVFE'],
    allow_failed_imports=False)
```

to the config file to avoid modifying the original code.

3. Use the voxel encoder in your config file

```
model = dict(
    ...
    voxel_encoder=dict(
        type='HardVFE',
        arg1=xxx,
        arg2=xxx),
    ...
```

22.1.2 Add a new backbone

Here we show how to develop new components with an example of SECOND (Sparsely Embedded Convolutional Detection).

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

Create a new file `mmdet3d/models/backbones/second.py`.

```
import torch.nn as nn

from ..builder import BACKBONES


@BACKBONES.register_module()
class SECOND(BaseModule):

    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 `mmdet3d/models/backbones/__init__.py`

```
from .second import SECOND
```

or alternatively add

```
custom_imports = dict(
    imports=['mmdet3d.models.backbones.second'],
    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='SECOND',
        arg1=xxx,
        arg2=xxx),
    ...
```

22.1.3 Add new necks

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

Create a new file `mmdet3d/models/necks/second_fpn.py`.

```
from ..builder import NECKS

@NECKS.register
class SECONDFPN(BaseModule):

    def __init__(self,
                 in_channels=[128, 128, 256],
                 out_channels=[256, 256, 256],
                 upsample_strides=[1, 2, 4],
                 norm_cfg=dict(type='BN', eps=1e-3, momentum=0.01),
                 upsample_cfg=dict(type='deconv', bias=False),
                 conv_cfg=dict(type='Conv2d', bias=False),
                 use_conv_for_no_stride=False,
                 init_cfg=None):
        pass

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

2. Import the module

You can either add the following line to `mmdet3D/models/necks/__init__.py`,

```
from .second_fpn import SECONDFPN
```

or alternatively add

```
custom_imports = dict(
    imports=['mmdet3d.models.necks.second_fpn'],
    allow_failed_imports=False)
```

to the config file and avoid modifying the original code.

3. Use the neck in your config file

```
model = dict(
    ...
    neck=dict(
        type='SECONDFPN',
        in_channels=[64, 128, 256],
        upsample_strides=[1, 2, 4],
        out_channels=[128, 128, 128]),
    ...
```

22.1.4 Add new heads

Here we show how to develop a new head with the example of PartA2 Head as the following.

Note: Here the example of PartA2 RoI Head is used in the second stage. For one-stage heads, please refer to examples in `mmdet3d/models/dense_heads/`. They are more commonly used in 3D detection for autonomous driving due to its simplicity and high efficiency.

First, add a new bbox head in `mmdet3d/models/roi_heads/bbox_heads/parta2_bbox_head.py`. PartA2 RoI Head 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. Sometimes other related functions like `loss` and `get_targets` are also required.

```
from mmdet.models.builder import HEADS
from .bbox_head import BBoxHead

@HEADS.register_module()
class PartA2BboxHead(BaseModule):
    """PartA2 RoI head."""

    def __init__(self,
                 num_classes,
                 seg_in_channels,
                 part_in_channels,
                 seg_conv_channels=None,
                 part_conv_channels=None,
                 merge_conv_channels=None,
```

(continues on next page)

(continued from previous page)

```
                 down_conv_channels=None,
                 shared_fc_channels=None,
                 cls_channels=None,
                 reg_channels=None,
                 dropout_ratio=0.1,
                 roi_feat_size=14,
                 with_corner_loss=True,
                 bbox_coder=dict(type='DeltaXYZWLHRBBoxCoder'),
                 conv_cfg=dict(type='Conv1d'),
                 norm_cfg=dict(type='BN1d', eps=1e-3, momentum=0.01),
                 loss_bbox=dict(
                     type='SmoothL1Loss', beta=1.0 / 9.0, loss_weight=2.0),
                 loss_cls=dict(
                     type='CrossEntropyLoss',
                     use_sigmoid=True,
                     reduction='none',
                     loss_weight=1.0),
                 init_cfg=None):
        super(PartA2BboxHead, self).__init__(init_cfg=init_cfg)

    def forward(self, seg_feats, part_feats):
```

Second, implement a new RoI Head if it is necessary. We plan to inherit the new `PartAggregationROIHead` from `Base3DRoIHead`. We can find that a `Base3DRoIHead` already implements the following functions.

```
from abc import ABCMeta, abstractmethod
from torch import nn as nn


@HEADS.register_module()
class Base3DRoIHead(BaseModule, metaclass=ABCMeta):
    """Base class for 3d RoIHeads."""

    def __init__(self,
                 bbox_head=None,
                 mask_roi_extractor=None,
                 mask_head=None,
                 train_cfg=None,
                 test_cfg=None,
                 init_cfg=None):

    @property
    def with_bbox(self):

    @property
    def with_mask(self):

    @abstractmethod
    def init_weights(self, pretrained):

    @abstractmethod
```

(continues on next page)

(continued from previous page)

```
    def init_bbox_head(self):

    @abstractmethod
    def init_mask_head(self):

    @abstractmethod
    def init_assigner_sampler(self):

    @abstractmethod
    def forward_train(self,
                      x,
                      img_metas,
                      proposal_list,
                      gt_bboxes,
                      gt_labels,
                      gt_bboxes_ignore=None,
                      **kwargs):

    def simple_test(self,
                    x,
                    proposal_list,
                    img_metas,
                    proposals=None,
                    rescale=False,
                    **kwargs):
        """Test without augmentation."""
        pass

    def aug_test(self, x, proposal_list, img_metas, rescale=False, **kwargs):
        """Test with augmentations.
        If rescale is False, then returned bboxes and masks will fit the scale
        of imgs[0].
        """
        pass
```

Double Head's modification is mainly in the bbox_forward logic, and it inherits other logics from the `Base3DRoIHead`. In the `mmdet3d/models/roi_heads/part_aggregation_roi_head.py`, we implement the new RoI Head as the following:

```
from torch.nn import functional as F

from mmdet3d.core import AssignResult
from mmdet3d.core.bbox import bbox3d2result, bbox3d2roi
from mmdet.core import build_assigner, build_sampler
from mmdet.models import HEADS
from ..builder import build_head, build_roi_extractor
from .base_3droi_head import Base3DRoIHead


@HEADS.register_module()
class PartAggregationROIHead(Base3DRoIHead):
```

(continues on next page)

(continued from previous page)

```
    """Part aggregation roi head for PartA2.
    Args:
        semantic_head (ConfigDict): Config of semantic head.
        num_classes (int): The number of classes.
        seg_roi_extractor (ConfigDict): Config of seg_roi_extractor.
        part_roi_extractor (ConfigDict): Config of part_roi_extractor.
        bbox_head (ConfigDict): Config of bbox_head.
        train_cfg (ConfigDict): Training config.
        test_cfg (ConfigDict): Testing config.
    """

    def __init__(self,
                 semantic_head,
                 num_classes=3,
                 seg_roi_extractor=None,
                 part_roi_extractor=None,
                 bbox_head=None,
                 train_cfg=None,
                 test_cfg=None,
                 init_cfg=None):
        super(PartAggregationROIHead, self).__init__(
            bbox_head=bbox_head,
            train_cfg=train_cfg,
            test_cfg=test_cfg,
            init_cfg=init_cfg)
        self.num_classes = num_classes
        assert semantic_head is not None
        self.semantic_head = build_head(semantic_head)

        if seg_roi_extractor is not None:
            self.seg_roi_extractor = build_roi_extractor(seg_roi_extractor)
        if part_roi_extractor is not None:
            self.part_roi_extractor = build_roi_extractor(part_roi_extractor)

        self.init_assigner_sampler()

    def _bbox_forward(self, seg_feats, part_feats, voxels_dict, rois):
        """Forward function of roi_extractor and bbox_head used in both
        training and testing.
        Args:
            seg_feats (torch.Tensor): Point-wise semantic features.
            part_feats (torch.Tensor): Point-wise part prediction features.
            voxels_dict (dict): Contains information of voxels.
            rois (Tensor): Roi boxes.
        Returns:
            dict: Contains predictions of bbox_head and
                features of roi_extractor.
        """
        pooled_seg_feats = self.seg_roi_extractor(seg_feats,
                                                  voxels_dict['voxel_centers'],
                                                  voxels_dict['coors'][..., 0],
                                                  rois)
```

(continues on next page)

(continued from previous page)

```
        pooled_part_feats = self.part_roi_extractor(
            part_feats, voxels_dict['voxel_centers'],
            voxels_dict['coors'][..., 0], rois)
        cls_score, bbox_pred = self.bbox_head(pooled_seg_feats,
                                              pooled_part_feats)

        bbox_results = dict(
            cls_score=cls_score,
            bbox_pred=bbox_pred,
            pooled_seg_feats=pooled_seg_feats,
            pooled_part_feats=pooled_part_feats)
        return bbox_results
```

Here we omit more details related to other functions. Please see the code for more details.

Last, the users need to add the module in `mmdet3d/models/bbox_heads/__init__.py` and `mmdet3d/models/roi_heads/__init__.py` thus the corresponding registry could find and load them.

Alternatively, the users can add

```
custom_imports=dict(
    imports=['mmdet3d.models.roi_heads.part_aggregation_roi_head', 'mmdet3d.models.roi_
↪heads.bbox_heads.parta2_bbox_head'])
```

to the config file and achieve the same goal.

The config file of PartAggregationROIHead is as the following

```
model = dict(
    ...
    roi_head=dict(
        type='PartAggregationROIHead',
        num_classes=3,
        semantic_head=dict(
            type='PointwiseSemanticHead',
            in_channels=16,
            extra_width=0.2,
            seg_score_thr=0.3,
            num_classes=3,
            loss_seg=dict(
                type='FocalLoss',
                use_sigmoid=True,
                reduction='sum',
                gamma=2.0,
                alpha=0.25,
                loss_weight=1.0),
            loss_part=dict(
                type='CrossEntropyLoss', use_sigmoid=True, loss_weight=1.0)),
        seg_roi_extractor=dict(
            type='Single3DRoIAwareExtractor',
            roi_layer=dict(
                type='RoIAwarePool3d',
                out_size=14,
                max_pts_per_voxel=128,
```

(continues on next page)

(continued from previous page)

```
                mode='max')),
        part_roi_extractor=dict(
            type='Single3DRoIAwareExtractor',
            roi_layer=dict(
                type='RoIAwarePool3d',
                out_size=14,
                max_pts_per_voxel=128,
                mode='avg')),
        bbox_head=dict(
            type='PartA2BboxHead',
            num_classes=3,
            seg_in_channels=16,
            part_in_channels=4,
            seg_conv_channels=[64, 64],
            part_conv_channels=[64, 64],
            merge_conv_channels=[128, 128],
            down_conv_channels=[128, 256],
            bbox_coder=dict(type='DeltaXYZWLHRBBoxCoder'),
            shared_fc_channels=[256, 512, 512, 512],
            cls_channels=[256, 256],
            reg_channels=[256, 256],
            dropout_ratio=0.1,
            roi_feat_size=14,
            with_corner_loss=True,
            loss_bbox=dict(
                type='SmoothL1Loss',
                beta=1.0 / 9.0,
                reduction='sum',
                loss_weight=1.0),
            loss_cls=dict(
                type='CrossEntropyLoss',
                use_sigmoid=True,
                reduction='sum',
                loss_weight=1.0)))
    ...
    )
```

Since MMDetection 2.0, the config system supports to inherit configs such that the users can focus on the modification. The second stage of PartA2 Head mainly uses a new `PartAggregationROIHead` and a new `PartA2BboxHead`, the arguments are set according to the `__init__` function of each module.

22.1.5 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 `mmdet3d/models/losses/my_loss.py`. The decorator `weighted_loss` enable the loss to be weighted for each element.

```
import torch
import torch.nn as nn

from ..builder import LOSSES
```

(continues on next page)

(continued from previous page)

```
from .utils import weighted_loss

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

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

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

    def forward(self,
                pred,
                target,
                weight=None,
                avg_factor=None,
                reduction_override=None):
        assert reduction_override in (None, 'none', 'mean', 'sum')
        reduction = (
            reduction_override if reduction_override else self.reduction)
        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 `mmdet3d/models/losses/__init__.py`.

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

Alternatively, you can add

```
custom_imports=dict(
    imports=['mmdet3d.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

TWENTYTHREE

TUTORIAL 5: CUSTOMIZE RUNTIME SETTINGS

23.1 Customize optimization settings

23.1.1 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.

23.1.2 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 `mmdet3d/core/optimizer`. And then implement the new optimizer in a file, e.g., in `mmdet3d/core/optimizer/my_optimizer.py`:

```
from mmcv.runner.optimizer 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.

- Add `mmdet3d/core/optimizer/__init__.py` to import it.

  The newly defined module should be imported in `mmdet3d/core/optimizer/__init__.py` so that the registry will find the new module and add it:

```
from .my_optimizer import MyOptimizer

__all__ = ['MyOptimizer']
```

You also need to import `optimizer` in `mmdet3d/core/__init__.py` by adding:

```
from .optimizer import *
```

Or use `custom_imports` in the config to manually import it

```
custom_imports = dict(imports=['mmdet3d.core.optimizer.my_optimizer'], allow_failed_
↪imports=False)
```

The module `mmdet3d.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. `mmdet3d.core.optimizer.my_optimizer.MyOptimizer` **cannot** be imported directly.

Actually users can use a totally different file directory structure in 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)
```

23.1.3 Customize optimizer constructor

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

```
from mmcv.utils import build_from_cfg

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

(continues on next page)

(continued from previous page)

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

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

    def __call__(self, model):

        return my_optimizer
```

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

23.1.4 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 in the base config. 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,
  )
  ```

23.2 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)
```

23.3 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 `max_epochs` in `runner` 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.

23.4 Customize hooks

23.4.1 Customize self-implemented hooks

1. Implement a new hook

There are some occasions when the users might need to implement a new hook. MMDetection supports customized hooks in training (#3395) since v2.3.0. Thus the users could implement a hook directly in mmdet or their mmdet-based codebases and use the hook by only modifying the config in training. Before v2.3.0, the users need to modify the code

to get the hook registered before training starts. Here we give an example of creating a new hook in mmdet3d and using it in training.

```
from mmcv.runner import HOOKS, Hook


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

    def __init__(self, a, b):
        pass

    def before_run(self, runner):
        pass

    def after_run(self, runner):
        pass

    def before_epoch(self, runner):
        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 hook is in `mmdet3d/core/utils/my_hook.py` there are two ways to do that:

- Modify `mmdet3d/core/utils/__init__.py` to import it.

  The newly defined module should be imported in `mmdet3d/core/utils/__init__.py` so that the registry will find the new module and add it:

```
from .my_hook import MyHook

__all__ = [..., 'MyHook']
```

Or use `custom_imports` in the config to manually import it

```
custom_imports = dict(imports=['mmdet3d.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 setting 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.

23.4.2 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

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

23.4.3 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 reveal 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 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 detailed usages can be found in the docs.

```
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

TWENTYFOUR

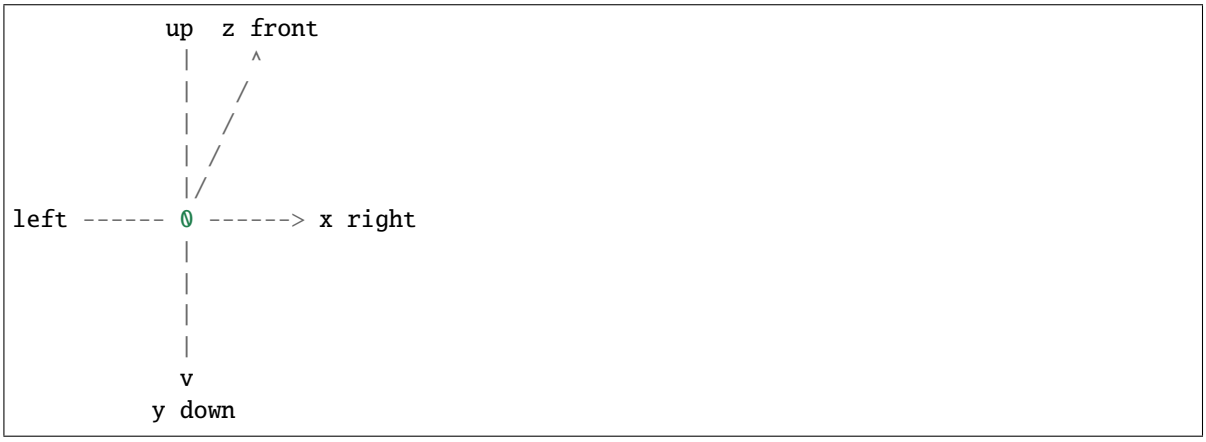
TUTORIAL 6: COORDINATE SYSTEM

24.1 Overview

MMDetection3D uses three different coordinate systems. The existence of different coordinate systems in the society of 3D object detection is necessary, because for various 3D data collection devices, such as LiDAR, depth camera, etc., the coordinate systems are not consistent, and different 3D datasets also follow different data formats. Early works, such as SECOND, VoteNet, convert the raw data to another format, forming conventions that some later works also follow, making the conversion between coordinate systems even more complicated.

Despite the variety of datasets and equipment, by summarizing the line of works on 3D object detection we can roughly categorize coordinate systems into three:

- Camera coordinate system – the coordinate system of most cameras, in which the positive direction of the y-axis points to the ground, the positive direction of the x-axis points to the right, and the positive direction of the z-axis points to the front.



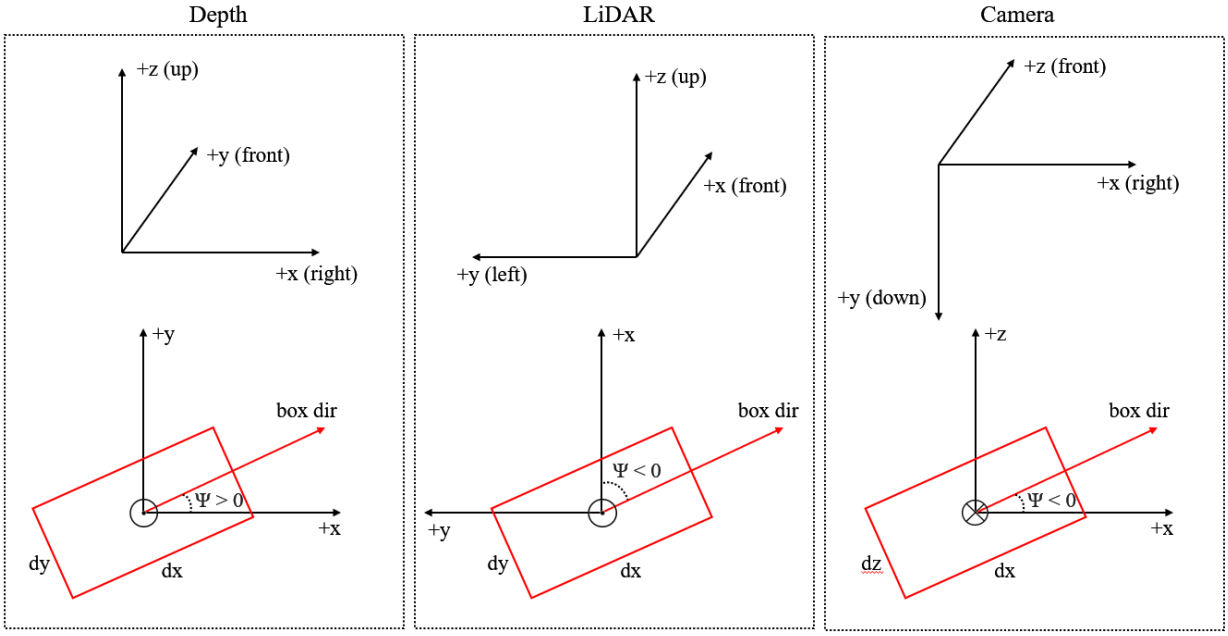
- LiDAR coordinate system – the coordinate system of many LiDARs, in which the negative direction of the z-axis points to the ground, the positive direction of the x-axis points to the front, and the positive direction of the y-axis points to the left.

- Depth coordinate system – the coordinate system used by VoteNet, H3DNet, etc., in which the negative direction of the z-axis points to the ground, the positive direction of the x-axis points to the right, and the positive direction of the y-axis points to the front.

```
           z up   y front
             ^     ^
             |    /
             |   /
             |  /
             | /
             |/
left ------ 0 ------> x right
```

The definition of coordinate systems in this tutorial is actually **more than just defining the three axes**. For a box in the form of (x, y, z, dx, dy, dz, r), our coordinate systems also define how to interpret the box dimensions (dx, dy, dz) and the yaw angle r.

The illustration of the three coordinate systems is shown below:



The three figures above are the 3D coordinate systems while the three figures below are the bird's eye view.

We will stick to the three coordinate systems defined in this tutorial in the future.

24.2 Definition of the yaw angle

Please refer to wikipedia for the standard definition of the yaw angle. In object detection, we choose an axis as the gravity axis, and a reference direction on the plane Π perpendicular to the gravity axis, then the reference direction has a yaw angle of 0, and other directions on Π have non-zero yaw angles depending on its angle with the reference direction.

Currently, for all supported datasets, annotations do not include pitch angle and roll angle, which means we need only consider the yaw angle when predicting boxes and calculating overlap between boxes.

In MMDetection3D, all three coordinate systems are right-handed coordinate systems, which means the ascending direction of the yaw angle is counter-clockwise if viewed from the negative direction of the gravity axis (the axis is

pointing at one's eyes).

The figure below shows that, in this right-handed coordinate system, if we set the positive direction of the x-axis as a reference direction, then the positive direction of the y-axis has a yaw angle of $\frac{\pi}{2}$.

```
                    z up   y front (yaw=0.5*pi)
                     ^     ^
                     |    /
                     |   /
                     |  /
                     | /
                     |/
left (yaw=pi)  ------ 0 ------> x right (yaw=0)
```

For a box, the value of its yaw angle equals its direction minus a reference direction. In all three coordinate systems in MMDetection3D, the reference direction is always the positive direction of the x-axis, while the direction of a box is defined to be parallel with the x-axis if its yaw angle is 0. The definition of the yaw angle of a box is illustrated in the figure below.

```
y front
  ^       box direction (yaw=0.5*pi)
 /|\         ^
  |         /|\
  |      ____|____
  |     |    |    |
  |     |    |    |
__|_____|____|____|______\ x right
  |     |    |    |      /
  |     |    |    |
  |     |____|____|
  |
```

24.3 Definition of the box dimensions

The definition of the box dimensions cannot be disentangled with the definition of the yaw angle. In the previous section, we said that the direction of a box is defined to be parallel with the x-axis if its yaw angle is 0. Then naturally, the dimension of a box which corresponds to the x-axis should be dx. However, this is not always the case in some datasets (we will address that later).

The following figures show the meaning of the correspondence between the x-axis and dx, and between the y-axis and dy.

```
y front
  ^       box direction (yaw=0.5*pi)
 /|\         ^
  |         /|\
  |      ____|____
  |     |    |    |
  |     |    |    | dx
__|_____|____|____|______\ x right
  |     |    |    |      /
  |     |    |    |
  |     |____|____|
  |          dy
```

Note that the box direction is always parallel with the edge dx.

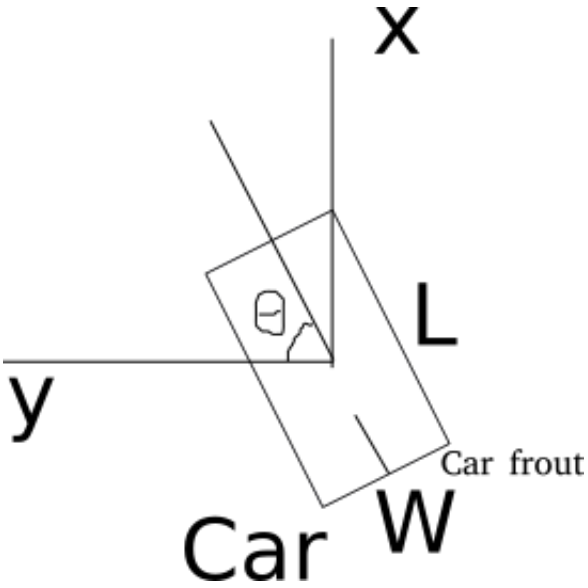
```
y front
  ^      _________
 /|\    |     |    |
  |     |     |    |
  |     |     |    | dy
  |     |____|____|____\  box direction (yaw=0)
  |     |     |    |    /
__|____|____|____|_________\ x right
  |     |     |    |         /
  |     |____|____|
  |          dx
  |
```

24.4 Relation with raw coordinate systems of supported datasets

24.4.1 KITTI

The raw annotation of KITTI is under camera coordinate system, see get_label_anno. In MMDetection3D, to train LiDAR-based models on KITTI, the data is first converted from camera coordinate system to LiDAR coordinate system, see get_ann_info. For training vision-based models, the data is kept in the camera coordinate system.

In SECOND, the LiDAR coordinate system for a box is defined as follows (a bird's eye view):



For each box, the dimensions are (w, l, h), and the reference direction for the yaw angle is the positive direction of the y axis. For more details, refer to the repo.

Our LiDAR coordinate system has two changes:

- The yaw angle is defined to be right-handed instead of left-handed for consistency;
- The box dimensions are (l, w, h) instead of (w, l, h), since w corresponds to dy and l corresponds to dx in KITTI.

24.4.2 Waymo

We use the KITTI-format data of Waymo dataset. Therefore, KITTI and Waymo also share the same coordinate system in our implementation.

24.4.3 NuScenes

NuScenes provides a toolkit for evaluation, in which each box is wrapped into a `Box` instance. The coordinate system of `Box` is different from our LiDAR coordinate system in that the first two elements of the box dimension correspond to (dy, dx), or (w, l), respectively, instead of the reverse. For more details, please refer to the NuScenes tutorial.

Readers may refer to the NuScenes development kit for the definition of a NuScenes box and implementation of NuScenes evaluation.

24.4.4 Lyft

Lyft shares the same data format with NuScenes as far as coordinate system is involved.

Please refer to the official website for more information.

24.4.5 ScanNet

The raw data of ScanNet is not point cloud but mesh. The sampled point cloud data is under our depth coordinate system. For ScanNet detection task, the box annotations are axis-aligned, and the yaw angle is always zero. Therefore the direction of the yaw angle in our depth coordinate system makes no difference regarding ScanNet.

24.4.6 SUN RGB-D

The raw data of SUN RGB-D is not point cloud but RGB-D image. By back projection, we obtain the corresponding point cloud for each image, which is under our Depth coordinate system. However, the annotation is not under our system and thus needs conversion.

For the conversion from raw annotation to annotation under our Depth coordinate system, please refer to sunrgbd_data_utils.py.

24.4.7 S3DIS

S3DIS shares the same coordinate system as ScanNet in our implementation. However, S3DIS is a segmentation-task-only dataset, and thus no annotation is coordinate system sensitive.

24.5 Examples

24.5.1 Box conversion (between different coordinate systems)

Take the conversion between our Camera coordinate system and LiDAR coordinate system as an example:

First, for points and box centers, the coordinates before and after the conversion satisfy the following relationship:

- $x_{LiDAR} = z_{camera}$
- $y_{LiDAR} = -x_{camera}$

- $z_{LiDAR} = -y_{camera}$

Then, the box dimensions before and after the conversion satisfy the following relationship:

- $dx_{LiDAR} = dx_{camera}$
- $dy_{LiDAR} = dz_{camera}$
- $dz_{LiDAR} = dy_{camera}$

Finally, the yaw angle should also be converted:

- $r_{LiDAR} = -\frac{\pi}{2} - r_{camera}$

See the code here for more details.

24.5.2 Bird's Eye View

The BEV of a camera coordinate system box is $(x, z, dx, dz, -r)$ if the 3D box is (x, y, z, dx, dy, dz, r). The inversion of the sign of the yaw angle is because the positive direction of the gravity axis of the Camera coordinate system points to the ground.

See the code here for more details.

24.5.3 Rotation of boxes

We set the rotation of all kinds of boxes to be counter-clockwise about the gravity axis. Therefore, to rotate a 3D box we first calculate the new box center, and then we add the rotation angle to the yaw angle.

See the code here for more details.

24.6 Common FAQ

24.6.1 Q1: Are the box related ops universal to all coordinate system types?

No. For example, RoI-Aware Pooling ops is applicable to boxes under Depth or LiDAR coordinate system only. The evaluation functions for KITTI dataset here are only applicable to boxes under Camera coordinate system since the rotation is clockwise if viewed from above.

For each box related op, we have marked the type of boxes to which we can apply the op.

24.6.2 Q2: In every coordinate system, do the three axes point exactly to the right, the front, and the ground, respectively?

No. For example, in KITTI, we need a calibration matrix when converting from Camera coordinate system to LiDAR coordinate system.

24.6.3 Q3: How does a phase difference of 2π in the yaw angle of a box affect evaluation?

For IoU calculation, a phase difference of 2π in the yaw angle will result in the same box, thus not affecting evaluation.

For angle prediction evaluation such as the NDS metric in NuScenes and the AOS metric in KITTI, the angle of predicted boxes will be first standardized, so the phase difference of 2π will not change the result.

24.6.4 Q4: How does a phase difference of π in the yaw angle of a box affect evaluation?

For IoU calculation, a phase difference of π in the yaw angle will result in the same box, thus not affecting evaluation.

However, for angle prediction evaluation, this will result in the exact opposite direction.

Just think about a car. The yaw angle is the angle between the direction of the car front and the positive direction of the x-axis. If we add π to this angle, the car front will become the car rear.

For categories such as barrier, the front and the rear have no difference, therefore a phase difference of π will not affect the angle prediction score.

CHAPTER

TWENTYFIVE

TUTORIAL 7: BACKENDS SUPPORT

We support different file client backends: Disk, Ceph and LMDB, etc. Here is an example of how to modify configs for Ceph-based data loading and saving.

25.1 Load data and annotations from Ceph

We support loading data and generated annotation info files (pkl and json) from Ceph:

```
# set file client backends as Ceph
file_client_args = dict(
    backend='petrel',
    path_mapping=dict({
        './data/nuscenes/':
        's3://openmmlab/datasets/detection3d/nuscenes/', # replace the path with your␣
↪data path on Ceph
        'data/nuscenes/':
        's3://openmmlab/datasets/detection3d/nuscenes/' # replace the path with your␣
↪data path on Ceph
    }))

db_sampler = dict(
    data_root=data_root,
    info_path=data_root + 'kitti_dbinfos_train.pkl',
    rate=1.0,
    prepare=dict(filter_by_difficulty=[-1], filter_by_min_points=dict(Car=5)),
    sample_groups=dict(Car=15),
    classes=class_names,
    # set file client for points loader to load training data
    points_loader=dict(
        type='LoadPointsFromFile',
        coord_type='LIDAR',
        load_dim=4,
        use_dim=4,
        file_client_args=file_client_args),
    # set file client for data base sampler to load db info file
    file_client_args=file_client_args)

train_pipeline = [
    # set file client for loading training data
    dict(type='LoadPointsFromFile', coord_type='LIDAR', load_dim=4, use_dim=4, file_
↪client_args=file_client_args),
```

(continues on next page)

(continued from previous page)

```
    # set file client for loading training data annotations
    dict(type='LoadAnnotations3D', with_bbox_3d=True, with_label_3d=True, file_client_
↪args=file_client_args),
    dict(type='ObjectSample', db_sampler=db_sampler),
    dict(
        type='ObjectNoise',
        num_try=100,
        translation_std=[0.25, 0.25, 0.25],
        global_rot_range=[0.0, 0.0],
        rot_range=[-0.15707963267, 0.15707963267]),
    dict(type='RandomFlip3D', flip_ratio_bev_horizontal=0.5),
    dict(
        type='GlobalRotScaleTrans',
        rot_range=[-0.78539816, 0.78539816],
        scale_ratio_range=[0.95, 1.05]),
    dict(type='PointsRangeFilter', point_cloud_range=point_cloud_range),
    dict(type='ObjectRangeFilter', point_cloud_range=point_cloud_range),
    dict(type='PointShuffle'),
    dict(type='DefaultFormatBundle3D', class_names=class_names),
    dict(type='Collect3D', keys=['points', 'gt_bboxes_3d', 'gt_labels_3d'])
]
test_pipeline = [
    # set file client for loading validation/testing data
    dict(type='LoadPointsFromFile', coord_type='LIDAR', load_dim=4, use_dim=4, file_
↪client_args=file_client_args),
    dict(
        type='MultiScaleFlipAug3D',
        img_scale=(1333, 800),
        pts_scale_ratio=1,
        flip=False,
        transforms=[
            dict(
                type='GlobalRotScaleTrans',
                rot_range=[0, 0],
                scale_ratio_range=[1., 1.],
                translation_std=[0, 0, 0]),
            dict(type='RandomFlip3D'),
            dict(
                type='PointsRangeFilter', point_cloud_range=point_cloud_range),
            dict(
                type='DefaultFormatBundle3D',
                class_names=class_names,
                with_label=False),
            dict(type='Collect3D', keys=['points'])
        ])
]

data = dict(
    # set file client for loading training info files (.pkl)
    train=dict(
        type='RepeatDataset',
        times=2,
```

(continues on next page)

(continued from previous page)

```
        dataset=dict(pipeline=train_pipeline, classes=class_names, file_client_args=file_
↪client_args)),
    # set file client for loading validation info files (.pkl)
    val=dict(pipeline=test_pipeline, classes=class_names,file_client_args=file_client_
↪args),
    # set file client for loading testing info files (.pkl)
    test=dict(pipeline=test_pipeline, classes=class_names, file_client_args=file_client_
↪args))
```

25.2 Load pretrained model from Ceph

```
model = dict(
    pts_backbone=dict(
        _delete_=True,
        type='NoStemRegNet',
        arch='regnetx_1.6gf',
        init_cfg=dict(
            type='Pretrained', checkpoint='s3://openmmlab/checkpoints/mmdetection3d/
↪regnetx_1.6gf'), # replace the path with your pretrained model path on Ceph
        ...
```

25.3 Load checkpoint from Ceph

```
# replace the path with your checkpoint path on Ceph
load_from = 's3://openmmlab/checkpoints/mmdetection3d/v0.1.0_models/pointpillars/hv_
↪pointpillars_secfpn_6x8_160e_kitti-3d-car/hv_pointpillars_secfpn_6x8_160e_kitti-3d-car_
↪20200620_230614-77663cd6.pth'
resume_from = None
workflow = [('train', 1)]
```

25.4 Save checkpoint into Ceph

```
# checkpoint saving
# replace the path with your checkpoint saving path on Ceph
checkpoint_config = dict(interval=1, max_keep_ckpts=2, out_dir='s3://openmmlab/
↪mmdetection3d')
```

25.5 EvalHook saves the best checkpoint into Ceph

```
# replace the path with your checkpoint saving path on Ceph
evaluation = dict(interval=1, save_best='bbox', out_dir='s3://openmmlab/mmdetection3d')
```

25.6 Save the training log into Ceph

The training log will be backed up to the specified Ceph path after training.

```
log_config = dict(
    interval=50,
    hooks=[
        dict(type='TextLoggerHook', out_dir='s3://openmmlab/mmdetection3d'),
    ])
```

You can also delete the local training log after backing up to the specified Ceph path by setting `keep_local = False`.

```
log_config = dict(
    interval=50,
    hooks=[
        dict(type='TextLoggerHook', out_dir='s3://openmmlab/mmdetection3d', keep_
↪local=False),
    ])
```

CHAPTER

TWENTYSIX

TUTORIAL 8: MMDETECTION3D MODEL DEPLOYMENT

To meet the speed requirement of the model in practical use, usually, we deploy the trained model to inference backends. MMDeploy is OpenMMLab model deployment framework. Now MMDeploy has supported MMDetection3D model deployment, and you can deploy the trained model to inference backends by MMDeploy.

26.1 Prerequisite

26.1.1 Install MMDeploy

```
git clone -b master git@github.com:open-mmlab/mmdeploy.git
cd mmdeploy
git submodule update --init --recursive
```

26.1.2 Install backend and build custom ops

According to MMDeploy documentation, choose to install the inference backend and build custom ops. Now supported inference backends for MMDetection3D include OnnxRuntime, TensorRT, OpenVINO.

26.2 Export model

Export the Pytorch model of MMDetection3D to the ONNX model file and the model file required by the backend. You could refer to MMDeploy docs how to convert model.

```
python ./tools/deploy.py \
    ${DEPLOY_CFG_PATH} \
    ${MODEL_CFG_PATH} \
    ${MODEL_CHECKPOINT_PATH} \
    ${INPUT_IMG} \
    --test-img ${TEST_IMG} \
    --work-dir ${WORK_DIR} \
    --calib-dataset-cfg ${CALIB_DATA_CFG} \
    --device ${DEVICE} \
    --log-level INFO \
    --show \
    --dump-info
```

26.2.1 Description of all arguments

- `deploy_cfg` : The path of deploy config file in MMDeploy codebase.
- `model_cfg` : The path of model config file in OpenMMLab codebase.
- `checkpoint` : The path of model checkpoint file.
- `img` : The path of point cloud file or image file that used to convert model.
- `--test-img` : The path of image file that used to test model. If not specified, it will be set to `None`.
- `--work-dir` : The path of work directory that used to save logs and models.
- `--calib-dataset-cfg` : Only valid in int8 mode. Config used for calibration. If not specified, it will be set to `None` and use "val" dataset in model config for calibration.
- `--device` : The device used for conversion. If not specified, it will be set to `cpu`.
- `--log-level` : To set log level which in `'CRITICAL'`, `'FATAL'`, `'ERROR'`, `'WARN'`, `'WARNING'`, `'INFO'`, `'DEBUG'`, `'NOTSET'`. If not specified, it will be set to `INFO`.
- `--show` : Whether to show detection outputs.
- `--dump-info` : Whether to output information for SDK.

26.2.2 Example

```
cd mmdeploy
python tools/deploy.py \
    configs/mmdet3d/voxel-detection/voxel-detection_tensorrt_dynamic-kitti.py \
    ${$MMDET3D_DIR}/configs/pointpillars/hv_pointpillars_secfpn_6x8_160e_kitti-3d-3class.
↪py \
    ${$MMDET3D_DIR}/checkpoints/hv_pointpillars_secfpn_6x8_160e_kitti-3d-3class_20200620_
↪230421-aa0f3adb.pth \
    ${$MMDET3D_DIR}/demo/data/kitti/kitti_000008.bin \
    --work-dir work-dir \
    --device cuda:0 \
    --show
```

26.3 Inference Model

Now you can do model inference with the APIs provided by the backend. But what if you want to test the model instantly? We have some backend wrappers for you.

```
from mmdeploy.apis import inference_model

result = inference_model(model_cfg, deploy_cfg, backend_files, img=img, device=device)
```

The `inference_model` will create a wrapper module and do the inference for you. The result has the same format as the original OpenMMLab repo.

26.4 Evaluate model (Optional)

You can test the accuracy and speed of the model in the inference backend. You could refer to MMDeploy docs how to measure performance of models.

```
python tools/test.py \
    ${DEPLOY_CFG} \
    ${MODEL_CFG} \
    --model ${BACKEND_MODEL_FILES} \
    [--out ${OUTPUT_PKL_FILE}] \
    [--format-only] \
    [--metrics ${METRICS}] \
    [--show] \
    [--show-dir ${OUTPUT_IMAGE_DIR}] \
    [--show-score-thr ${SHOW_SCORE_THR}] \
    --device ${DEVICE} \
    [--cfg-options ${CFG_OPTIONS}] \
    [--metric-options ${METRIC_OPTIONS}] \
    [--log2file work_dirs/output.txt]
```

26.4.1 Example

```
cd mmdeploy
python tools/test.py \
    configs/mmdet3d/voxel-detection/voxel-detection_onnxruntime_dynamic.py \
    ${MMDET3D_DIR}/configs/centerpoint/centerpoint_02pillar_second_secfpn_circlenms_4x8_
↪cyclic_20e_nus.py \
    --model work-dir/end2end.onnx \
    --metrics bbox \
    --device cpu
```

26.5 Supported models

26.6 Note

- MMDeploy version >= 0.4.0.
- Currently, CenterPoint has only supported the pillar version.

CHAPTER

TWENTYSEVEN

TUTORIAL 9: USE PURE POINT CLOUD DATASET

27.1 Data Pre-Processing

27.1.1 Convert Point cloud format

Currently, we only support bin format point cloud training and inference, before training on your own datasets, you need to transform your point cloud format to bin file. The common point cloud data formats include pcd and las, we provide some open-source tools for reference.

1. Convert pcd to bin: https://github.com/leofansq/Tools_RosBag2KITTI
2. Convert las to bin: The common conversion path is las -> pcd -> bin, and the conversion from las -> pcd can be achieved through this tool.

27.1.2 Point cloud annotation

MMDetection3D does not support point cloud annotation. Some open-source annotation tools are offered for reference:

- SUSTechPOINTS
- LATTE

Besides, we improved LATTE for better usage. More details can be found here.

27.2 Support new data format

To support a new data format, you can either convert them to existing formats or directly convert them to the middle format. You could also choose to convert them offline (before training by a script) or online (implement a new dataset and do the conversion at training).

27.2.1 Reorganize new data formats to existing format

Once your datasets only contain point cloud file and 3D Bounding box annotations, without calib file. We recommend converting it into the basic formats, the annotations files in basic format has the following necessary keys:

```
[
    {'sample_idx':
     'lidar_points': {'lidar_path': velodyne_path,
```

(continues on next page)

(continued from previous page)

```
                    ....
                   },
    'annos': {'box_type_3d':  (str)  'LiDAR/Camera/Depth'
              'gt_bboxes_3d':  <np.ndarray> (n, 7)
              'gt_names':  [list]
              ....
            }
    'calib': { .....}
    'images': { .....}
   }
]
```

In MMDetection3D, for the data that is inconvenient to read directly online, we recommend converting it into into basic format as above and do the conversion offline, thus you only need to modify the config's data annotation paths and classes after the conversion. To use data that share a similar format as the existing datasets, e.g., Lyft has a similar format as the nuScenes dataset, we recommend directly implementing a new data converter and a dataset class to convert the data and load the data, respectively. In this procedure, the code can inherit from the existing dataset classes to reuse the code.

27.2.2 Reorganize new data format to middle format

There is also a way if users do not want to convert the annotation format to existing formats. Actually, we convert all the supported datasets into pickle files, which summarize useful information for model training and inference.

The annotation of a dataset is a list of dict, each dict corresponds to a frame. A basic example (used in KITTI) is as follows. A frame consists of several keys, like `image`, `point_cloud`, `calib` and `annos`. As long as we could directly read data according to these information, the organization of raw data could also be different from existing ones. With this design, we provide an alternative choice for customizing datasets.

```
[
    {'image': {'image_idx': 0, 'image_path': 'training/image_2/000000.png', 'image_shape
↪': array([ 370, 1224], dtype=int32)},
    'point_cloud': {'num_features': 4, 'velodyne_path': 'training/velodyne/000000.bin'},
    'calib': {'P0': array([[707.0493,   0.    , 604.0814,   0.    ],
      [  0.    , 707.0493, 180.5066,   0.    ],
      [  0.    ,   0.    ,   1.    ,   0.    ],
      [  0.    ,   0.    ,   0.    ,   1.    ]]),
      'P1': array([[ 707.0493,    0.    ,  604.0814, -379.7842],
      [   0.    ,  707.0493,  180.5066,    0.    ],
      [   0.    ,    0.    ,    1.    ,    0.    ],
      [   0.    ,    0.    ,    0.    ,    1.    ]]),
      'P2': array([[ 7.070493e+02,  0.000000e+00,  6.040814e+02,  4.575831e+01],
      [ 0.000000e+00,  7.070493e+02,  1.805066e+02, -3.454157e-01],
      [ 0.000000e+00,  0.000000e+00,  1.000000e+00,  4.981016e-03],
      [ 0.000000e+00,  0.000000e+00,  0.000000e+00,  1.000000e+00]]),
      'P3': array([[ 7.070493e+02,  0.000000e+00,  6.040814e+02, -3.341081e+02],
      [ 0.000000e+00,  7.070493e+02,  1.805066e+02,  2.330660e+00],
      [ 0.000000e+00,  0.000000e+00,  1.000000e+00,  3.201153e-03],
      [ 0.000000e+00,  0.000000e+00,  0.000000e+00,  1.000000e+00]]),
      'R0_rect': array([[ 0.9999128 ,  0.01009263, -0.00851193,  0.        ],
```

(continues on next page)

(continued from previous page)

```
       [-0.01012729,  0.9999406 , -0.00403767,  0.        ],
       [ 0.00847068,  0.00412352,  0.9999556 ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  1.        ]]),
       'Tr_velo_to_cam': array([[ 0.00692796, -0.9999722 , -0.00275783, -0.02457729],
       [-0.00116298,  0.00274984, -0.9999955 , -0.06127237],
       [ 0.9999753 ,  0.00693114, -0.0011439 , -0.3321029 ],
       [ 0.        ,  0.        ,  0.        ,  1.        ]]),
       'Tr_imu_to_velo': array([[ 9.999976e-01,  7.553071e-04, -2.035826e-03, -8.086759e-
↪01],
       [-7.854027e-04,  9.998898e-01, -1.482298e-02,  3.195559e-01],
       [ 2.024406e-03,  1.482454e-02,  9.998881e-01, -7.997231e-01],
       [ 0.000000e+00,  0.000000e+00,  0.000000e+00,  1.000000e+00]])},
     'annos': {'name': array(['Pedestrian'], dtype='<U10'), 'truncated': array([0.]),
↪'occluded': array([0]), 'alpha': array([-0.2]), 'bbox': array([[712.4 , 143.  , 810.73,
↪ 307.92]]), 'dimensions': array([[1.2 , 1.89, 0.48]]), 'location': array([[1.84, 1.47,␣
↪8.41]]), 'rotation_y': array([0.01]), 'score': array([0.]), 'index': array([0],␣
↪dtype=int32), 'group_ids': array([0], dtype=int32), 'difficulty': array([0],␣
↪dtype=int32), 'num_points_in_gt': array([377], dtype=int32)}}
    ...
]
```

On top of this you can write a new Dataset class inherited from `Custom3DDataset`, and overwrite related methods, like KittiDataset and ScanNetDataset.

27.2.3 An example of customized dataset

Here we provide an example of customized dataset.

Assume the annotation has been reorganized into a list of dict in pickle files like basic format. The bounding boxes annotations are stored in `annotation.pkl` as the following

```
{'sample_idx': 120,
 'lidar_points': {'lidar_path': 'training/000004.bin'},
 'annos': {'bbox_type_3d': 'LiDAR',
           'gt_bboxes_3d': array([[1.48129511,  3.52074146,  1.85652947, 1.74445975, 0.
↪23195696, 0.57235193, -0.25525],
           [ 2.90395617, -3.48033905,  1.52682471,[0.66077662, 0.17072392, 0.67153597, 2.
↪23145]]),
           'gt_names': ['car', 'pedestrian']
         }
}
```

If the pkl only contains the necessary keys, you can directly use the `Custom3DDataset` for training:

Then in the config, to use `Custom3DDataset` you can modify the config as the following

```
dataset_A_train = dict(
    type='Custom3DDataset',
    ann_file = 'annotation.pkl',
    pipeline=train_pipeline
)
```

otherwise you need to create a new dataset in `mmdet3d/datasets/my_dataset.py` to load the data and rewrite the `get_ann_info` method.

```
import numpy as np
from os import path as osp

from mmdet3d.core import show_result
from mmdet3d.core.bbox import DepthInstance3DBoxes
from mmdet.datasets import DATASETS
from .custom_3d import Custom3DDataset


@DATASETS.register_module()
class MyDataset(Custom3DDataset):
    CLASSES = ('cabinet', 'bed', 'chair', 'sofa', 'table', 'door', 'window',
               'bookshelf', 'picture', 'counter', 'desk', 'curtain',
               'refrigerator', 'showercurtrain', 'toilet', 'sink', 'bathtub',
               'garbagebin')

    def __init__(self,
                 data_root,
                 ann_file,
                 pipeline=None,
                 classes=None,
                 modality=None,
                 box_type_3d='Depth',
                 filter_empty_gt=True,
                 test_mode=False):
        super().__init__(
            data_root=data_root,
            ann_file=ann_file,
            pipeline=pipeline,
            classes=classes,
            modality=modality,
            box_type_3d=box_type_3d,
            filter_empty_gt=filter_empty_gt,
            test_mode=test_mode)

    def get_ann_info(self, index):
        # Use index to get the annos, thus the evalhook could also use this api
        info = self.data_infos[index]
        if info['annos']['gt_num'] != 0:
            gt_bboxes_3d = info['annos']['gt_boxes_upright_depth'].astype(
                np.float32)  # k, 6
            gt_labels_3d = info['annos']['class'].astype(np.int64)
        else:
            gt_bboxes_3d = np.zeros((0, 6), dtype=np.float32)
            gt_labels_3d = np.zeros((0, ), dtype=np.int64)

        # to target box structure
        gt_bboxes_3d = DepthInstance3DBoxes(
            gt_bboxes_3d,
            box_dim=gt_bboxes_3d.shape[-1],
```

(continues on next page)

(continued from previous page)

```
            with_yaw=False,
            origin=(0.5, 0.5, 0.5)).convert_to(self.box_mode_3d)

        pts_instance_mask_path = osp.join(self.data_root,
                                          info['pts_instance_mask_path'])
        pts_semantic_mask_path = osp.join(self.data_root,
                                          info['pts_semantic_mask_path'])

        anns_results = dict(
            gt_bboxes_3d=gt_bboxes_3d,
            gt_labels_3d=gt_labels_3d,
            pts_instance_mask_path=pts_instance_mask_path,
            pts_semantic_mask_path=pts_semantic_mask_path)
        return anns_results
```

Then in the config, to use `MyDataset` you can modify the config as the following

```
dataset_A_train = dict(
    type='MyDataset',
    ann_file = 'annotation.pkl',
    pipeline=train_pipeline
)
```

27.3 Customize datasets by dataset wrappers

MMDetection3D also supports many dataset wrappers to mix the dataset or modify the dataset distribution for training like MMDetection. 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.

27.3.1 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
        )
    )
```

27.3.2 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
        )
    )
```

You may refer to source code for details.

27.3.3 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,
   ```

(continues on next page)

(continued from previous page)

```
    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,
```

(continues on next page)

(continued from previous page)

```
    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
)
```

27.4 Modify Dataset Classes

With existing dataset types, we can modify the class names of them to train subset of the annotations. For example, if you want to train only three classes of the current dataset, you can modify the classes of dataset. The dataset will filter out the ground truth boxes of other classes automatically.

```
classes = ('person', 'bicycle', 'car')
data = dict(
    train=dict(classes=classes),
    val=dict(classes=classes),
    test=dict(classes=classes))
```

MMDetection V2.0 also supports to read the classes from a file, which is common in real applications. For example, assume the `classes.txt` contains the name of classes as the following.

```
person
bicycle
car
```

Users can set the classes as a file path, the dataset will load it and convert it to a list automatically.

```
classes = 'path/to/classes.txt'
data = dict(
    train=dict(classes=classes),
    val=dict(classes=classes),
    test=dict(classes=classes))
```

27.5 Loading Point Clouds Adjustment

Generally speaking, the most basic bin data contains (x, y, z) information, and some also include intensity, elongation (point cloud elongation), timestamp, and the point cloud dimension ranges from 3 to 6. In MMDetection3D, you need to adjust the some settings in config while customized dataset training:

```
dict(
    type='LoadPointsFromFile',
    coord_type='LIDAR',
    # adjust accordingly according to the dimension
    # of the point cloud of your own dataset
    load_dim=3,
    # actually used dimensionyou can also specify the
    # specific dimension in list format
    use_dim=3),
```

27.6 Training Setting Adjustment

In order to avoid some problems in the training process and improve the performance of the model on the custom dataset, some training settings need to be adjusted according to the dataset.

27.6.1 Adjust Point Cloud Range and Annotations in Config

For example, we can adjust `point_cloud_range` in config file to change training point cloud range. In KITTI dataset, the `point_cloud_range` is set to be `[0, -39.68, -3, 69.12, 39.68, 1]`. By setting point cloud range, the `PointsRangeFilter` is used to filter point cloud and its mask (semantic and instance), and `ObjectRangeFilter` is used to filter 3D bounding boxes.

```
dict(type='PointsRangeFilter', point_cloud_range=point_cloud_range),
dict(type='ObjectRangeFilter', point_cloud_range=point_cloud_range),
```

27.6.2 Adjust Voxel Size in Config

Here you can refer to the setting of the existing datasets. theoretically, `voxel_size` is linked to the setting of `point_cloud_range`. Setting a smaller `voxel_size` will increase the voxel num and the corresponding memory consumption. In addition, the following issues need to be noted:

if the `point_cloud_range` and `voxel_size` are set to be `[0, -40, -3, 70.4, 40, 1]` and `[0.05, 0.05, 0.1]` respectively, then the shape of intermediate feature map should be `[(1-(-3))/0.1+1, (40-(-40))/0.05, (70.4-0)/0.05]=[41, 1600, 1408]`. More details refers to this issue.

27.6.3 Adjust Anchor Range and Size in Config

```
anchor_generator=dict(
    type='Anchor3DRangeGenerator',
    ranges=[
        [0, -40.0, -0.6, 70.4, 40.0, -0.6],
        [0, -40.0, -0.6, 70.4, 40.0, -0.6],
        [0, -40.0, -1.78, 70.4, 40.0, -1.78],
    ],
    sizes=[[0.8, 0.6, 1.73], [1.76, 0.6, 1.73], [3.9, 1.6, 1.56]],
    rotations=[0, 1.57],
    reshape_out=False),
```

Regarding the setting of `anchor_range`, it is generally adjusted according to dataset. Note that `z` value needs to be adjusted accordingly to the position of the point cloud, please refer to this issue.

Regarding the setting of `anchor_size`, it is usually necessary to count the average length, width and height of the entire training dataset as `anchor_size` to obtain the best results.

Note (related to MMDetection):

- Before MMDetection v2.5.0, the dataset will filter out the empty GT images automatically if the classes are set and there is no way to disable that through config. This is an undesirable behavior and introduces confusion because if the classes are not set, the dataset only filters the empty GT images when `filter_empty_gt=True` and `test_mode=False`. After MMDetection v2.5.0, we decouple the image filtering process and the classes modification, i.e., the dataset will only filter empty GT images when `filter_empty_gt=True` and `test_mode=False`, no matter whether the classes are set. Thus, setting the classes only influences the annotations of classes used for training and users could decide whether to filter empty GT images by themselves.
- Since the middle format only has box labels and does not contain the class names, when using `CustomDataset`, users cannot filter out the empty GT images through configs but only do this offline.
- The features for setting dataset classes and dataset filtering will be refactored to be more user-friendly in the future (depends on the progress).

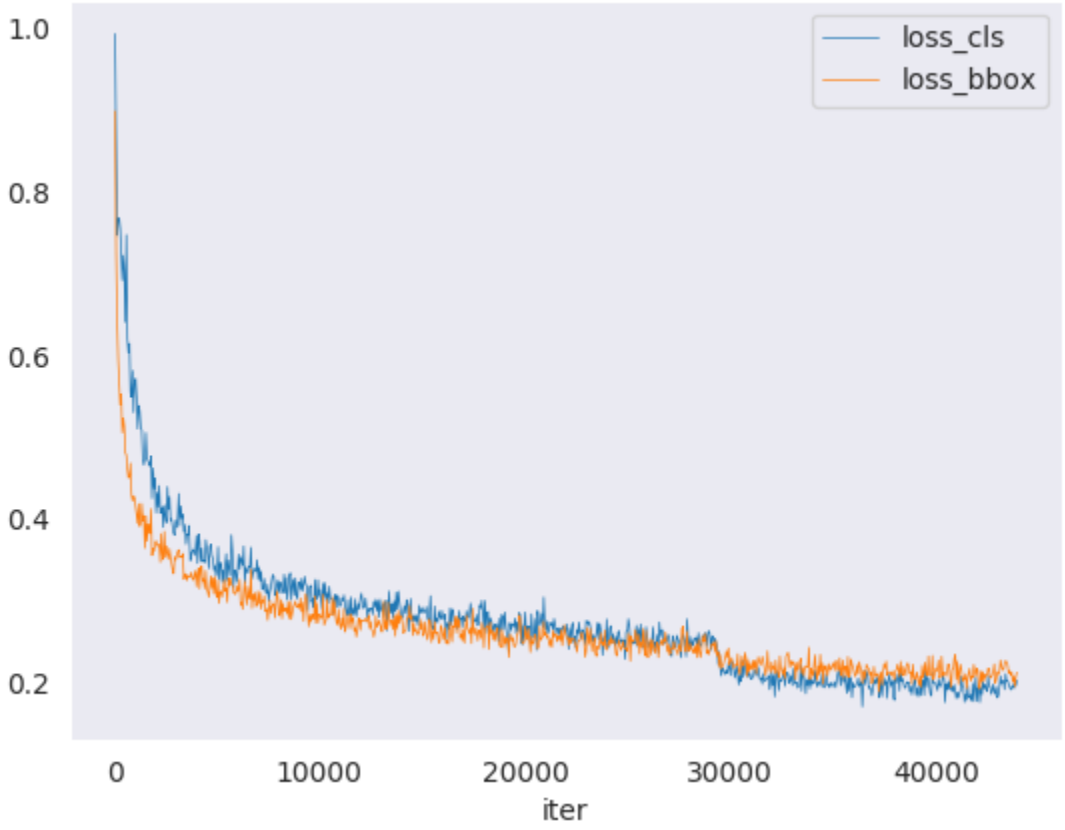
We provide lots of useful tools under `tools/` directory.

CHAPTER

TWENTYEIGHT

LOG ANALYSIS

You can plot loss/mAP curves given a training log file. Run `pip install seaborn` first to install the dependency.




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

Notice: If the metric you want to plot is calculated in the eval stage, you need to add the flag `--mode eval`. If you perform evaluation with an interval of `${INTERVAL}`, you need to add the args `--interval ${INTERVAL}`.

Examples:

- Plot the classification loss of some run.

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

- Plot the classification and regression loss of some run, and save the figure to a pdf.

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

- Compare the bbox mAP of two runs in the same figure.

```
# evaluate PartA2 and second on KITTI according to Car_3D_moderate_strict
python tools/analysis_tools/analyze_logs.py plot_curve tools/logs/PartA2.log.json␣
↪tools/logs/second.log.json --keys KITTI/Car_3D_moderate_strict --legend PartA2␣
↪second --mode eval --interval 1
# evaluate PointPillars for car and 3 classes on KITTI according to Car_3D_moderate_
↪strict
python tools/analysis_tools/analyze_logs.py plot_curve tools/logs/pp-3class.log.
↪json tools/logs/pp.log.json --keys KITTI/Car_3D_moderate_strict --legend pp-
↪3class pp --mode eval --interval 2
```

You can also compute the average training speed.

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

The output is expected to be like the following.

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

CHAPTER

TWENTYNINE

VISUALIZATION

29.1 Results

To see the prediction results of trained models, you can run the following command

```
python tools/test.py ${CONFIG_FILE} ${CKPT_PATH} --show --show-dir ${SHOW_DIR}
```

After running this command, plotted results including input data and the output of networks visualized on the input (e.g. `***_points.obj` and `***_pred.obj` in single-modality 3D detection task) will be saved in `${SHOW_DIR}`.

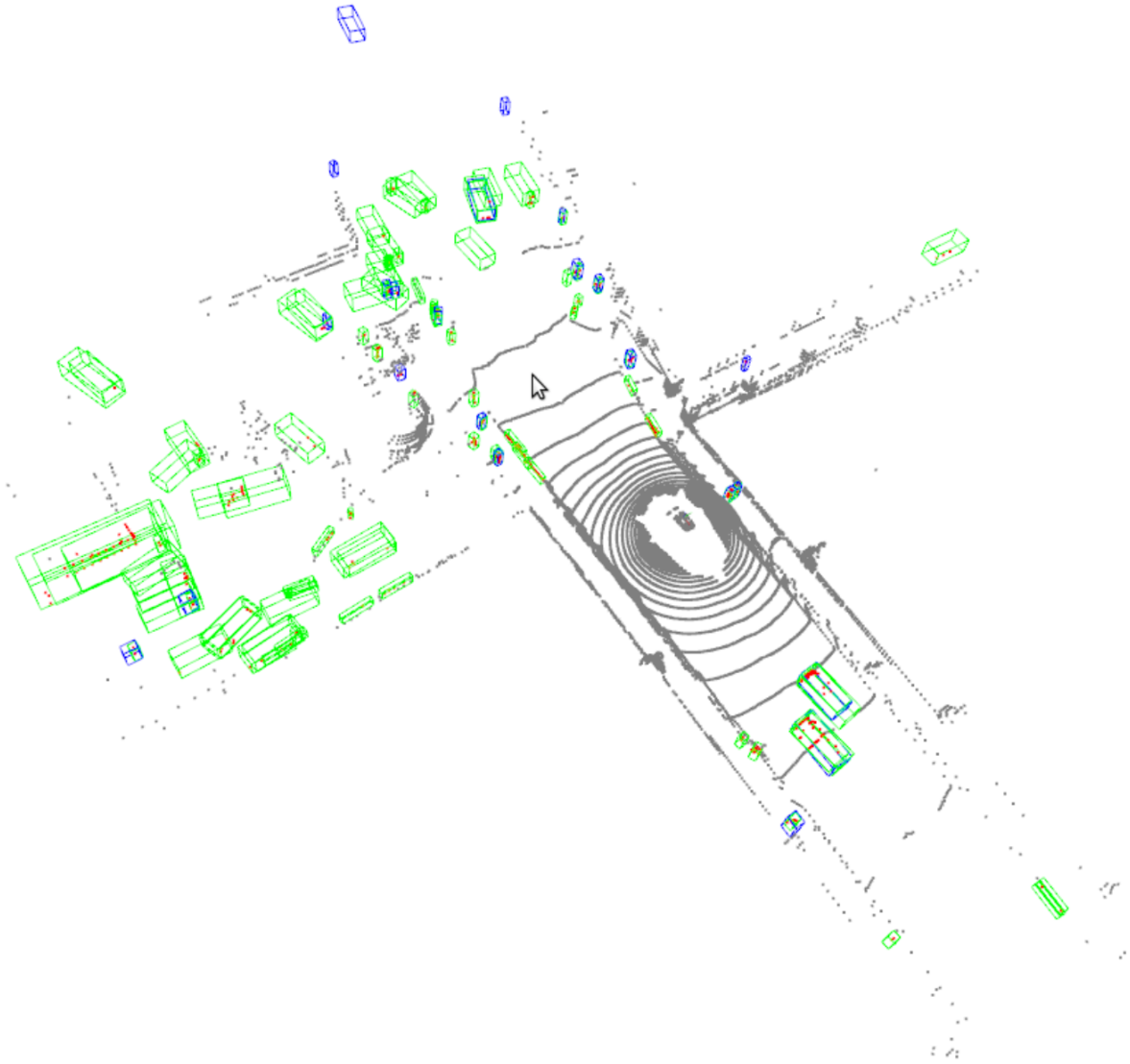
To see the prediction results during evaluation, you can run the following command

```
python tools/test.py ${CONFIG_FILE} ${CKPT_PATH} --eval 'mAP' --eval-options 'show=True'
↪'out_dir=${SHOW_DIR}'
```

After running this command, you will obtain the input data, the output of networks and ground-truth labels visualized on the input (e.g. `***_points.obj`, `***_pred.obj`, `***_gt.obj`, `***_img.png` and `***_pred.png` in multi-modality detection task) in `${SHOW_DIR}`. When `show` is enabled, Open3D will be used to visualize the results online. If you are running test in remote server without GUI, the online visualization is not supported, you can set `show=False` to only save the output results in `{SHOW_DIR}`.

As for offline visualization, you will have two options. To visualize the results with `Open3D` backend, you can run the following command

```
python tools/misc/visualize_results.py ${CONFIG_FILE} --result ${RESULTS_PATH} --show-
↪dir ${SHOW_DIR}
```



Or you can use 3D visualization software such as the MeshLab to open these files under `${SHOW_DIR}` to see the 3D detection output. Specifically, open `***_points.obj` to see the input point cloud and open `***_pred.obj` to see the predicted 3D bounding boxes. This allows the inference and results generation to be done in remote server and the users can open them on their host with GUI.

Notice: The visualization API is a little unstable since we plan to refactor these parts together with MMDetection in the future.

29.2 Dataset

We also provide scripts to visualize the dataset without inference. You can use `tools/misc/browse_dataset.py` to show loaded data and ground-truth online and save them on the disk. Currently we support single-modality 3D detection and 3D segmentation on all the datasets, multi-modality 3D detection on KITTI and SUN RGB-D, as well as monocular 3D detection on nuScenes. To browse the KITTI dataset, you can run the following command

```
python tools/misc/browse_dataset.py configs/_base_/datasets/kitti-3d-3class.py --task␣
↪det --output-dir ${OUTPUT_DIR} --online
```

Notice: Once specifying `--output-dir`, the images of views specified by users will be saved when pressing _ESC_ in open3d window. If you don't have a monitor, you can remove the `--online` flag to only save the visualization results and browse them offline.

To verify the data consistency and the effect of data augmentation, you can also add `--aug` flag to visualize the data after data augmentation using the command as below:

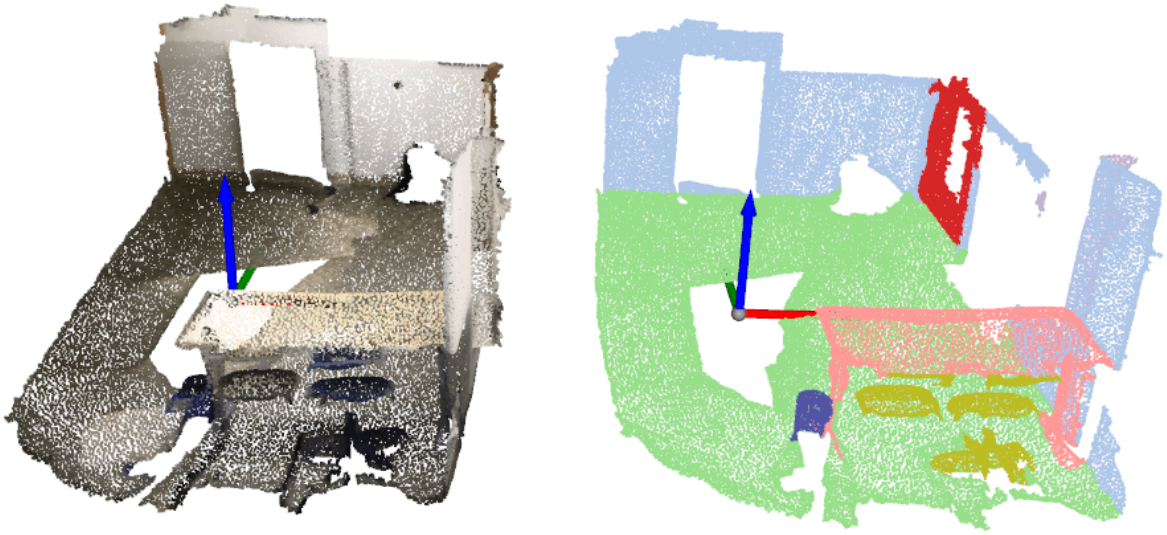
```
python tools/misc/browse_dataset.py configs/_base_/datasets/kitti-3d-3class.py --task␣
↪det --aug --output-dir ${OUTPUT_DIR} --online
```

If you also want to show 2D images with 3D bounding boxes projected onto them, you need to find a config that supports multi-modality data loading, and then change the `--task` args to `multi_modality-det`. An example is showed below

```
python tools/misc/browse_dataset.py configs/mvxnet/dv_mvx-fpn_second_secfpn_adamw_2x8_
↪80e_kitti-3d-3class.py --task multi_modality-det --output-dir ${OUTPUT_DIR} --online
```

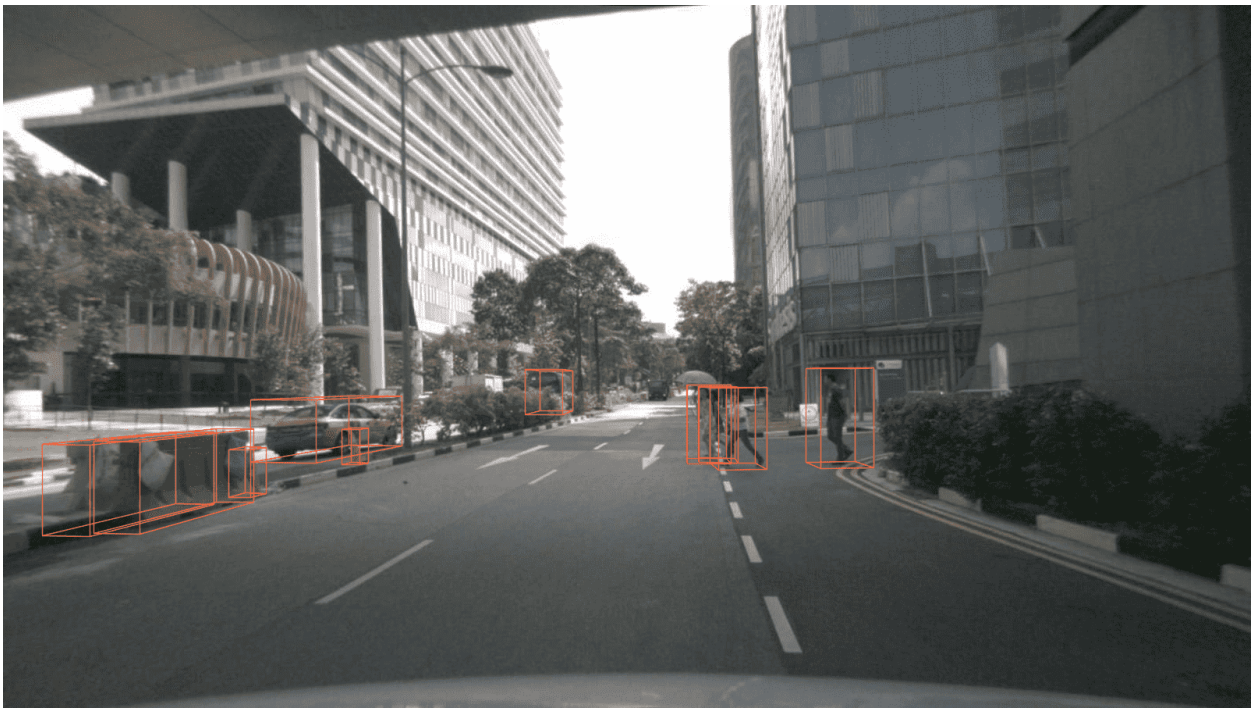
You can simply browse different datasets using different configs, e.g. visualizing the ScanNet dataset in 3D semantic segmentation task

```
python tools/misc/browse_dataset.py configs/_base_/datasets/scannet_seg-3d-20class.py --
↪task seg --output-dir ${OUTPUT_DIR} --online
```



And browsing the nuScenes dataset in monocular 3D detection task

```
python tools/misc/browse_dataset.py configs/_base_/datasets/nus-mono3d.py --task mono-
↪det --output-dir ${OUTPUT_DIR} --online
```



CHAPTER

THIRTY

MODEL SERVING

Note: This tool is still experimental now, only SECOND is supported to be served with TorchServe. We'll support more models in the future.

In order to serve an `MMDetection3D` model with TorchServe, you can follow the steps:

30.1 1. Convert the model from MMDetection3D to TorchServe

```
python tools/deployment/mmdet3d2torchserve.py ${CONFIG_FILE} ${CHECKPOINT_FILE} \
--output-folder ${MODEL_STORE} \
--model-name ${MODEL_NAME}
```

Note: ${MODEL_STORE} needs to be an absolute path to a folder.

30.2 2. Build `mmdet3d-serve` docker image

```
docker build -t mmdet3d-serve:latest docker/serve/
```

30.3 3. Run `mmdet3d-serve`

Check the official docs for running TorchServe with docker.

In order to run it on the GPU, you need to install nvidia-docker. You can omit the `--gpus` argument in order to run on the CPU.

Example:

```
docker run --rm \
--cpus 8 \
--gpus device=0 \
-p8080:8080 -p8081:8081 -p8082:8082 \
--mount type=bind,source=$MODEL_STORE,target=/home/model-server/model-store \
mmdet3d-serve:latest
```

Read the docs about the Inference (8080), Management (8081) and Metrics (8082) APis

30.4 4. Test deployment

You can use `test_torchserver.py` to compare result of torchserver and pytorch.

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

Example:

```
python tools/deployment/test_torchserver.py demo/data/kitti/kitti_000008.bin configs/
↪second/hv_second_secfpn_6x8_80e_kitti-3d-car.py checkpoints/hv_second_secfpn_6x8_80e_
↪kitti-3d-car_20200620_230238-393f000c.pth second
```

CHAPTER

THIRTYONE

MODEL COMPLEXITY

You can use `tools/analysis_tools/get_flops.py` in MMDetection3D, a script adapted from flops-counter.pytorch, to compute the FLOPs and params of a given model.

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

You will get the results like this.

```
==============================
Input shape: (40000, 4)
Flops: 5.78 GFLOPs
Params: 953.83 k
==============================
```

Note: This tool is still experimental and we do not guarantee that the number is absolutely correct. You may well use the result for simple comparisons, but double check it before you adopt it in technical reports or papers.

1. FLOPs are related to the input shape while parameters are not. The default input shape is (1, 40000, 4).
2. Some operators are not counted into FLOPs like GN and custom operators. Refer to `mmcv.cnn.get_model_complexity_info()` for details.
3. We currently only support FLOPs calculation of single-stage models with single-modality input (point cloud or image). We will support two-stage and multi-modality models in the future.

CHAPTER

THIRTYTWO

MODEL CONVERSION

32.1 RegNet model to MMDetection

`tools/model_converters/regnet2mmdet.py` convert keys in pycls pretrained RegNet models to MMDetection style.

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

32.2 Detectron ResNet to Pytorch

`tools/detectron2pytorch.py` in MMDetection could convert keys in the original detectron pretrained ResNet models to PyTorch style.

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

32.3 Prepare a model for publishing

`tools/model_converters/publish_model.py` helps users to prepare their model for publishing.

Before you upload a model to AWS, you may want to

1. convert model weights to CPU tensors
2. delete the optimizer states and
3. compute the hash of the checkpoint file and append the hash id to the filename.

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

E.g.,

```
python tools/model_converters/publish_model.py work_dirs/faster_rcnn/latest.pth faster_
↪rcnn_r50_fpn_1x_20190801.pth
```

The final output filename will be `faster_rcnn_r50_fpn_1x_20190801-{hash id}.pth`.

CHAPTER

THIRTYTHREE

DATASET CONVERSION

`tools/data_converter/` contains tools for converting datasets to other formats. Most of them convert datasets to pickle based info files, like kitti, nuscenes and lyft. Waymo converter is used to reorganize waymo raw data like KITTI style. Users could refer to them for our approach to converting data format. It is also convenient to modify them to use as scripts like nuImages converter.

To convert the nuImages dataset into COCO format, please use the command below:

```
python -u tools/data_converter/nuimage_converter.py --data-root ${DATA_ROOT} --version $
↪{VERSIONS} \
                                                    --out-dir ${OUT_DIR} --nproc ${NUM_
↪WORKERS} --extra-tag ${TAG}
```

- `--data-root`: the root of the dataset, defaults to `./data/nuimages`.
- `--version`: the version of the dataset, defaults to `v1.0-mini`. To get the full dataset, please use `--version v1.0-train v1.0-val v1.0-mini`
- `--out-dir`: the output directory of annotations and semantic masks, defaults to `./data/nuimages/annotations/`.
- `--nproc`: number of workers for data preparation, defaults to `4`. Larger number could reduce the preparation time as images are processed in parallel.
- `--extra-tag`: extra tag of the annotations, defaults to `nuimages`. This can be used to separate different annotations processed in different time for study.

More details could be referred to the doc for dataset preparation and README for nuImages dataset.

CHAPTER

THIRTYFOUR

MISCELLANEOUS

34.1 Print the entire config

`tools/misc/print_config.py` prints the whole config verbatim, expanding all its imports.

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

CHAPTER

THIRTYFIVE

BENCHMARKS

Here we benchmark the training and testing speed of models in MMDetection3D, with some other open source 3D detection codebases.

35.1 Settings

- Hardwares: 8 NVIDIA Tesla V100 (32G) GPUs, Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz
- Software: Python 3.7, CUDA 10.1, cuDNN 7.6.5, PyTorch 1.3, numba 0.48.0.
- Model: Since all the other codebases implements different models, we compare the corresponding models including SECOND, PointPillars, Part-A2, and VoteNet with them separately.
- Metrics: We use the average throughput in iterations of the entire training run and skip the first 50 iterations of each epoch to skip GPU warmup time.

35.2 Main Results

We compare the training speed (samples/s) with other codebases if they implement the similar models. The results are as below, the greater the numbers in the table, the faster of the training process. The models that are not supported by other codebases are marked by ×.

35.3 Details of Comparison

35.3.1 Modification for Calculating Speed

- **MMDetection3D**: We try to use as similar settings as those of other codebases as possible using benchmark configs.
- **Det3D**: For comparison with Det3D, we use the commit 519251e.
- **OpenPCDet**: For comparison with OpenPCDet, we use the commit b32fbddb.

  For training speed, we add code to record the running time in the file `./tools/train_utils/train_utils.py`. We calculate the speed of each epoch, and report the average speed of all the epochs.

```
diff --git a/tools/train_utils/train_utils.py b/tools/train_utils/train_utils.py
index 91f21dd..021359d 100644
--- a/tools/train_utils/train_utils.py
```

(continues on next page)

(continued from previous page)

```
+++ b/tools/train_utils/train_utils.py
@@ -2,6 +2,7 @@ import torch
 import os
 import glob
 import tqdm
+import datetime
 from torch.nn.utils import clip_grad_norm_


@@ -13,7 +14,10 @@ def train_one_epoch(model, optimizer, train_loader, model_func,␣
↪lr_scheduler, ac
     if rank == 0:
         pbar = tqdm.tqdm(total=total_it_each_epoch, leave=leave_pbar, desc='train',
↪ dynamic_ncols=True)

+    start_time = None
     for cur_it in range(total_it_each_epoch):
+        if cur_it > 49 and start_time is None:
+            start_time = datetime.datetime.now()
         try:
             batch = next(dataloader_iter)
         except StopIteration:
@@ -55,9 +59,11 @@ def train_one_epoch(model, optimizer, train_loader, model_func,␣
↪lr_scheduler, ac
                 tb_log.add_scalar('learning_rate', cur_lr, accumulated_iter)
                 for key, val in tb_dict.items():
                     tb_log.add_scalar('train_' + key, val, accumulated_iter)
+    endtime = datetime.datetime.now()
+    speed = (endtime - start_time).seconds / (total_it_each_epoch - 50)
     if rank == 0:
         pbar.close()
-    return accumulated_iter
+    return accumulated_iter, speed


 def train_model(model, optimizer, train_loader, model_func, lr_scheduler, optim_
↪cfg,
@@ -65,6 +71,7 @@ def train_model(model, optimizer, train_loader, model_func, lr_
↪scheduler, optim_
                 lr_warmup_scheduler=None, ckpt_save_interval=1, max_ckpt_save_
↪num=50,
                 merge_all_iters_to_one_epoch=False):
     accumulated_iter = start_iter
+    speeds = []
     with tqdm.trange(start_epoch, total_epochs, desc='epochs', dynamic_ncols=True,␣
↪leave=(rank == 0)) as tbar:
         total_it_each_epoch = len(train_loader)
         if merge_all_iters_to_one_epoch:
@@ -82,7 +89,7 @@ def train_model(model, optimizer, train_loader, model_func, lr_
↪scheduler, optim_
                 cur_scheduler = lr_warmup_scheduler
             else:
```

(continues on next page)

(continued from previous page)

```
                cur_scheduler = lr_scheduler
-           accumulated_iter = train_one_epoch(
+           accumulated_iter, speed = train_one_epoch(
                model, optimizer, train_loader, model_func,
                lr_scheduler=cur_scheduler,
                accumulated_iter=accumulated_iter, optim_cfg=optim_cfg,
@@ -91,7 +98,7 @@ def train_model(model, optimizer, train_loader, model_func, lr_
↪scheduler, optim_
                total_it_each_epoch=total_it_each_epoch,
                dataloader_iter=dataloader_iter
            )
-
+           speeds.append(speed)
            # save trained model
            trained_epoch = cur_epoch + 1
            if trained_epoch % ckpt_save_interval == 0 and rank == 0:
@@ -107,6 +114,8 @@ def train_model(model, optimizer, train_loader, model_func, lr_
↪scheduler, optim_
                save_checkpoint(
                    checkpoint_state(model, optimizer, trained_epoch, accumulated_
↪iter), filename=ckpt_name,
                )
+           print(speed)
+    print(f'*******{sum(speeds) / len(speeds)}******')


 def model_state_to_cpu(model_state):
```

35.3.2 VoteNet

- **MMDetection3D**: With release v0.1.0, run

```
./tools/dist_train.sh configs/votenet/votenet_16x8_sunrgbd-3d-10class.py 8 --no-
↪validate
```

- **votenet**: At commit 2f6d6d3, run

```
python train.py --dataset sunrgbd --batch_size 16
```

Then benchmark the test speed by running

```
python eval.py --dataset sunrgbd --checkpoint_path log_sunrgbd/checkpoint.tar --
↪batch_size 1 --dump_dir eval_sunrgbd --cluster_sampling seed_fps --use_3d_nms --
↪use_cls_nms --per_class_proposal
```

Note that eval.py is modified to compute inference time.

```
diff --git a/eval.py b/eval.py
  index c0b2886..04921e9 100644
  --- a/eval.py
  +++ b/eval.py
```

(continues on next page)

(continued from previous page)

```
 @@ -10,6 +10,7 @@ import os
  import sys
  import numpy as np
  from datetime import datetime
 +import time
  import argparse
  import importlib
  import torch
 @@ -28,7 +29,7 @@ parser.add_argument('--checkpoint_path', default=None, help=
↪'Model checkpoint pa
  parser.add_argument('--dump_dir', default=None, help='Dump dir to save sample␣
↪outputs [default: None]')
  parser.add_argument('--num_point', type=int, default=20000, help='Point Number␣
↪[default: 20000]')
  parser.add_argument('--num_target', type=int, default=256, help='Point Number␣
↪[default: 256]')
 -parser.add_argument('--batch_size', type=int, default=8, help='Batch Size during␣
↪training [default: 8]')
 +parser.add_argument('--batch_size', type=int, default=1, help='Batch Size during␣
↪training [default: 8]')
  parser.add_argument('--vote_factor', type=int, default=1, help='Number of votes␣
↪generated from each seed [default: 1]')
  parser.add_argument('--cluster_sampling', default='vote_fps', help='Sampling␣
↪strategy for vote clusters: vote_fps, seed_fps, random [default: vote_fps]')
  parser.add_argument('--ap_iou_thresholds', default='0.25,0.5', help='A list of␣
↪AP IoU thresholds [default: 0.25,0.5]')
 @@ -132,6 +133,7 @@ CONFIG_DICT = {'remove_empty_box': (not FLAGS.faster_eval),
↪'use_3d_nms': FLAGS.
  # ------------------------------------------------------------------------␣
↪GLOBAL CONFIG END

  def evaluate_one_epoch():
 +    time_list = list()
      stat_dict = {}
      ap_calculator_list = [APCalculator(iou_thresh, DATASET_CONFIG.class2type) \
          for iou_thresh in AP_IOU_THRESHOLDS]
 @@ -144,6 +146,8 @@ def evaluate_one_epoch():

          # Forward pass
          inputs = {'point_clouds': batch_data_label['point_clouds']}
 +        torch.cuda.synchronize()
 +        start_time = time.perf_counter()
          with torch.no_grad():
              end_points = net(inputs)

 @@ -161,6 +165,12 @@ def evaluate_one_epoch():

          batch_pred_map_cls = parse_predictions(end_points, CONFIG_DICT)
          batch_gt_map_cls = parse_groundtruths(end_points, CONFIG_DICT)
 +        torch.cuda.synchronize()
 +        elapsed = time.perf_counter() - start_time
 +        time_list.append(elapsed)
```

(continues on next page)

(continued from previous page)

```
 +
 +        if len(time_list==200):
 +            print("average inference time: %4f"%(sum(time_list[5:])/len(time_
↪list[5:])))
          for ap_calculator in ap_calculator_list:
              ap_calculator.step(batch_pred_map_cls, batch_gt_map_cls)
```

35.3.3 PointPillars-car

- **MMDetection3D**: With release v0.1.0, run

```
./tools/dist_train.sh configs/benchmark/hv_pointpillars_secfpn_3x8_100e_det3d_kitti-
↪3d-car.py 8 --no-validate
```

- **Det3D**: At commit 519251e, use `kitti_point_pillars_mghead_syncbn.py` and run

```
./tools/scripts/train.sh --launcher=slurm --gpus=8
```

Note that the config in train.sh is modified to train point pillars.

```
diff --git a/tools/scripts/train.sh b/tools/scripts/train.sh
index 3a93f95..461e0ea 100755
--- a/tools/scripts/train.sh
+++ b/tools/scripts/train.sh
@@ -16,9 +16,9 @@ then
 fi

 # Voxelnet
-python -m torch.distributed.launch --nproc_per_node=8 ./tools/train.py examples/
↪second/configs/  kitti_car_vfev3_spmiddlefhd_rpn1_mghead_syncbn.py --work_dir=
↪$SECOND_WORK_DIR
+# python -m torch.distributed.launch --nproc_per_node=8 ./tools/train.py examples/
↪second/configs/  kitti_car_vfev3_spmiddlefhd_rpn1_mghead_syncbn.py --work_dir=
↪$SECOND_WORK_DIR
 # python -m torch.distributed.launch --nproc_per_node=8 ./tools/train.py examples/
↪cbgs/configs/  nusc_all_vfev3_spmiddleresnetfhd_rpn2_mghead_syncbn.py --work_dir=
↪$NUSC_CBGS_WORK_DIR
 # python -m torch.distributed.launch --nproc_per_node=8 ./tools/train.py examples/
↪second/configs/  lyft_all_vfev3_spmiddleresnetfhd_rpn2_mghead_syncbn.py --work_
↪dir=$LYFT_CBGS_WORK_DIR

 # PointPillars
-# python -m torch.distributed.launch --nproc_per_node=8 ./tools/train.py ./
↪examples/point_pillars/configs/  original_pp_mghead_syncbn_kitti.py --work_dir=
↪$PP_WORK_DIR
+python -m torch.distributed.launch --nproc_per_node=8 ./tools/train.py ./examples/
↪point_pillars/configs/  kitti_point_pillars_mghead_syncbn.py
```

35.3.4 PointPillars-3class

- **MMDetection3D**: With release v0.1.0, run

```
./tools/dist_train.sh configs/benchmark/hv_pointpillars_secfpn_4x8_80e_pcdet_kitti-
↪3d-3class.py 8 --no-validate
```

- **OpenPCDet**: At commit b32fbddb, run

```
cd tools
sh scripts/slurm_train.sh ${PARTITION} ${JOB_NAME} 8  --cfg_file ./cfgs/kitti_
↪models/pointpillar.yaml --batch_size 32  --workers 32 --epochs 80
```

35.3.5 SECOND

For SECOND, we mean the SECONDv1.5 that was first implemented in second.Pytorch. Det3D's implementation of SECOND uses its self-implemented Multi-Group Head, so its speed is not compatible with other codebases.

- **MMDetection3D**: With release v0.1.0, run

```
./tools/dist_train.sh configs/benchmark/hv_second_secfpn_4x8_80e_pcdet_kitti-3d-
↪3class.py 8 --no-validate
```

- **OpenPCDet**: At commit b32fbddb, run

```
cd tools
sh ./scripts/slurm_train.sh ${PARTITION} ${JOB_NAME} 8  --cfg_file ./cfgs/kitti_
↪models/second.yaml --batch_size 32  --workers 32 --epochs 80
```

35.3.6 Part-A2

- **MMDetection3D**: With release v0.1.0, run

```
./tools/dist_train.sh configs/benchmark/hv_PartA2_secfpn_4x8_cyclic_80e_pcdet_kitti-
↪3d-3class.py 8 --no-validate
```

- **OpenPCDet**: At commit b32fbddb, train the model by running

```
cd tools
sh ./scripts/slurm_train.sh ${PARTITION} ${JOB_NAME} 8  --cfg_file ./cfgs/kitti_
↪models/PartA2.yaml --batch_size 32 --workers 32 --epochs 80
```

CHAPTER

THIRTYSIX

FAQ

We list some potential troubles encountered by users and developers, along with their corresponding solutions. Feel free to enrich the list if you find any frequent issues and contribute your solutions to solve them. If you have any trouble with environment configuration, model training, etc, please create an issue using the provided templates and fill in all required information in the template.

36.1 MMCV/MMDet/MMDet3D Installation

- Compatibility issue between MMCV, MMDetection, MMSegmentation and MMDection3D; "ConvWS is already registered in conv layer"; "AssertionError: MMCV==xxx is used but incompatible. Please install mmcv>=xxx, <=xxx."

  The required versions of MMCV, MMDetection and MMSegmentation for different versions of MMDetection3D are as below. Please install the correct version of MMCV, MMDetection and MMSegmentation to avoid installation issues.

- If you faced the error shown below when importing open3d:

  ```
  OSError: /lib/x86_64-linux-gnu/libm.so.6: version 'GLIBC_2.27' not found
  ```

  please downgrade open3d to 0.9.0.0, because the latest open3d needs the support of file 'GLIBC_2.27', which only exists in Ubuntu 18.04, not in Ubuntu 16.04.

- If you faced the error when importing pycocotools, this is because nuscenes-devkit installs pycocotools but mmdet relies on mmpycocotools. The current workaround is as below. We will migrate to use pycocotools in the future.

  ```
  pip uninstall pycocotools mmpycocotools
  pip install mmpycocotools
  ```

  NOTE: We have migrated to use pycocotools in mmdet3d >= 0.13.0.

- If you face the error shown below when importing pycocotools:

  ```
  ValueError: numpy.ndarray size changed, may indicate binary incompatibility.
  Expected 88 from C header, got 80 from PyObject
  ```

  please downgrade pycocotools to 2.0.1 because of the incompatibility between the newest pycocotools and numpy < 1.20.0. Or you can compile and install the latest pycocotools from source as below:

  ```
  pip install -e "git+https://github.com/cocodataset/cocoapi#egg=pycocotools&subdirectory=PythonAPI"
  ```

  or

  ```
  pip install -e "git+https://github.com/ppwwyyxx/cocoapi#egg=pycocotools&subdirectory=PythonAPI"
  ```

36.2 How to annotate point cloud?

MMDetection3D does not support point cloud annotation. Some open-source annotation tool are offered for reference:

- SUSTechPOINTS
- LATTE

Besides, we improved LATTE for better use. More details can be found here.

CHAPTER

THIRTYSEVEN

V1.0.0RC4

37.1 Number of points in SUN RGB-D preprocessing

Before mmdet3d version 1.0.0rc4 we sampled 50000 points following VoteNet preprocessing. Since 1.0.0v4 version we do not limit the maximum number of points during preprocessing, giving the users more flexibility with `PointSample`. The SUN RGB-D annotations should be updated in case the user wants to achieve best metrics for models supporting more than 50000 input points.

Please refer to the SUN RGB-D README.md for more details.

37.2 Fix a small amount of missing objects during S3DIS preprocessing

We fixed a bug in `tools/data_converter/s3dis_data_utils.py` leading to miss not more than one object per scene. Users need to regenerate the annotations with `tools/create_data.py`.

CHAPTER

THIRTYEIGHT

V1.0.0RC1

38.1 Operators Migration

We have adopted CUDA operators compiled from mmcv and removed all the CUDA operators in mmdet3d. We now do not need to compile the CUDA operators in mmdet3d anymore.

38.2 Waymo dataset converter refactoring

In this version we did a major code refactoring that boosted the performance of waymo dataset conversion by multiprocessing. Meanwhile, we also fixed the imprecise timestamps saving issue in waymo dataset conversion. This change introduces following backward compatibility breaks:

- The point cloud .bin files of waymo dataset need to be regenerated. In the .bin files each point occupies 6 `float32` and the meaning of the last `float32` now changed from **imprecise timestamps** to **range frame offset**. The **range frame offset** for each point is calculated as`ri * h * w + row * w + col` if the point is from the **TOP** lidar or `-1` otherwise. The `h`, `w` denote the height and width of the TOP lidar's range frame. The `ri`, `row`, `col` denote the return index, the row and the column of the range frame where each point locates. Following tables show the difference across the change:

Before

After

- The objects' point cloud .bin files in the GT-database of waymo dataset need to be regenerated because we also dumped the range frame offset for each point into it. Following tables show the difference across the change:

Before

After

- Any configuration that uses waymo dataset with GT Augmentation should change the `db_sampler.points_loader.load_dim` from `5` to `6`.

CHAPTER

THIRTYNINE

V1.0.0RC0

39.1 Coordinate system refactoring

In this version, we did a major code refactoring which improved the consistency among the three coordinate systems (and corresponding box representation), LiDAR, Camera, and Depth. A brief summary for this refactoring is as follows:

- The three coordinate systems are all right-handed now (which means the yaw angle increases in the counter-clockwise direction).
- The LiDAR system `(x_size, y_size, z_size)` corresponds to `(l, w, h)` instead of `(w, l, h)`. This is more natural since `l` is parallel with the direction where the yaw angle is zero, and we prefer using the positive direction of the `x` axis as that direction, which is exactly how we define yaw angle in Depth and Camera coordinate systems.
- The APIs for box-related operations are improved and now are more user-friendly.

39.1.1 *NOTICE!!*

Since definitions of box representation have changed, the annotation data of most datasets require updating:

- SUN RGB-D: Yaw angles in the annotation should be reversed.
- KITTI: For LiDAR boxes in GT databases, (x_size, y_size, z_size, yaw) out of (x, y, z, x_size, y_size, z_size) should be converted from the old LiDAR coordinate system to the new one. The training/validation data annotations should be left unchanged since they are under the Camera coordinate system, which is unmodified after the refactoring.
- Waymo: Same as KITTI.
- nuScenes: For LiDAR boxes in training/validation data and GT databases, (x_size, y_size, z_size, yaw) out of (x, y, z, x_size, y_size, z_size) should be converted.
- Lyft: Same as nuScenes.

Please regenerate the data annotation/GT database files or use `update_data_coords.py` to update the data.

To use boxes under Depth and LiDAR coordinate systems, or to convert boxes between different coordinate systems, users should be aware of the difference between the old and new definitions. For example, the rotation, flipping, and bev functions of `DepthInstance3DBoxes` and `LiDARInstance3DBoxes` and box conversion functions have all been reimplemented in the refactoring.

Consequently, functions like `output_to_lyft_box` undergo small modification to adapt to the new LiDAR/Depth box.

Since the LiDAR system `(x_size, y_size, z_size)` now corresponds to `(l, w, h)` instead of `(w, l, h)`, the anchor sizes for LiDAR boxes are also changed, e.g., from `[1.6, 3.9, 1.56]` to `[3.9, 1.6, 1.56]`.

Functions only involving points are generally unaffected except if they rely on some refactored utility functions such as `rotation_3d_in_axis`.

39.1.2 Other BC-breaking or new features:

- `array_converter`: Please refer to array_converter.py. Functions wrapped with `array_converter` can convert array-like input types of `torch.Tensor`, `np.ndarray`, and `list/tuple/float` to `torch.Tensor` to process in an unified PyTorch pipeline. The result may finally be converted back to the input type. Most functions in utils.py are wrapped with `array_converter`.
- `points_in_boxes` and `points_in_boxes_batch` will be deprecated soon. They are renamed to `points_in_boxes_part` and `points_in_boxes_all` respectively, with more detailed docstrings. The major difference of the two functions is that if a point is enclosed by multiple boxes, `points_in_boxes_part` will only return the index of the first enclosing box while `points_in_boxes_all` will return all the indices of enclosing boxes.
- `rotation_3d_in_axis`: Please refer to utils.py. Now this function supports multiple input types and more options. The function with the same name in box_np_ops.py is deleted since we do not need another function to tackle with NumPy data. `rotation_2d`, `points_cam2img`, and `limit_period` in box_np_ops.py are also deleted for the same reason.
- `bev` method of `CameraInstance3DBoxes`: Changed it to be consistent with the definition of bev in Depth and LiDAR coordinate systems.
- Data augmentation utils in data_augment_utils.py now follow the rules of a right-handed system.
- We do not need the yaw hacking in KITTI anymore after refining `get_direction_target`. Interested users may refer to PR #677 .

CHAPTER

FORTY

0.16.0

40.1 Returned values of `QueryAndGroup` operation

We modified the returned `grouped_xyz` value of operation `QueryAndGroup` to support PAConv segmentor. Originally, the `grouped_xyz` is centered by subtracting the grouping centers, which represents the relative positions of grouped points. Now, we didn't perform such subtraction and the returned `grouped_xyz` stands for the absolute coordinates of these points.

Note that, the other returned variables of `QueryAndGroup` such as `new_features`, `unique_cnt` and `grouped_idx` are not affected.

40.2 NuScenes coco-style data pre-processing

We remove the rotation and dimension hack in the monocular 3D detection on nuScenes. Specifically, we transform the rotation and dimension of boxes defined by nuScenes devkit to the coordinate system of our `CameraInstance3DBoxes` in the pre-processing and transform them back in the post-processing. In this way, we can remove the corresponding hack used in the visualization tools. The modification also guarantees the correctness of all the operations based on our `CameraInstance3DBoxes` (such as NMS and flip augmentation) when training monocular 3D detectors.

The modification only influences nuScenes coco-style json files. Please re-run the nuScenes data preparation script if necessary. See more details in the PR #744.

40.3 ScanNet dataset for ImVoxelNet

We adopt a new pre-processing procedure for the ScanNet dataset in order to support ImVoxelNet, which is a multi-view method requiring image data. In previous versions of MMDetection3D, ScanNet dataset was only used for point cloud based 3D detection and segmentation methods. We plan adding ImVoxelNet to our model zoo, thus updating ScanNet correspondingly by adding image-related pre-processing steps. Specifically, we made these changes:

- Add script for extracting RGB data.
- Update script for annotation creating.
- Add instructions in the documents on preparing image data.

Please refer to the ScanNet README.md for more details.

CHAPTER

FORTYONE

0.15.0

41.1 MMCV Version

In order to fix the problem that the priority of EvalHook is too low, all hook priorities have been re-adjusted in 1.3.8, so MMDetection 2.14.0 needs to rely on the latest MMCV 1.3.8 version. For related information, please refer to #1120, for related issues, please refer to #5343.

41.2 Unified parameter initialization

To unify the parameter initialization in OpenMMLab projects, MMCV supports `BaseModule` that accepts `init_cfg` to allow the modules' parameters initialized in a flexible and unified manner. Now the users need to explicitly call `model.init_weights()` in the training script to initialize the model (as in here, previously this was handled by the detector. Please refer to PR #622 for details.

41.3 BackgroundPointsFilter

We modified the dataset augmentation function `BackgroundPointsFilter`(here). In previous version of MMdetection3D, `BackgroundPointsFilter` changes the gt_bboxes_3d's bottom center to the gravity center. In MMDetection3D 0.15.0, `BackgroundPointsFilter` will not change it. Please refer to PR #609 for details.

41.4 Enhance `IndoorPatchPointSample` transform

We enhance the pipeline function `IndoorPatchPointSample` used in point cloud segmentation task by adding more choices for patch selection. Also, we plan to remove the unused parameter `sample_rate` in the future. Please modify the code as well as the config files accordingly if you use this transform.

CHAPTER

FORTYTWO

0.14.0

42.1 Dataset class for 3D segmentation task

We remove a useless parameter `label_weight` from segmentation datasets including `Custom3DSegDataset`, `ScanNetSegDataset` and `S3DISSegDataset` since this weight is utilized in the loss function of model class. Please modify the code as well as the config files accordingly if you use or inherit from these codes.

42.2 ScanNet data pre-processing

We adopt new pre-processing and conversion steps of ScanNet dataset. In previous versions of MMDetection3D, ScanNet dataset was only used for 3D detection task, where we trained on the training set and tested on the validation set. In MMDetection3D 0.14.0, we further support 3D segmentation task on ScanNet, which includes online benchmarking on test set. Since the alignment matrix is not provided for test set data, we abandon the alignment of points in data generation steps to support both tasks. Besides, as 3D segmentation requires per-point prediction, we also remove the down-sampling step in data generation.

- In the new ScanNet processing scripts, we save the unaligned points for all the training, validation and test set. For train and val set with annotations, we also store the `axis_align_matrix` in data infos. For ground-truth bounding boxes, we store boxes in both aligned and unaligned coordinates with key `gt_boxes_upright_depth` and key `unaligned_gt_boxes_upright_depth` respectively in data infos.
- In `ScanNetDataset`, we now load the `axis_align_matrix` as a part of data annotations. If it is not contained in old data infos, we will use identity matrix for compatibility. We also add a transform function `GlobalAlignment` in ScanNet detection data pipeline to align the points.
- Since the aligned boxes share the same key as in old data infos, we do not need to modify the code related to it. But do remember that they are not in the same coordinate system as the saved points.
- There is an `PointSample` pipeline in the data pipelines for ScanNet detection task which down-samples points. So removing down-sampling in data generation will not affect the code.

We have trained a VoteNet model on the newly processed ScanNet dataset and get similar benchmark results. In order to prepare ScanNet data for both detection and segmentation tasks, please re-run the new pre-processing scripts following the ScanNet README.md.

CHAPTER

FORTYTHREE

0.12.0

43.1 SUNRGBD dataset for ImVoteNet

We adopt a new pre-processing procedure for the SUNRGBD dataset in order to support ImVoteNet, which is a multi-modality method requiring both image and point cloud data. In previous versions of MMDetection3D, SUNRGBD dataset was only used for point cloud based 3D detection methods. In MMDetection3D 0.12.0, we add ImVoteNet to our model zoo, thus updating SUNRGBD correspondingly by adding image-related pre-processing steps. Specifically, we made these changes:

- Fix a bug in the image file path in meta data.
- Convert calibration matrices from double to float to avoid type mismatch in further operations.
- Add instructions in the documents on preparing image data.

Please refer to the SUNRGBD README.md for more details.

CHAPTER

FORTYFOUR

0.6.0

44.1 VoteNet and H3DNet model structure update

In MMDetection 0.6.0, we updated the model structures of VoteNet and H3DNet, therefore model checkpoints generated by MMDetection < 0.6.0 should be first converted to a format compatible with the latest structures via convert_votenet_checkpoints.py and convert_h3dnet_checkpoints.py . For more details, please refer to the VoteNet README.md and H3DNet README.md.

CHAPTER

FORTYFIVE

MMDET3D.CORE

45.1 anchor

class mmdet3d.core.anchor.**AlignedAnchor3DRangeGenerator**(*align_corner=False*, ***kwargs*)

Aligned 3D Anchor Generator by range.

This anchor generator uses a different manner to generate the positions of anchors' centers from *Anchor3DRangeGenerator*.

Note: The *align* means that the anchor's center is aligned with the voxel grid, which is also the feature grid. The previous implementation of *Anchor3DRangeGenerator* does not generate the anchors' center according to the voxel grid. Rather, it generates the center by uniformly distributing the anchors inside the minimum and maximum anchor ranges according to the feature map sizes. However, this makes the anchors center does not match the feature grid. The *AlignedAnchor3DRangeGenerator* add + 1 when using the feature map sizes to obtain the corners of the voxel grid. Then it shifts the coordinates to the center of voxel grid and use the left up corner to distribute anchors.

Parameters **anchor_corner** (*bool, optional*) – Whether to align with the corner of the voxel grid. By default it is False and the anchor's center will be the same as the corresponding voxel's center, which is also the center of the corresponding greature grid. Defaults to False.

anchors_single_range(*feature_size*, *anchor_range*, *scale*, *sizes=[[3.9, 1.6, 1.56]]*, *rotations=[0, 1.5707963]*, *device='cuda'*)

Generate anchors in a single range.

Parameters

- **feature_size** (*list[float] | tuple[float]*) – Feature map size. It is either a list of a tuple of [D, H, W](in order of z, y, and x).
- **anchor_range** (*torch.Tensor | list[float]*) – Range of anchors with shape [6]. The order is consistent with that of anchors, i.e., (x_min, y_min, z_min, x_max, y_max, z_max).
- **scale** (*float | int*) – The scale factor of anchors.
- **sizes** (*list[list] | np.ndarray | torch.Tensor, optional*) – Anchor size with shape [N, 3], in order of x, y, z. Defaults to [[3.9, 1.6, 1.56]].
- **rotations** (*list[float] | np.ndarray | torch.Tensor, optional*) – Rotations of anchors in a single feature grid. Defaults to [0, 1.5707963].
- **device** (*str, optional*) – Devices that the anchors will be put on. Defaults to 'cuda'.

Returns

Anchors with shape [*feature_size, num_sizes, num_rots, 7].

Return type torch.Tensor

class mmdet3d.core.anchor.**AlignedAnchor3DRangeGeneratorPerCls**(***kwargs*)

3D Anchor Generator by range for per class.

This anchor generator generates anchors by the given range for per class. Note that feature maps of different classes may be different.

Parameters **kwargs** (*dict*) – Arguments are the same as those in *AlignedAnchor3DRangeGenerator*.

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

Generate grid anchors in multiple feature levels.

Parameters

- **featmap_sizes** (*list[tuple]*) – List of feature map sizes for different classes in a single feature level.
- **device** (*str, optional*) – Device where the anchors will be put on. Defaults to 'cuda'.

Returns

Anchors in multiple feature levels. Note that in this anchor generator, we currently only support single feature level. The sizes of each tensor should be [num_sizes/ranges*num_rots*featmap_size, box_code_size].

Return type list[list[torch.Tensor]]

multi_cls_grid_anchors(*featmap_sizes*, *scale*, *device='cuda'*)

Generate grid anchors of a single level feature map for multi-class with different feature map sizes.

This function is usually called by method `self.grid_anchors`.

Parameters

- **featmap_sizes** (*list[tuple]*) – List of feature map sizes for different classes in a single feature level.
- **scale** (*float*) – Scale factor of the anchors in the current level.
- **device** (*str, optional*) – Device the tensor will be put on. Defaults to 'cuda'.

Returns Anchors in the overall feature map.

Return type torch.Tensor

class mmdet3d.core.anchor.**Anchor3DRangeGenerator**(*ranges*, *sizes=[[3.9, 1.6, 1.56]]*, *scales=[1]*, *rotations=[0, 1.5707963]*, *custom_values=()*, *reshape_out=True*, *size_per_range=True*)

3D Anchor Generator by range.

This anchor generator generates anchors by the given range in different feature levels. Due the convention in 3D detection, different anchor sizes are related to different ranges for different categories. However we find this setting does not effect the performance much in some datasets, e.g., nuScenes.

Parameters

- **ranges** (*list[list[float]]*) – Ranges of different anchors. The ranges are the same across different feature levels. But may vary for different anchor sizes if size_per_range is True.

- **sizes** (*list[list[float]], optional*) – 3D sizes of anchors. Defaults to [[3.9, 1.6, 1.56]].
- **scales** (*list[int], optional*) – Scales of anchors in different feature levels. Defaults to [1].
- **rotations** (*list[float], optional*) – Rotations of anchors in a feature grid. Defaults to [0, 1.5707963].
- **custom_values** (*tuple[float], optional*) – Customized values of that anchor. For example, in nuScenes the anchors have velocities. Defaults to ().
- **reshape_out** (*bool, optional*) – Whether to reshape the output into (N x 4). Defaults to True.
- **size_per_range** (*bool, optional*) – Whether to use separate ranges for different sizes. If size_per_range is True, the ranges should have the same length as the sizes, if not, it will be duplicated. Defaults to True.

anchors_single_range(*feature_size*, *anchor_range*, *scale=1*, *sizes=[[3.9, 1.6, 1.56]]*, *rotations=[0, 1.5707963]*, *device='cuda'*)

Generate anchors in a single range.

Parameters

- **feature_size** (*list[float] | tuple[float]*) – Feature map size. It is either a list of a tuple of [D, H, W](in order of z, y, and x).
- **anchor_range** (*torch.Tensor | list[float]*) – Range of anchors with shape [6]. The order is consistent with that of anchors, i.e., (x_min, y_min, z_min, x_max, y_max, z_max).
- **scale** (*float | int, optional*) – The scale factor of anchors. Defaults to 1.
- **sizes** (*list[list] | np.ndarray | torch.Tensor, optional*) – Anchor size with shape [N, 3], in order of x, y, z. Defaults to [[3.9, 1.6, 1.56]].
- **rotations** (*list[float] | np.ndarray | torch.Tensor, optional*) – Rotations of anchors in a single feature grid. Defaults to [0, 1.5707963].
- **device** (*str*) – Devices that the anchors will be put on. Defaults to 'cuda'.

Returns

Anchors with shape [*feature_size, num_sizes, num_rots, 7].

Return type torch.Tensor

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

Generate grid anchors in multiple feature levels.

Parameters

- **featmap_sizes** (*list[tuple]*) – List of feature map sizes in multiple feature levels.
- **device** (*str, optional*) – Device where the anchors will be put on. Defaults to 'cuda'.

Returns

Anchors in multiple feature levels. The sizes of each tensor should be [N, 4], where N = width * height * num_base_anchors, width and height are the sizes of the corresponding feature level, num_base_anchors is the number of anchors for that level.

Return type list[torch.Tensor]

property num_base_anchors
Total number of base anchors in a feature grid.

Type list[int]

property num_levels
Number of feature levels that the generator is applied to.

Type int

single_level_grid_anchors(*featmap_size*, *scale*, *device='cuda'*)
Generate grid anchors of a single level feature map.

This function is usually called by method `self.grid_anchors`.

Parameters

- **featmap_size** (*tuple[int]*) – Size of the feature map.
- **scale** (*float*) – Scale factor of the anchors in the current level.
- **device** (*str, optional*) – Device the tensor will be put on. Defaults to 'cuda'.

Returns Anchors in the overall feature map.

Return type torch.Tensor

45.2 bbox

class mmdet3d.core.bbox.**AssignResult**(*num_gts*, *gt_inds*, *max_overlaps*, *labels=None*)
Stores assignments between predicted and truth boxes.

num_gts
the number of truth boxes considered when computing this assignment

Type int

gt_inds
for each predicted box indicates the 1-based index of the assigned truth box. 0 means unassigned and -1 means ignore.

Type LongTensor

max_overlaps
the iou between the predicted box and its assigned truth box.

Type FloatTensor

labels
If specified, for each predicted box indicates the category label of the assigned truth box.

Type None | LongTensor

Example

```
>>> # An assign result between 4 predicted boxes and 9 true boxes
>>> # where only two boxes were assigned.
>>> num_gts = 9
>>> max_overlaps = torch.LongTensor([0, .5, .9, 0])
>>> gt_inds = torch.LongTensor([-1, 1, 2, 0])
>>> labels = torch.LongTensor([0, 3, 4, 0])
>>> self = AssignResult(num_gts, gt_inds, max_overlaps, labels)
>>> print(str(self))  # xdoctest: +IGNORE_WANT
<AssignResult(num_gts=9, gt_inds.shape=(4,), max_overlaps.shape=(4,),
              labels.shape=(4,))>
>>> # Force addition of gt labels (when adding gt as proposals)
>>> new_labels = torch.LongTensor([3, 4, 5])
>>> self.add_gt_(new_labels)
>>> print(str(self))  # xdoctest: +IGNORE_WANT
<AssignResult(num_gts=9, gt_inds.shape=(7,), max_overlaps.shape=(7,),
              labels.shape=(7,))>
```

add_gt_(*gt_labels*)
Add ground truth as assigned results.

Parameters **gt_labels** (*torch.Tensor*) – Labels of gt boxes

get_extra_property(*key*)
Get user-defined property.

property info
a dictionary of info about the object

Type dict

property num_preds
the number of predictions in this assignment

Type int

classmethod random(***kwargs*)
Create random AssignResult for tests or debugging.

Parameters

- **num_preds** – number of predicted boxes
- **num_gts** – number of true boxes
- **p_ignore** (*float*) – probability of a predicted box assigned to an ignored truth
- **p_assigned** (*float*) – probability of a predicted box not being assigned
- **p_use_label** (*float | bool*) – with labels or not
- **rng** (*None | int | numpy.random.RandomState*) – seed or state

Returns Randomly generated assign results.

Return type *AssignResult*

Example

```
>>> from mmdet.core.bbox.assigners.assign_result import *  # NOQA
>>> self = AssignResult.random()
>>> print(self.info)
```

set_extra_property(*key*, *value*)
Set user-defined new property.

class mmdet3d.core.bbox.**AxisAlignedBboxOverlaps3D**
Axis-aligned 3D Overlaps (IoU) Calculator.

class mmdet3d.core.bbox.**BaseAssigner**
Base assigner that assigns boxes to ground truth boxes.

abstract assign(*bboxes*, *gt_bboxes*, *gt_bboxes_ignore=None*, *gt_labels=None*)
Assign boxes to either a ground truth boxes or a negative boxes.

class mmdet3d.core.bbox.**BaseInstance3DBoxes**(*tensor*, *box_dim=7*, *with_yaw=True*, *origin=(0.5, 0.5, 0)*)
Base class for 3D Boxes.

Note: The box is bottom centered, i.e. the relative position of origin in the box is (0.5, 0.5, 0).

Parameters

- **tensor** (*torch.Tensor | np.ndarray | list*) – a N x box_dim matrix.
- **box_dim** (*int*) – Number of the dimension of a box. Each row is (x, y, z, x_size, y_size, z_size, yaw). Defaults to 7.
- **with_yaw** (*bool*) – Whether the box is with yaw rotation. If False, the value of yaw will be set to 0 as minmax boxes. Defaults to True.
- **origin** (*tuple[float], optional*) – Relative position of the box origin. Defaults to (0.5, 0.5, 0). This will guide the box be converted to (0.5, 0.5, 0) mode.

tensor
Float matrix of N x box_dim.

Type torch.Tensor

box_dim
Integer indicating the dimension of a box. Each row is (x, y, z, x_size, y_size, z_size, yaw, ...).

Type int

with_yaw
If True, the value of yaw will be set to 0 as minmax boxes.

Type bool

property bev
2D BEV box of each box with rotation in XYWHR format, in shape (N, 5).

Type torch.Tensor

property bottom_center
A tensor with center of each box in shape (N, 3).

Type torch.Tensor

property bottom_height
torch.Tensor: A vector with bottom's height of each box in shape (N,).

classmethod cat(*boxes_list*)
Concatenate a list of Boxes into a single Boxes.

Parameters **boxes_list** (list[*BaseInstance3DBoxes*]) – List of boxes.

Returns The concatenated Boxes.

Return type *BaseInstance3DBoxes*

property center
Calculate the center of all the boxes.

Note: In MMDetection3D's convention, the bottom center is usually taken as the default center.

The relative position of the centers in different kinds of boxes are different, e.g., the relative center of a boxes is (0.5, 1.0, 0.5) in camera and (0.5, 0.5, 0) in lidar. It is recommended to use `bottom_center` or `gravity_center` for clearer usage.

Returns A tensor with center of each box in shape (N, 3).

Return type torch.Tensor

clone()
Clone the Boxes.

Returns

Box object with the same properties as self.

Return type *BaseInstance3DBoxes*

abstract convert_to(*dst*, *rt_mat=None*)
Convert self to `dst` mode.

Parameters

- **dst** (*Box3DMode*) – The target Box mode.
- **rt_mat** (*np.ndarray | torch.Tensor, optional*) – The rotation and translation matrix between different coordinates. Defaults to None. The conversion from *src* coordinates to *dst* coordinates usually comes along the change of sensors, e.g., from camera to LiDAR. This requires a transformation matrix.

Returns

The converted box of the same type in the *dst* mode.

Return type *BaseInstance3DBoxes*

property corners
torch.Tensor: a tensor with 8 corners of each box in shape (N, 8, 3).

property device
The device of the boxes are on.

Type str

property dims
Size dimensions of each box in shape (N, 3).

Type torch.Tensor

abstract flip(*bev_direction='horizontal'*)
Flip the boxes in BEV along given BEV direction.

Parameters **bev_direction** (*str, optional*) – Direction by which to flip. Can be chosen from 'horizontal' and 'vertical'. Defaults to 'horizontal'.

property gravity_center
A tensor with center of each box in shape (N, 3).

Type torch.Tensor

property height
A vector with height of each box in shape (N,).

Type torch.Tensor

classmethod height_overlaps(*boxes1*, *boxes2*, *mode='iou'*)
Calculate height overlaps of two boxes.

Note: This function calculates the height overlaps between boxes1 and boxes2, boxes1 and boxes2 should be in the same type.

Parameters

- **boxes1** (*BaseInstance3DBoxes*) – Boxes 1 contain N boxes.
- **boxes2** (*BaseInstance3DBoxes*) – Boxes 2 contain M boxes.
- **mode** (*str, optional*) – Mode of IoU calculation. Defaults to 'iou'.

Returns Calculated iou of boxes.

Return type torch.Tensor

in_range_3d(*box_range*)
Check whether the boxes are in the given range.

Parameters **box_range** (*list | torch.Tensor*) – The range of box (x_min, y_min, z_min, x_max, y_max, z_max)

Note: In the original implementation of SECOND, checking whether a box in the range checks whether the points are in a convex polygon, we try to reduce the burden for simpler cases.

Returns

A binary vector indicating whether each box is inside the reference range.

Return type torch.Tensor

in_range_bev(*box_range*)
Check whether the boxes are in the given range.

Parameters **box_range** (*list | torch.Tensor*) – the range of box (x_min, y_min, x_max, y_max)

Note: The original implementation of SECOND checks whether boxes in a range by checking whether the points are in a convex polygon, we reduce the burden for simpler cases.

Returns Whether each box is inside the reference range.

Return type torch.Tensor

`limit_yaw`(*offset=0.5*, *period=3.141592653589793*)
Limit the yaw to a given period and offset.

Parameters

- **offset** (*float, optional*) – The offset of the yaw. Defaults to 0.5.
- **period** (*float, optional*) – The expected period. Defaults to np.pi.

`property nearest_bev`
A tensor of 2D BEV box of each box without rotation.

Type torch.Tensor

`new_box`(*data*)
Create a new box object with data.

The new box and its tensor has the similar properties as self and self.tensor, respectively.

Parameters **data** (*torch.Tensor | numpy.array | list*) – Data to be copied.

Returns

A new bbox object with data, the object's other properties are similar to `self`.

Return type *BaseInstance3DBoxes*

`nonempty`(*threshold=0.0*)
Find boxes that are non-empty.

A box is considered empty, if either of its side is no larger than threshold.

Parameters **threshold** (*float, optional*) – The threshold of minimal sizes. Defaults to 0.0.

Returns

A binary vector which represents whether each box is empty (False) or non-empty (True).

Return type torch.Tensor

`classmethod overlaps`(*boxes1*, *boxes2*, *mode='iou'*)
Calculate 3D overlaps of two boxes.

Note: This function calculates the overlaps between `boxes1` and `boxes2`, `boxes1` and `boxes2` should be in the same type.

Parameters

- **boxes1** (*BaseInstance3DBoxes*) – Boxes 1 contain N boxes.
- **boxes2** (*BaseInstance3DBoxes*) – Boxes 2 contain M boxes.

- **mode** (*str, optional*) – Mode of iou calculation. Defaults to 'iou'.

Returns Calculated 3D overlaps of the boxes.

Return type torch.Tensor

points_in_boxes_all(*points*, *boxes_override=None*)
Find all boxes in which each point is.

Parameters

- **points** (*torch.Tensor*) – Points in shape (1, M, 3) or (M, 3), 3 dimensions are (x, y, z) in LiDAR or depth coordinate.
- **boxes_override** (*torch.Tensor, optional*) – Boxes to override *self.tensor*. Defaults to None.

Returns

A tensor indicating whether a point is in a box, in shape (M, T). T is the number of boxes. Denote this tensor as A, if the m^th point is in the t^th box, then *A[m, t] == 1*, elsewise *A[m, t] == 0*.

Return type torch.Tensor

points_in_boxes_part(*points*, *boxes_override=None*)
Find the box in which each point is.

Parameters

- **points** (*torch.Tensor*) – Points in shape (1, M, 3) or (M, 3), 3 dimensions are (x, y, z) in LiDAR or depth coordinate.
- **boxes_override** (*torch.Tensor, optional*) – Boxes to override *self.tensor*. Defaults to None.

Returns

The index of the first box that each point is in, in shape (M,). Default value is -1 (if the point is not enclosed by any box).

Return type torch.Tensor

Note: If a point is enclosed by multiple boxes, the index of the first box will be returned.

---

abstract rotate(*angle*, *points=None*)
Rotate boxes with points (optional) with the given angle or rotation matrix.

Parameters

- **angle** (*float | torch.Tensor | np.ndarray*) – Rotation angle or rotation matrix.
- **(torch.Tensor | numpy.ndarray |** (*points*) – BasePoints, optional): Points to rotate. Defaults to None.

scale(*scale_factor*)
Scale the box with horizontal and vertical scaling factors.

Parameters **scale_factors** (*float*) – Scale factors to scale the boxes.

to(*device*)
Convert current boxes to a specific device.

Parameters **device** (str | torch.device) – The name of the device.

Returns

A new boxes object on the specific device.

Return type *BaseInstance3DBoxes*

property top_height
torch.Tensor: A vector with the top height of each box in shape (N,).

translate(*trans_vector*)
Translate boxes with the given translation vector.

Parameters **trans_vector** (*torch.Tensor*) – Translation vector of size (1, 3).

property volume
A vector with volume of each box.

Type torch.Tensor

property yaw
A vector with yaw of each box in shape (N,).

Type torch.Tensor

class mmdet3d.core.bbox.**BaseSampler**(*num*, *pos_fraction*, *neg_pos_ub=- 1*, *add_gt_as_proposals=True*, ***kwargs*)
Base class of samplers.

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** (*Tensor*) – Boxes to be sampled from.
- **gt_bboxes** (*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 mmdet3d.core.bbox.**BboxOverlaps3D**(*coordinate*)
3D IoU Calculator.

Parameters **coordinate** (*str*) – The coordinate system, valid options are 'camera', 'lidar', and 'depth'.

class mmdet3d.core.bbox.**BboxOverlapsNearest3D**(*coordinate='lidar'*)
Nearest 3D IoU Calculator.

Note: This IoU calculator first finds the nearest 2D boxes in bird eye view (BEV), and then calculates the 2D IoU using `bbox_overlaps()`.

Parameters **coordinate** (*str*) – 'camera', 'lidar', or 'depth' coordinate system.

class mmdet3d.core.bbox.**Box3DMode**(*value*)
Enum of different ways to represent a box.

Coordinates in LiDAR:

```
            up z
               ^   x front
               |  /
               | /
left y <------ 0
```

The relative coordinate of bottom center in a LiDAR box is (0.5, 0.5, 0), and the yaw is around the z axis, thus the rotation axis=2.

Coordinates in camera:

```
        z front
       /
      /
     0 ------> x right
     |
     |
     v
down y
```

The relative coordinate of bottom center in a CAM box is [0.5, 1.0, 0.5], and the yaw is around the y axis, thus the rotation axis=1.

Coordinates in Depth mode:

```
up z
   ^   y front
   |  /
   | /
   0 ------> x right
```

The relative coordinate of bottom center in a DEPTH box is (0.5, 0.5, 0), and the yaw is around the z axis, thus the rotation axis=2.

static **convert**(*box*, *src*, *dst*, *rt_mat=None*, *with_yaw=True*)
Convert boxes from *src* mode to *dst* mode.

Parameters

- **(tuple | list | np.ndarray |** (*box*) – torch.Tensor | *BaseInstance3DBoxes*): Can be a k-tuple, k-list or an Nxk array/tensor, where k = 7.
- **src** (*Box3DMode*) – The src Box mode.
- **dst** (*Box3DMode*) – The target Box mode.
- **rt_mat** (*np.ndarray | torch.Tensor, optional*) – The rotation and translation matrix between different coordinates. Defaults to None. The conversion from *src* coordinates to *dst* coordinates usually comes along the change of sensors, e.g., from camera to LiDAR. This requires a transformation matrix.
- **with_yaw** (*bool, optional*) – If *box* is an instance of *BaseInstance3DBoxes*, whether or not it has a yaw angle. Defaults to True.

Returns

(tuple | list | np.ndarray | torch.Tensor | *BaseInstance3DBoxes*): The converted box of the same type.

class mmdet3d.core.bbox.**CameraInstance3DBoxes**(*tensor*, *box_dim=7*, *with_yaw=True*, *origin=(0.5, 1.0, 0.5)*)

3D boxes of instances in CAM coordinates.

Coordinates in camera:

```
        z front (yaw=-0.5*pi)
       /
      /
     0 ------> x right (yaw=0)
     |
     |
     v
down y
```

The relative coordinate of bottom center in a CAM box is (0.5, 1.0, 0.5), and the yaw is around the y axis, thus the rotation axis=1. The yaw is 0 at the positive direction of x axis, and decreases from the positive direction of x to the positive direction of z.

tensor
Float matrix in shape (N, box_dim).

Type torch.Tensor

box_dim
Integer indicating the dimension of a box Each row is (x, y, z, x_size, y_size, z_size, yaw, ...).

Type int

with_yaw
If True, the value of yaw will be set to 0 as axis-aligned boxes tightly enclosing the original boxes.

Type bool

property bev
2D BEV box of each box with rotation in XYWHR format, in shape (N, 5).

Type torch.Tensor

property bottom_height
torch.Tensor: A vector with bottom's height of each box in shape (N,).

convert_to(*dst*, *rt_mat=None*)
Convert self to `dst` mode.

Parameters

- **dst** (*Box3DMode*) – The target Box mode.
- **rt_mat** (*np.ndarray | torch.Tensor, optional*) – The rotation and translation matrix between different coordinates. Defaults to None. The conversion from `src` coordinates to `dst` coordinates usually comes along the change of sensors, e.g., from camera to LiDAR. This requires a transformation matrix.

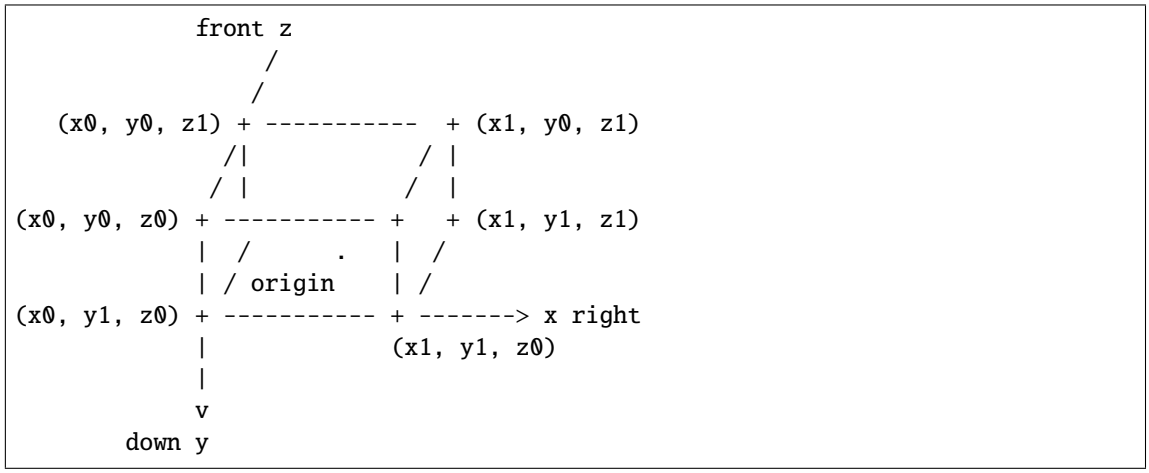
Returns The converted box of the same type in the `dst` mode.

Return type *BaseInstance3DBoxes*

property corners

Coordinates of corners of all the boxes in shape (N, 8, 3).

Convert the boxes to in clockwise order, in the form of (x0y0z0, x0y0z1, x0y1z1, x0y1z0, x1y0z0, x1y0z1, x1y1z1, x1y1z0)



Type torch.Tensor

flip(*bev_direction='horizontal'*, *points=None*)
Flip the boxes in BEV along given BEV direction.

In CAM coordinates, it flips the x (horizontal) or z (vertical) axis.

Parameters

- **bev_direction** (*str*) – Flip direction (horizontal or vertical).
- **points** (torch.Tensor | np.ndarray | `BasePoints`, optional) – Points to flip. Defaults to None.

Returns Flipped points.

Return type torch.Tensor, numpy.ndarray or None

property gravity_center
A tensor with center of each box in shape (N, 3).

Type torch.Tensor

property height
A vector with height of each box in shape (N,).

Type torch.Tensor

classmethod height_overlaps(*boxes1*, *boxes2*, *mode='iou'*)
Calculate height overlaps of two boxes.

This function calculates the height overlaps between `boxes1` and `boxes2`, where `boxes1` and `boxes2` should be in the same type.

Parameters

- **boxes1** (*CameraInstance3DBoxes*) – Boxes 1 contain N boxes.
- **boxes2** (*CameraInstance3DBoxes*) – Boxes 2 contain M boxes.
- **mode** (*str, optional*) – Mode of iou calculation. Defaults to 'iou'.

Returns Calculated iou of boxes' heights.

Return type torch.Tensor

property local_yaw
torch.Tensor: A vector with local yaw of each box in shape (N,). local_yaw equals to alpha in kitti, which is commonly used in monocular 3D object detection task, so only *CameraInstance3DBoxes* has the property.

points_in_boxes_all(*points*, *boxes_override=None*)
Find all boxes in which each point is.

Parameters

- **points** (*torch.Tensor*) – Points in shape (1, M, 3) or (M, 3), 3 dimensions are (x, y, z) in LiDAR or depth coordinate.
- **boxes_override** (*torch.Tensor, optional*) – Boxes to override `self.tensor `. Defaults to None.

Returns

The index of all boxes in which each point is, in shape (B, M, T).

Return type torch.Tensor

points_in_boxes_part(*points*, *boxes_override=None*)
Find the box in which each point is.

Parameters

- **points** (*torch.Tensor*) – Points in shape (1, M, 3) or (M, 3), 3 dimensions are (x, y, z) in LiDAR or depth coordinate.
- **boxes_override** (*torch.Tensor, optional*) – Boxes to override `self.tensor `. Defaults to None.

Returns

The index of the box in which each point is, in shape (M,). Default value is -1 (if the point is not enclosed by any box).

Return type torch.Tensor

rotate(*angle*, *points=None*)
Rotate boxes with points (optional) with the given angle or rotation matrix.

Parameters

- **angle** (*float | torch.Tensor | np.ndarray*) – Rotation angle or rotation matrix.
- **points** (torch.Tensor | np.ndarray | BasePoints, optional) – Points to rotate. Defaults to None.

Returns

When points is None, the function returns None, otherwise it returns the rotated points and the rotation matrix rot_mat_T.

Return type tuple or None

property top_height

torch.Tensor: A vector with the top height of each box in shape (N,).

class mmdet3d.core.bbox.**CombinedSampler**(*pos_sampler*, *neg_sampler*, ***kwargs*)

A sampler that combines positive sampler and negative sampler.

class mmdet3d.core.bbox.**Coord3DMode**(*value*)

Enum of different ways to represent a box and point cloud.

Coordinates in LiDAR:

```
            up z
               ^   x front
               |  /
               | /
left y <------ 0
```

The relative coordinate of bottom center in a LiDAR box is (0.5, 0.5, 0), and the yaw is around the z axis, thus the rotation axis=2.

Coordinates in camera:

```
        z front
       /
      /
     0 ------> x right
     |
     |
     v
down y
```

The relative coordinate of bottom center in a CAM box is [0.5, 1.0, 0.5], and the yaw is around the y axis, thus the rotation axis=1.

Coordinates in Depth mode:

```
up z
   ^   y front
   |  /
   | /
   0 ------> x right
```

The relative coordinate of bottom center in a DEPTH box is (0.5, 0.5, 0), and the yaw is around the z axis, thus the rotation axis=2.

static convert(*input*, *src*, *dst*, *rt_mat=None*, *with_yaw=True*, *is_point=True*)

Convert boxes or points from *src* mode to *dst* mode.

Parameters

- **(tuple | list | np.ndarray | torch.Tensor |** (*input*) – *BaseInstance3DBoxes* | BasePoints): Can be a k-tuple, k-list or an Nxk array/tensor, where k = 7.
- **src** (*Box3DMode* | *Coord3DMode*) – The source mode.
- **dst** (*Box3DMode* | *Coord3DMode*) – The target mode.
- **rt_mat** (*np.ndarray | torch.Tensor, optional*) – The rotation and translation matrix between different coordinates. Defaults to None. The conversion from *src* coordinates to *dst* coordinates usually comes along the change of sensors, e.g., from camera to LiDAR. This requires a transformation matrix.
- **with_yaw** (*bool*) – If *box* is an instance of *BaseInstance3DBoxes*, whether or not it has a yaw angle. Defaults to True.
- **is_point** (*bool*) – If *input* is neither an instance of *BaseInstance3DBoxes* nor an instance of BasePoints, whether or not it is point data. Defaults to True.

Returns

(tuple | list | np.ndarray | torch.Tensor | *BaseInstance3DBoxes* | BasePoints): The converted box of the same type.

static convert_box(*box*, *src*, *dst*, *rt_mat=None*, *with_yaw=True*)

Convert boxes from *src* mode to *dst* mode.

Parameters

- **(tuple | list | np.ndarray |** (*box*) – torch.Tensor | *BaseInstance3DBoxes*): Can be a k-tuple, k-list or an Nxk array/tensor, where k = 7.
- **src** (*Box3DMode*) – The src Box mode.
- **dst** (*Box3DMode*) – The target Box mode.
- **rt_mat** (*np.ndarray | torch.Tensor, optional*) – The rotation and translation matrix between different coordinates. Defaults to None. The conversion from *src* coordinates to *dst* coordinates usually comes along the change of sensors, e.g., from camera to LiDAR. This requires a transformation matrix.
- **with_yaw** (*bool*) – If *box* is an instance of *BaseInstance3DBoxes*, whether or not it has a yaw angle. Defaults to True.

Returns

(tuple | list | np.ndarray | torch.Tensor | *BaseInstance3DBoxes*): The converted box of the same type.

static convert_point(*point*, *src*, *dst*, *rt_mat=None*)

Convert points from *src* mode to *dst* mode.

Parameters

- **(tuple | list | np.ndarray |** (*point*) – torch.Tensor | BasePoints): Can be a k-tuple, k-list or an Nxk array/tensor.
- **src** (CoordMode) – The src Point mode.
- **dst** (CoordMode) – The target Point mode.
- **rt_mat** (*np.ndarray | torch.Tensor, optional*) – The rotation and translation matrix between different coordinates. Defaults to None. The conversion from *src* coordinates

to *dst* coordinates usually comes along the change of sensors, e.g., from camera to LiDAR. This requires a transformation matrix.

Returns The converted point of the same type.

Return type (tuple | list | np.ndarray | torch.Tensor | BasePoints)

class mmdet3d.core.bbox.**DeltaXYZWLHRBBoxCoder**(*code_size=7*)

Bbox Coder for 3D boxes.

Parameters **code_size** (*int*) – The dimension of boxes to be encoded.

static **decode**(*anchors*, *deltas*)

Apply transformation *deltas* (dx, dy, dz, dx_size, dy_size, dz_size, dr, dv*) to *boxes*.

Parameters

- **anchors** (*torch.Tensor*) – Parameters of anchors with shape (N, 7).
- **deltas** (*torch.Tensor*) – Encoded boxes with shape (N, 7+n) [x, y, z, x_size, y_size, z_size, r, velo*].

Returns Decoded boxes.

Return type torch.Tensor

static **encode**(*src_boxes*, *dst_boxes*)

Get box regression transformation deltas (dx, dy, dz, dx_size, dy_size, dz_size, dr, dv*) that can be used to transform the *src_boxes* into the *target_boxes*.

Parameters

- **src_boxes** (*torch.Tensor*) – source boxes, e.g., object proposals.
- **dst_boxes** (*torch.Tensor*) – target of the transformation, e.g., ground-truth boxes.

Returns Box transformation deltas.

Return type torch.Tensor

class mmdet3d.core.bbox.**DepthInstance3DBoxes**(*tensor*, *box_dim=7*, *with_yaw=True*, *origin=(0.5, 0.5, 0)*)

3D boxes of instances in Depth coordinates.

Coordinates in Depth:

```
up z    y front (yaw=-0.5*pi)
   ^   ^
   |  /
   | /
   0 ------> x right (yaw=0)
```

The relative coordinate of bottom center in a Depth box is (0.5, 0.5, 0), and the yaw is around the z axis, thus the rotation axis=2. The yaw is 0 at the positive direction of x axis, and decreases from the positive direction of x to the positive direction of y. Also note that rotation of DepthInstance3DBoxes is counterclockwise, which is reverse to the definition of the yaw angle (clockwise).

A refactor is ongoing to make the three coordinate systems easier to understand and convert between each other.

tensor

Float matrix of N x box_dim.

Type torch.Tensor

box_dim
Integer indicates the dimension of a box Each row is (x, y, z, x_size, y_size, z_size, yaw, ...).

Type int

with_yaw
If True, the value of yaw will be set to 0 as minmax boxes.

Type bool

convert_to(*dst*, *rt_mat=None*)
Convert self to `dst` mode.

Parameters

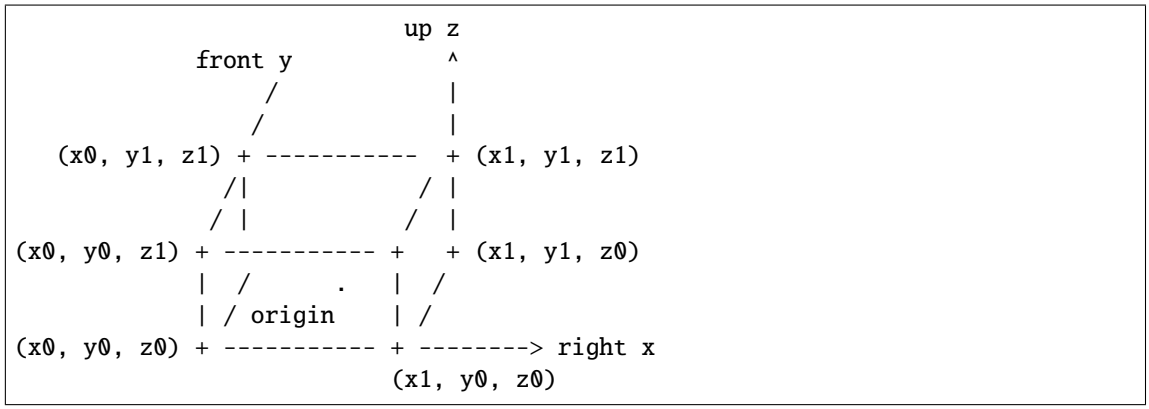
- **dst** (*Box3DMode*) – The target Box mode.
- **rt_mat** (*np.ndarray | torch.Tensor, optional*) – The rotation and translation matrix between different coordinates. Defaults to None. The conversion from `src` coordinates to `dst` coordinates usually comes along the change of sensors, e.g., from camera to LiDAR. This requires a transformation matrix.

Returns The converted box of the same type in the `dst` mode.

Return type *DepthInstance3DBoxes*

property corners
Coordinates of corners of all the boxes in shape (N, 8, 3).

Convert the boxes to corners in clockwise order, in form of `(x0y0z0, x0y0z1, x0y1z1, x0y1z0, x1y0z0, x1y0z1, x1y1z1, x1y1z0)`



Type torch.Tensor

enlarged_box(*extra_width*)
Enlarge the length, width and height boxes.

Parameters **extra_width** (*float | torch.Tensor*) – Extra width to enlarge the box.

Returns Enlarged boxes.

Return type *DepthInstance3DBoxes*

flip(*bev_direction='horizontal'*, *points=None*)
Flip the boxes in BEV along given BEV direction.

In Depth coordinates, it flips x (horizontal) or y (vertical) axis.

Parameters

- **bev_direction** (*str, optional*) – Flip direction (horizontal or vertical). Defaults to 'horizontal'.
- **points** (torch.Tensor | np.ndarray | BasePoints, optional) – Points to flip. Defaults to None.

Returns Flipped points.

Return type torch.Tensor, numpy.ndarray or None

get_surface_line_center()
Compute surface and line center of bounding boxes.

Returns Surface and line center of bounding boxes.

Return type torch.Tensor

property gravity_center
A tensor with center of each box in shape (N, 3).

Type torch.Tensor

rotate(*angle*, *points=None*)
Rotate boxes with points (optional) with the given angle or rotation matrix.

Parameters

- **angle** (*float | torch.Tensor | np.ndarray*) – Rotation angle or rotation matrix.
- **points** (torch.Tensor | np.ndarray | BasePoints, optional) – Points to rotate. Defaults to None.

Returns

When points is None, the function returns None, otherwise it returns the rotated points and the rotation matrix rot_mat_T.

Return type tuple or None

class mmdet3d.core.bbox.**InstanceBalancedPosSampler**(*num*, *pos_fraction*, *neg_pos_ub=- 1*, *add_gt_as_proposals=True*, ***kwargs*)
Instance balanced sampler that samples equal number of positive samples for each instance.

class mmdet3d.core.bbox.**IoUBalancedNegSampler**(*num*, *pos_fraction*, *floor_thr=- 1*, *floor_fraction=0*, *num_bins=3*, ***kwargs*)
IoU Balanced Sampling.

arXiv: https://arxiv.org/pdf/1904.02701.pdf (CVPR 2019)

Sampling proposals according to their IoU. *floor_fraction* of needed RoIs are sampled from proposals whose IoU are lower than *floor_thr* randomly. The others are sampled from proposals whose IoU are higher than *floor_thr*. These proposals are sampled from some bins evenly, which are split by *num_bins* via IoU evenly.

Parameters

- **num** (*int*) – number of proposals.
- **pos_fraction** (*float*) – fraction of positive proposals.
- **floor_thr** (*float*) – threshold (minimum) IoU for IoU balanced sampling, set to -1 if all using IoU balanced sampling.
- **floor_fraction** (*float*) – sampling fraction of proposals under floor_thr.
- **num_bins** (*int*) – number of bins in IoU balanced sampling.

sample_via_interval(*max_overlaps*, *full_set*, *num_expected*)
Sample according to the iou interval.

Parameters

- **max_overlaps** (*torch.Tensor*) – IoU between bounding boxes and ground truth boxes.
- **full_set** (*set(int)*) – A full set of indices of boxes
- **num_expected** (*int*) – Number of expected samples

Returns Indices of samples

Return type np.ndarray

class mmdet3d.core.bbox.**LiDARInstance3DBoxes**(*tensor*, *box_dim=7*, *with_yaw=True*, *origin=(0.5, 0.5, 0)*)
3D boxes of instances in LIDAR coordinates.

Coordinates in LiDAR:

```
                        up z    x front (yaw=0)
                           ^   ^
                           |  /
                           | /
(yaw=0.5*pi) left y <------ 0
```

The relative coordinate of bottom center in a LiDAR box is (0.5, 0.5, 0), and the yaw is around the z axis, thus the rotation axis=2. The yaw is 0 at the positive direction of x axis, and increases from the positive direction of x to the positive direction of y.

A refactor is ongoing to make the three coordinate systems easier to understand and convert between each other.

tensor
Float matrix of N x box_dim.

Type torch.Tensor

box_dim
Integer indicating the dimension of a box. Each row is (x, y, z, x_size, y_size, z_size, yaw, ...).

Type int

with_yaw
If True, the value of yaw will be set to 0 as minmax boxes.

Type bool

convert_to(*dst*, *rt_mat=None*)
Convert self to `dst` mode.

Parameters

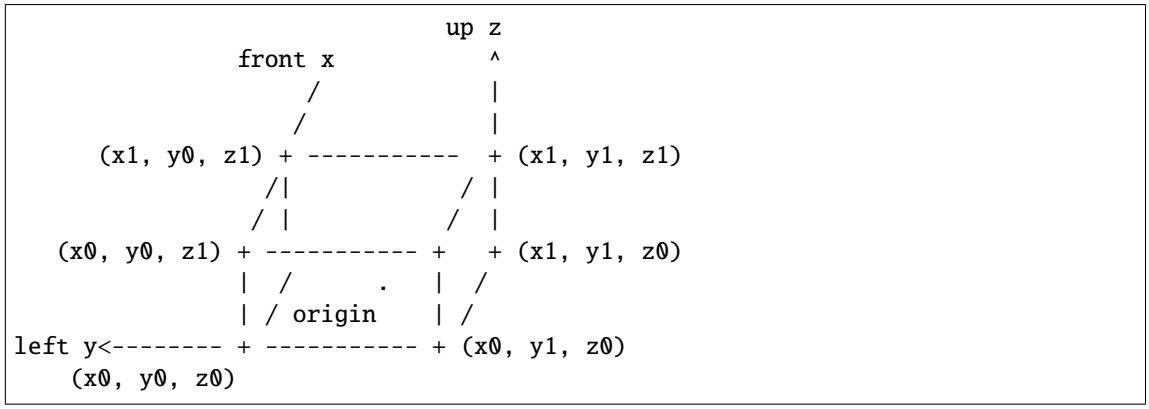
- **dst** (*Box3DMode*) – the target Box mode
- **rt_mat** (*np.ndarray | torch.Tensor, optional*) – The rotation and translation matrix between different coordinates. Defaults to None. The conversion from `src` coordinates to `dst` coordinates usually comes along the change of sensors, e.g., from camera to LiDAR. This requires a transformation matrix.

Returns The converted box of the same type in the `dst` mode.

Return type *BaseInstance3DBoxes*

property corners
Coordinates of corners of all the boxes in shape (N, 8, 3).

Convert the boxes to corners in clockwise order, in form of `(x0y0z0, x0y0z1, x0y1z1, x0y1z0, x1y0z0, x1y0z1, x1y1z1, x1y1z0)`



Type torch.Tensor

enlarged_box(*extra_width*)
Enlarge the length, width and height boxes.

Parameters **extra_width** (*float | torch.Tensor*) – Extra width to enlarge the box.

Returns Enlarged boxes.

Return type *LiDARInstance3DBoxes*

flip(*bev_direction='horizontal'*, *points=None*)
Flip the boxes in BEV along given BEV direction.

In LIDAR coordinates, it flips the y (horizontal) or x (vertical) axis.

Parameters

- **bev_direction** (*str*) – Flip direction (horizontal or vertical).
- **points** (torch.Tensor | np.ndarray | BasePoints, optional) – Points to flip. Defaults to None.

Returns Flipped points.

Return type torch.Tensor, numpy.ndarray or None

property gravity_center
A tensor with center of each box in shape (N, 3).

Type torch.Tensor

rotate(*angle*, *points=None*)
Rotate boxes with points (optional) with the given angle or rotation matrix.

Parameters

- **angles** (*float | torch.Tensor | np.ndarray*) – Rotation angle or rotation matrix.
- **points** (torch.Tensor | np.ndarray | BasePoints, optional) – Points to rotate. Defaults to None.

Returns

When points is None, the function returns None, otherwise it returns the rotated points and the rotation matrix rot_mat_T.

Return type tuple or None

class mmdet3d.core.bbox.**MaxIoUAssigner**(*pos_iou_thr*, *neg_iou_thr*, *min_pos_iou=0.0*, *gt_max_assign_all=True*, *ignore_iof_thr=-1*, *ignore_wrt_candidates=True*, *match_low_quality=True*, *gpu_assign_thr=-1*, *iou_calculator={'type': 'BboxOverlaps2D'}*)

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). *min_pos_iou* is set to avoid assigning bboxes that have extremely small iou with GT as positive samples. It brings about 0.3 mAP improvements in 1x schedule but does not affect the performance of 3x schedule. More comparisons can be found in PR #7464.
- **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.
- **match_low_quality** (*bool*) – Whether to allow low quality matches. This is usually allowed for RPN and single stage detectors, but not allowed in the second stage. Details are demonstrated in Step 4.
- **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(*bboxes*, *gt_bboxes*, *gt_bboxes_ignore=None*, *gt_labels=None*)

Assign gt to bboxes.

This method assign a gt bbox to every bbox (proposal/anchor), each bbox will be assigned with -1, or a semi-positive number. -1 means negative sample, semi-positive number is the index (0-based) of assigned gt. The assignment is done in following steps, the order matters.

1. assign every bbox to the background
2. assign proposals whose iou with all gts < neg_iou_thr to 0
3. for each bbox, if the iou with its nearest gt >= pos_iou_thr, assign it to that bbox
4. for each gt bbox, assign its nearest proposals (may be more than one) to itself

Parameters

- **bboxes** (*Tensor*) – Bounding boxes to be assigned, shape(n, 4).
- **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*

Example

```
>>> self = MaxIoUAssigner(0.5, 0.5)
>>> bboxes = torch.Tensor([[0, 0, 10, 10], [10, 10, 20, 20]])
>>> gt_bboxes = torch.Tensor([[0, 0, 10, 9]])
>>> assign_result = self.assign(bboxes, gt_bboxes)
>>> expected_gt_inds = torch.LongTensor([1, 0])
>>> assert torch.all(assign_result.gt_inds == expected_gt_inds)
```

assign_wrt_overlaps(*overlaps*, *gt_labels=None*)
Assign w.r.t. the overlaps of bboxes with gts.

Parameters

- **overlaps** (*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*

class mmdet3d.core.bbox.**PseudoSampler**(***kwargs*)
A pseudo sampler that does not do sampling actually.

sample(*assign_result*, *bboxes*, *gt_bboxes*, **args*, ***kwargs*)
Directly returns the positive and negative indices of samples.

Parameters

- **assign_result** (*AssignResult*) – Assigned results
- **bboxes** (*torch.Tensor*) – Bounding boxes
- **gt_bboxes** (*torch.Tensor*) – Ground truth boxes

Returns sampler results

Return type *SamplingResult*

class mmdet3d.core.bbox.**RandomSampler**(*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

class mmdet3d.core.bbox.**SamplingResult**(*pos_inds*, *neg_inds*, *bboxes*, *gt_bboxes*, *assign_result*, *gt_flags*)

Bbox sampling result.

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> from mmdet.core.bbox.samplers.sampling_result import *  # NOQA
>>> self = SamplingResult.random(rng=10)
>>> print(f'self = {self}')
self = <SamplingResult({
    'neg_bboxes': torch.Size([12, 4]),
    'neg_inds': tensor([ 0,  1,  2,  4,  5,  6,  7,  8,  9, 10, 11, 12]),
    'num_gts': 4,
    'pos_assigned_gt_inds': tensor([], dtype=torch.int64),
    'pos_bboxes': torch.Size([0, 4]),
    'pos_inds': tensor([], dtype=torch.int64),
    'pos_is_gt': tensor([], dtype=torch.uint8)
})>
```

property bboxes

concatenated positive and negative boxes

Type torch.Tensor

property info

Returns a dictionary of info about the object.

classmethod random(*rng=None*, ***kwargs*)

Parameters

- **rng** (*None | int | numpy.random.RandomState*) – seed or state.
- **kwargs** (*keyword arguments*) –
  - num_preds: number of predicted boxes
  - num_gts: number of true boxes

- p_ignore (float): probability of a predicted box assigned to an ignored truth.
- p_assigned (float): probability of a predicted box not being assigned.
- p_use_label (float | bool): with labels or not.

Returns Randomly generated sampling result.

Return type *SamplingResult*

Example

```
>>> from mmdet.core.bbox.samplers.sampling_result import *  # NOQA
>>> self = SamplingResult.random()
>>> print(self.__dict__)
```

to(*device*)

Change the device of the data inplace.

Example

```
>>> self = SamplingResult.random()
>>> print(f'self = {self.to(None)}')
>>> # xdoctest: +REQUIRES(--gpu)
>>> print(f'self = {self.to(0)}')
```

mmdet3d.core.bbox.**axis_aligned_bbox_overlaps_3d**(*bboxes1*, *bboxes2*, *mode='iou'*, *is_aligned=False*, *eps=1e-06*)

Calculate overlap between two set of axis aligned 3D bboxes. If `is_aligned` is `False`, then calculate the overlaps between each bbox of bboxes1 and bboxes2, otherwise the overlaps between each aligned pair of bboxes1 and bboxes2.

Parameters

- **bboxes1** (*Tensor*) – shape (B, m, 6) in <x1, y1, z1, x2, y2, z2> format or empty.
- **bboxes2** (*Tensor*) – shape (B, n, 6) in <x1, y1, z1, x2, y2, z2> format or empty. B indicates the batch dim, in shape (B1, B2, ..., Bn). If `is_aligned` is `True`, then m and n must be equal.
- **mode** (*str*) – "iou" (intersection over union) or "giou" (generalized intersection over union).
- **is_aligned** (*bool, optional*) – If True, then m and n must be equal. Defaults to False.
- **eps** (*float, optional*) – A value added to the denominator for numerical stability. Defaults to 1e-6.

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

Return type Tensor

Example

```
>>> bboxes1 = torch.FloatTensor([
>>>     [0, 0, 0, 10, 10, 10],
>>>     [10, 10, 10, 20, 20, 20],
>>>     [32, 32, 32, 38, 40, 42],
>>> ])
>>> bboxes2 = torch.FloatTensor([
>>>     [0, 0, 0, 10, 20, 20],
>>>     [0, 10, 10, 10, 19, 20],
>>>     [10, 10, 10, 20, 20, 20],
>>> ])
>>> overlaps = axis_aligned_bbox_overlaps_3d(bboxes1, bboxes2)
>>> assert overlaps.shape == (3, 3)
>>> overlaps = bbox_overlaps(bboxes1, bboxes2, is_aligned=True)
>>> assert overlaps.shape == (3, )
```

Example

```
>>> empty = torch.empty(0, 6)
>>> nonempty = torch.FloatTensor([[0, 0, 0, 10, 9, 10]])
>>> assert tuple(bbox_overlaps(empty, nonempty).shape) == (0, 1)
>>> assert tuple(bbox_overlaps(nonempty, empty).shape) == (1, 0)
>>> assert tuple(bbox_overlaps(empty, empty).shape) == (0, 0)
```

mmdet3d.core.bbox.**bbox3d2result**(*bboxes*, *scores*, *labels*, *attrs=None*)

Convert detection results to a list of numpy arrays.

Parameters

- **bboxes** (*torch.Tensor*) – Bounding boxes with shape (N, 5).
- **labels** (*torch.Tensor*) – Labels with shape (N,).
- **scores** (*torch.Tensor*) – Scores with shape (N,).
- **attrs** (*torch.Tensor, optional*) – Attributes with shape (N,). Defaults to None.

Returns

Bounding box results in cpu mode.

- boxes_3d (torch.Tensor): 3D boxes.
- scores (torch.Tensor): Prediction scores.
- labels_3d (torch.Tensor): Box labels.
- attrs_3d (torch.Tensor, optional): Box attributes.

Return type dict[str, torch.Tensor]

mmdet3d.core.bbox.**bbox3d2roi**(*bbox_list*)

Convert a list of bounding boxes to roi format.

Parameters **bbox_list** (*list[torch.Tensor]*) – A list of bounding boxes corresponding to a batch of images.

Returns

Region of interests in shape (n, c), where the channels are in order of [batch_ind, x, y ...].

Return type torch.Tensor

mmdet3d.core.bbox.**bbox3d_mapping_back**(*bboxes*, *scale_factor*, *flip_horizontal*, *flip_vertical*)
Map bboxes from testing scale to original image scale.

Parameters

- **bboxes** (*BaseInstance3DBoxes*) – Boxes to be mapped back.
- **scale_factor** (*float*) – Scale factor.
- **flip_horizontal** (*bool*) – Whether to flip horizontally.
- **flip_vertical** (*bool*) – Whether to flip vertically.

Returns Boxes mapped back.

Return type *BaseInstance3DBoxes*

mmdet3d.core.bbox.**bbox_overlaps_3d**(*bboxes1*, *bboxes2*, *mode='iou'*, *coordinate='camera'*)
Calculate 3D IoU using cuda implementation.

Note: This function calculates the IoU of 3D boxes based on their volumes. IoU calculator *BboxOverlaps3D* uses this function to calculate the actual IoUs of boxes.

Parameters

- **bboxes1** (*torch.Tensor*) – with shape (N, 7+C), (x, y, z, x_size, y_size, z_size, ry, v*).
- **bboxes2** (*torch.Tensor*) – with shape (M, 7+C), (x, y, z, x_size, y_size, z_size, ry, v*).
- **mode** (*str*) – "iou" (intersection over union) or iof (intersection over foreground).
- **coordinate** (*str*) – 'camera' or 'lidar' coordinate system.

Returns

Bbox overlaps results of bboxes1 and bboxes2 with shape (M, N) (aligned mode is not supported currently).

Return type torch.Tensor

mmdet3d.core.bbox.**bbox_overlaps_nearest_3d**(*bboxes1*, *bboxes2*, *mode='iou'*, *is_aligned=False*, *coordinate='lidar'*)
Calculate nearest 3D IoU.

Note: This function first finds the nearest 2D boxes in bird eye view (BEV), and then calculates the 2D IoU using `bbox_overlaps()`. This IoU calculator *BboxOverlapsNearest3D* uses this function to calculate IoUs of boxes.

If `is_aligned` is `False`, then it calculates the ious between each bbox of bboxes1 and bboxes2, otherwise the ious between each aligned pair of bboxes1 and bboxes2.

Parameters

- **bboxes1** (*torch.Tensor*) – with shape (N, 7+C), (x, y, z, x_size, y_size, z_size, ry, v*).
- **bboxes2** (*torch.Tensor*) – with shape (M, 7+C), (x, y, z, x_size, y_size, z_size, ry, v*).

- **mode** (*str*) – "iou" (intersection over union) or iof (intersection over foreground).
- **is_aligned** (*bool*) – Whether the calculation is aligned

Returns

If is_aligned is True, return ious between bboxes1 and bboxes2 with shape (M, N). If is_aligned is False, return shape is M.

Return type torch.Tensor

mmdet3d.core.bbox.**get_box_type**(*box_type*)

Get the type and mode of box structure.

Parameters **box_type** (*str*) – The type of box structure. The valid value are "LiDAR", "Camera", or "Depth".

Raises **ValueError** – A ValueError is raised when *box_type* does not belong to the three valid types.

Returns Box type and box mode.

Return type tuple

mmdet3d.core.bbox.**limit_period**(*val*, *offset=0.5*, *period=3.141592653589793*)

Limit the value into a period for periodic function.

Parameters

- **val** (*torch.Tensor | np.ndarray*) – The value to be converted.
- **offset** (*float, optional*) – Offset to set the value range. Defaults to 0.5.
- **period** (*[type], optional*) – Period of the value. Defaults to np.pi.

Returns

Value in the range of [-offset * period, (1-offset) * period]

Return type (torch.Tensor | np.ndarray)

mmdet3d.core.bbox.**mono_cam_box2vis**(*cam_box*)

This is a post-processing function on the bboxes from Mono-3D task. If we want to perform projection visualization, we need to:

1. rotate the box along x-axis for np.pi / 2 (roll)
2. change orientation from local yaw to global yaw
3. convert yaw by (np.pi / 2 - yaw)

After applying this function, we can project and draw it on 2D images.

Parameters **cam_box** (*CameraInstance3DBoxes*) – 3D bbox in camera coordinate system before conversion. Could be gt bbox loaded from dataset or network prediction output.

Returns Box after conversion.

Return type *CameraInstance3DBoxes*

mmdet3d.core.bbox.**points_cam2img**(*points_3d*, *proj_mat*, *with_depth=False*)

Project points in camera coordinates to image coordinates.

Parameters

- **points_3d** (*torch.Tensor | np.ndarray*) – Points in shape (N, 3)
- **proj_mat** (*torch.Tensor | np.ndarray*) – Transformation matrix between coordinates.

- **with_depth** (*bool, optional*) – Whether to keep depth in the output. Defaults to False.

Returns

Points in image coordinates, with shape [N, 2] if *with_depth=False*, else [N, 3].

Return type (torch.Tensor | np.ndarray)

mmdet3d.core.bbox.**points_img2cam**(*points*, *cam2img*)
Project points in image coordinates to camera coordinates.

Parameters

- **points** (*torch.Tensor*) – 2.5D points in 2D images, [N, 3], 3 corresponds with x, y in the image and depth.
- **cam2img** (*torch.Tensor*) – Camera intrinsic matrix. The shape can be [3, 3], [3, 4] or [4, 4].

Returns

points in 3D space. [N, 3], 3 corresponds with x, y, z in 3D space.

Return type torch.Tensor

mmdet3d.core.bbox.**xywhr2xyxyr**(*boxes_xywhr*)
Convert a rotated boxes in XYWHR format to XYXYR format.

Parameters **boxes_xywhr** (*torch.Tensor | np.ndarray*) – Rotated boxes in XYWHR format.

Returns Converted boxes in XYXYR format.

Return type (torch.Tensor | np.ndarray)

45.3 evaluation

mmdet3d.core.evaluation.**indoor_eval**(*gt_annos*, *dt_annos*, *metric*, *label2cat*, *logger=None*, *box_type_3d=None*, *box_mode_3d=None*)
Indoor Evaluation.

Evaluate the result of the detection.

Parameters

- **gt_annos** (*list[dict]*) – Ground truth annotations.
- **dt_annos** (*list[dict]*) – Detection annotations. the dict includes the following keys
  - labels_3d (torch.Tensor): Labels of boxes.
  - **boxes_3d (BaseInstance3DBoxes):** 3D bounding boxes in Depth coordinate.
  - scores_3d (torch.Tensor): Scores of boxes.
- **metric** (*list[float]*) – IoU thresholds for computing average precisions.
- **label2cat** (*dict*) – Map from label to category.
- **logger** (*logging.Logger | str, optional*) – The way to print the mAP summary. See *mmdet.utils.print_log()* for details. Default: None.

Returns Dict of results.

Return type dict[str, float]

mmdet3d.core.evaluation.**instance_seg_eval**(*gt_semantic_masks*, *gt_instance_masks*, *pred_instance_masks*, *pred_instance_labels*, *pred_instance_scores*, *valid_class_ids*, *class_labels*, *options=None*, *logger=None*)

Instance Segmentation Evaluation.

Evaluate the result of the instance segmentation.

Parameters

- **gt_semantic_masks** (*list[torch.Tensor]*) – Ground truth semantic masks.
- **gt_instance_masks** (*list[torch.Tensor]*) – Ground truth instance masks.
- **pred_instance_masks** (*list[torch.Tensor]*) – Predicted instance masks.
- **pred_instance_labels** (*list[torch.Tensor]*) – Predicted instance labels.
- **pred_instance_scores** (*list[torch.Tensor]*) – Predicted instance labels.
- **valid_class_ids** (*tuple[int]*) – Ids of valid categories.
- **class_labels** (*tuple[str]*) – Names of valid categories.
- **options** (*dict, optional*) – Additional options. Keys may contain: *overlaps*, *min_region_sizes*, *distance_threshes*, *distance_confs*. Default: None.
- **logger** (*logging.Logger | str, optional*) – The way to print the mAP summary. See *mmdet.utils.print_log()* for details. Default: None.

Returns Dict of results.

Return type dict[str, float]

mmdet3d.core.evaluation.**kitti_eval**(*gt_annos*, *dt_annos*, *current_classes*, *eval_types=['bbox', 'bev', '3d']*)

KITTI evaluation.

Parameters

- **gt_annos** (*list[dict]*) – Contain gt information of each sample.
- **dt_annos** (*list[dict]*) – Contain detected information of each sample.
- **current_classes** (*list[str]*) – Classes to evaluation.
- **eval_types** (*list[str], optional*) – Types to eval. Defaults to [‘bbox’, ‘bev’, ‘3d’].

Returns String and dict of evaluation results.

Return type tuple

mmdet3d.core.evaluation.**kitti_eval_coco_style**(*gt_annos*, *dt_annos*, *current_classes*)

coco style evaluation of kitti.

Parameters

- **gt_annos** (*list[dict]*) – Contain gt information of each sample.
- **dt_annos** (*list[dict]*) – Contain detected information of each sample.
- **current_classes** (*list[str]*) – Classes to evaluation.

Returns Evaluation results.

Return type string

mmdet3d.core.evaluation.**lyft_eval**(*lyft*, *data_root*, *res_path*, *eval_set*, *output_dir*, *logger=None*)

Evaluation API for Lyft dataset.

Parameters

- **lyft** (LyftDataset) – Lyft class in the sdk.
- **data_root** (*str*) – Root of data for reading splits.
- **res_path** (*str*) – Path of result json file recording detections.
- **eval_set** (*str*) – Name of the split for evaluation.
- **output_dir** (*str*) – Output directory for output json files.
- **logger** (*logging.Logger | str, optional*) – Logger used for printing related information during evaluation. Default: None.

Returns The evaluation results.

Return type dict[str, float]

mmdet3d.core.evaluation.**seg_eval**(*gt_labels*, *seg_preds*, *label2cat*, *ignore_index*, *logger=None*)
Semantic Segmentation Evaluation.

Evaluate the result of the Semantic Segmentation.

Parameters

- **gt_labels** (*list[torch.Tensor]*) – Ground truth labels.
- **seg_preds** (*list[torch.Tensor]*) – Predictions.
- **label2cat** (*dict*) – Map from label to category name.
- **ignore_index** (*int*) – Index that will be ignored in evaluation.
- **logger** (*logging.Logger | str, optional*) – The way to print the mAP summary. See *mmdet.utils.print_log()* for details. Default: None.

Returns Dict of results.

Return type dict[str, float]

45.4 visualizer

mmdet3d.core.visualizer.**show_multi_modality_result**(*img*, *gt_bboxes*, *pred_bboxes*, *proj_mat*, *out_dir*, *filename*, *box_mode='lidar'*, *img_metas=None*, *show=False*, *gt_bbox_color=(61, 102, 255)*, *pred_bbox_color=(241, 101, 72)*)
Convert multi-modality detection results into 2D results.

Project the predicted 3D bbox to 2D image plane and visualize them.

Parameters

- **img** (*np.ndarray*) – The numpy array of image in cv2 fashion.
- **gt_bboxes** (BaseInstance3DBoxes) – Ground truth boxes.
- **pred_bboxes** (BaseInstance3DBoxes) – Predicted boxes.
- **proj_mat** (*numpy.array, shape=[4, 4]*) – The projection matrix according to the camera intrinsic parameters.
- **out_dir** (*str*) – Path of output directory.
- **filename** (*str*) – Filename of the current frame.

- **box_mode** (*str, optional*) – Coordinate system the boxes are in. Should be one of 'depth', 'lidar' and 'camera'. Defaults to 'lidar'.
- **img_metas** (*dict, optional*) – Used in projecting depth bbox. Defaults to None.
- **show** (*bool, optional*) – Visualize the results online. Defaults to False.
- **gt_bbox_color** (*str or tuple(int), optional*) – Color of bbox lines. The tuple of color should be in BGR order. Default: (255, 102, 61).
- **pred_bbox_color** (*str or tuple(int), optional*) – Color of bbox lines. The tuple of color should be in BGR order. Default: (72, 101, 241).

mmdet3d.core.visualizer.**show_result**(*points*, *gt_bboxes*, *pred_bboxes*, *out_dir*, *filename*, *show=False*, *snapshot=False*, *pred_labels=None*)

Convert results into format that is directly readable for meshlab.

Parameters

- **points** (*np.ndarray*) – Points.
- **gt_bboxes** (*np.ndarray*) – Ground truth boxes.
- **pred_bboxes** (*np.ndarray*) – Predicted boxes.
- **out_dir** (*str*) – Path of output directory
- **filename** (*str*) – Filename of the current frame.
- **show** (*bool, optional*) – Visualize the results online. Defaults to False.
- **snapshot** (*bool, optional*) – Whether to save the online results. Defaults to False.
- **pred_labels** (*np.ndarray, optional*) – Predicted labels of boxes. Defaults to None.

mmdet3d.core.visualizer.**show_seg_result**(*points*, *gt_seg*, *pred_seg*, *out_dir*, *filename*, *palette*, *ignore_index=None*, *show=False*, *snapshot=False*)

Convert results into format that is directly readable for meshlab.

Parameters

- **points** (*np.ndarray*) – Points.
- **gt_seg** (*np.ndarray*) – Ground truth segmentation mask.
- **pred_seg** (*np.ndarray*) – Predicted segmentation mask.
- **out_dir** (*str*) – Path of output directory
- **filename** (*str*) – Filename of the current frame.
- **palette** (*np.ndarray*) – Mapping between class labels and colors.
- **ignore_index** (*int, optional*) – The label index to be ignored, e.g. unannotated points. Defaults to None.
- **show** (*bool, optional*) – Visualize the results online. Defaults to False.
- **snapshot** (*bool, optional*) – Whether to save the online results. Defaults to False.

45.5 voxel

class mmdet3d.core.voxel.**VoxelGenerator**(*voxel_size*, *point_cloud_range*, *max_num_points*, *max_voxels=20000*)

Voxel generator in numpy implementation.

Parameters

- **voxel_size** (*list[float]*) – Size of a single voxel
- **point_cloud_range** (*list[float]*) – Range of points
- **max_num_points** (*int*) – Maximum number of points in a single voxel
- **max_voxels** (*int, optional*) – Maximum number of voxels. Defaults to 20000.

generate(*points*)

Generate voxels given points.

property grid_size

The size of grids.

Type np.ndarray

property max_num_points_per_voxel

Maximum number of points per voxel.

Type int

property point_cloud_range

Range of point cloud.

Type list[float]

property voxel_size

Size of a single voxel.

Type list[float]

mmdet3d.core.voxel.**build_voxel_generator**(*cfg*, ***kwargs*)

Builder of voxel generator.

45.6 post_processing

mmdet3d.core.post_processing.**aligned_3d_nms**(*boxes*, *scores*, *classes*, *thresh*)

3D NMS for aligned boxes.

Parameters

- **boxes** (*torch.Tensor*) – Aligned box with shape [n, 6].
- **scores** (*torch.Tensor*) – Scores of each box.
- **classes** (*torch.Tensor*) – Class of each box.
- **thresh** (*float*) – IoU threshold for nms.

Returns Indices of selected boxes.

Return type torch.Tensor

mmdet3d.core.post_processing.**box3d_multiclass_nms**(*mlvl_bboxes*, *mlvl_bboxes_for_nms*, *mlvl_scores*, *score_thr*, *max_num*, *cfg*, *mlvl_dir_scores=None*, *mlvl_attr_scores=None*, *mlvl_bboxes2d=None*)

Multi-class NMS for 3D boxes. The IoU used for NMS is defined as the 2D IoU between BEV boxes.

Parameters

- **mlvl_bboxes** (*torch.Tensor*) – Multi-level boxes with shape (N, M). M is the dimensions of boxes.
- **mlvl_bboxes_for_nms** (*torch.Tensor*) – Multi-level boxes with shape (N, 5) ([x1, y1, x2, y2, ry]). N is the number of boxes. The coordinate system of the BEV boxes is counter-clockwise.
- **mlvl_scores** (*torch.Tensor*) – Multi-level boxes with shape (N, C + 1). N is the number of boxes. C is the number of classes.
- **score_thr** (*float*) – Score threshold to filter boxes with low confidence.
- **max_num** (*int*) – Maximum number of boxes will be kept.
- **cfg** (*dict*) – Configuration dict of NMS.
- **mlvl_dir_scores** (*torch.Tensor, optional*) – Multi-level scores of direction classifier. Defaults to None.
- **mlvl_attr_scores** (*torch.Tensor, optional*) – Multi-level scores of attribute classifier. Defaults to None.
- **mlvl_bboxes2d** (*torch.Tensor, optional*) – Multi-level 2D bounding boxes. Defaults to None.

Returns

Return results after nms, including 3D bounding boxes, scores, labels, direction scores, attribute scores (optional) and 2D bounding boxes (optional).

Return type tuple[torch.Tensor]

mmdet3d.core.post_processing.**circle_nms**(*dets*, *thresh*, *post_max_size=83*)

Circular NMS.

An object is only counted as positive if no other center with a higher confidence exists within a radius r using a bird-eye view distance metric.

Parameters

- **dets** (*torch.Tensor*) – Detection results with the shape of [N, 3].
- **thresh** (*float*) – Value of threshold.
- **post_max_size** (*int, optional*) – Max number of prediction to be kept. Defaults to 83.

Returns Indexes of the detections to be kept.

Return type torch.Tensor

mmdet3d.core.post_processing.**merge_aug_bboxes**(*aug_bboxes*, *aug_scores*, *img_metas*, *rcnn_test_cfg*)

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,).

- **rcnn_test_cfg** (*dict*) – rcnn test config.

Returns (bboxes, scores)

Return type tuple

mmdet3d.core.post_processing.**merge_aug_bboxes_3d**(*aug_results*, *img_metas*, *test_cfg*)
Merge augmented detection 3D bboxes and scores.

Parameters

- **aug_results** (*list[dict]*) – The dict of detection results. The dict contains the following keys
  - boxes_3d (BaseInstance3DBoxes): Detection bbox.
  - scores_3d (torch.Tensor): Detection scores.
  - labels_3d (torch.Tensor): Predicted box labels.
- **img_metas** (*list[dict]*) – Meta information of each sample.
- **test_cfg** (*dict*) – Test config.

Returns

Bounding boxes results in cpu mode, containing merged results.

- boxes_3d (BaseInstance3DBoxes): Merged detection bbox.
- scores_3d (torch.Tensor): Merged detection scores.
- labels_3d (torch.Tensor): Merged predicted box labels.

Return type dict

mmdet3d.core.post_processing.**merge_aug_masks**(*aug_masks*, *img_metas*, *rcnn_test_cfg*, *weights=None*)
Merge augmented mask prediction.

Parameters

- **aug_masks** (*list[ndarray]*) – shape (n, #class, h, w)
- **img_shapes** (*list[ndarray]*) – shape (3,).
- **rcnn_test_cfg** (*dict*) – rcnn test config.

Returns (bboxes, scores)

Return type tuple

mmdet3d.core.post_processing.**merge_aug_proposals**(*aug_proposals*, *img_metas*, *cfg*)
Merge augmented proposals (multiscale, flip, etc.)

Parameters

- **aug_proposals** (*list[Tensor]*) – proposals from different testing schemes, shape (n, 5). Note that they are not rescaled to the original image size.
- **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*.
- **cfg** (*dict*) – rpn test config.

Returns shape (n, 4), proposals corresponding to original image scale.

Return type Tensor

mmdet3d.core.post_processing.**merge_aug_scores**(*aug_scores*)
Merge augmented bbox scores.

mmdet3d.core.post_processing.**multiclass_nms**(*multi_bboxes*, *multi_scores*, *score_thr*, *nms_cfg*, *max_num=- 1*, *score_factors=None*, *return_inds=False*)
NMS for multi-class bboxes.

Parameters

- **multi_bboxes** (*Tensor*) – shape (n, #class*4) or (n, 4)
- **multi_scores** (*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_cfg** (*dict*) – a dict that contains the arguments of nms operations
- **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

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

Return type tuple

mmdet3d.core.post_processing.**nms_bev**(*boxes*, *scores*, *thresh*, *pre_max_size=None*, *post_max_size=None*)
NMS function GPU implementation (for BEV boxes). The overlap of two boxes for IoU calculation is defined as the exact overlapping area of the two boxes. In this function, one can also set `pre_max_size` and `post_max_size`.

Parameters

- **boxes** (*torch.Tensor*) – Input boxes with the shape of [N, 5] ([x1, y1, x2, y2, ry]).
- **scores** (*torch.Tensor*) – Scores of boxes with the shape of [N].
- **thresh** (*float*) – Overlap threshold of NMS.
- **pre_max_size** (*int, optional*) – Max size of boxes before NMS. Default: None.
- **post_max_size** (*int, optional*) – Max size of boxes after NMS. Default: None.

Returns Indexes after NMS.

Return type torch.Tensor

mmdet3d.core.post_processing.**nms_normal_bev**(*boxes*, *scores*, *thresh*)
Normal NMS function GPU implementation (for BEV boxes). The overlap of two boxes for IoU calculation is defined as the exact overlapping area of the two boxes WITH their yaw angle set to 0.

Parameters

- **boxes** (*torch.Tensor*) – Input boxes with shape (N, 5).
- **scores** (*torch.Tensor*) – Scores of predicted boxes with shape (N).
- **thresh** (*float*) – Overlap threshold of NMS.

Returns Remaining indices with scores in descending order.

Return type torch.Tensor

CHAPTER

FORTYSIX

MMDET3D.DATASETS

class mmdet3d.datasets.**AffineResize**(*img_scale*, *down_ratio*, *bbox_clip_border=True*)

Get the affine transform matrices to the target size.

Different from RandomAffine in MMDetection, this class can calculate the affine transform matrices while resizing the input image to a fixed size. The affine transform matrices include: 1) matrix transforming original image to the network input image size. 2) matrix transforming original image to the network output feature map size.

Parameters

- **img_scale** (*tuple*) – Images scales for resizing.
- **down_ratio** (*int*) – The down ratio of feature map. Actually the arg should be >= 1.
- **bbox_clip_border** (*bool, optional*) – Whether clip the objects outside the border of the image. Defaults to True.

class mmdet3d.datasets.**BackgroundPointsFilter**(*bbox_enlarge_range*)

Filter background points near the bounding box.

Parameters **bbox_enlarge_range** (*tuple[float], float*) – Bbox enlarge range.

class mmdet3d.datasets.**Custom3DDataset**(*data_root*, *ann_file*, *pipeline=None*, *classes=None*, *modality=None*, *box_type_3d='LiDAR'*, *filter_empty_gt=True*, *test_mode=False*, *file_client_args={'backend': 'disk'}*)

Customized 3D dataset.

This is the base dataset of SUNRGB-D, ScanNet, nuScenes, and KITTI dataset.

[

{'sample_idx':

'lidar_points': {'lidar_path': velodyne_path,

},

'annos': {'box_type_3d': (str) 'LiDAR/Camera/Depth'

'gt_bboxes_3d': <np.ndarray> (n, 7) 'gt_names': [list] ....

}

'calib': { .....} 'images': { .....}

}

]

Parameters

- **data_root** (*str*) – Path of dataset root.
- **ann_file** (*str*) – Path of annotation file.
- **pipeline** (*list[dict], optional*) – Pipeline used for data processing. Defaults to None.
- **classes** (*tuple[str], optional*) – Classes used in the dataset. Defaults to None.
- **modality** (*dict, optional*) – Modality to specify the sensor data used as input. Defaults to None.
- **box_type_3d** (*str, optional*) – Type of 3D box of this dataset. Based on the *box_type_3d*, the dataset will encapsulate the box to its original format then converted them to *box_type_3d*. Defaults to 'LiDAR'. Available options includes
  - 'LiDAR': Box in LiDAR coordinates.
  - 'Depth': Box in depth coordinates, usually for indoor dataset.
  - 'Camera': Box in camera coordinates.
- **filter_empty_gt** (*bool, optional*) – Whether to filter empty GT. Defaults to True.
- **test_mode** (*bool, optional*) – Whether the dataset is in test mode. Defaults to False.

evaluate(*results*, *metric=None*, *iou_thr=(0.25, 0.5)*, *logger=None*, *show=False*, *out_dir=None*, *pipeline=None*)

Evaluate.

Evaluation in indoor protocol.

Parameters

- **results** (*list[dict]*) – List of results.
- **metric** (*str | list[str], optional*) – Metrics to be evaluated. Defaults to None.
- **iou_thr** (*list[float]*) – AP IoU thresholds. Defaults to (0.25, 0.5).
- **logger** (*logging.Logger | str, optional*) – Logger used for printing related information during evaluation. Defaults to None.
- **show** (*bool, optional*) – Whether to visualize. Default: False.
- **out_dir** (*str, optional*) – Path to save the visualization results. Default: None.
- **pipeline** (*list[dict], optional*) – raw data loading for showing. Default: None.

Returns Evaluation results.

Return type dict

format_results(*outputs*, *pklfile_prefix=None*, *submission_prefix=None*)

Format the results to pkl file.

Parameters

- **outputs** (*list[dict]*) – Testing results of the dataset.
- **pklfile_prefix** (*str*) – The prefix of pkl files. It includes the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.

Returns

(outputs, tmp_dir), outputs is the detection results, tmp_dir is the temporal directory created for saving json files when `jsonfile_prefix` is not specified.

Return type tuple

`get_ann_info`(*index*)
Get annotation info according to the given index.

Parameters **`index`** (*`int`*) – Index of the annotation data to get.

Returns

Annotation information consists of the following keys:

- **gt_bboxes_3d (`LiDARInstance3DBoxes`):** 3D ground truth bboxes
- gt_labels_3d (np.ndarray): Labels of ground truths.
- gt_names (list[str]): Class names of ground truths.

Return type dict

`classmethod get_classes`(*classes=None*)
Get class names of current dataset.

Parameters **`classes`** (*`Sequence[str] | str`*) – If classes is None, use default CLASSES defined by builtin dataset. If classes is a string, take it as a file name. The file contains the name of classes where each line contains one class name. If classes is a tuple or list, override the CLASSES defined by the dataset.

Returns A list of class names.

Return type list[str]

`get_data_info`(*index*)
Get data info according to the given index.

Parameters **`index`** (*`int`*) – Index of the sample data to get.

Returns

Data information that will be passed to the data preprocessing pipelines. It includes the following keys:

- sample_idx (str): Sample index.
- pts_filename (str): Filename of point clouds.
- file_name (str): Filename of point clouds.
- ann_info (dict): Annotation info.

Return type dict

`load_annotations`(*ann_file*)
Load annotations from ann_file.

Parameters **`ann_file`** (*`str`*) – Path of the annotation file.

Returns List of annotations.

Return type list[dict]

`pre_pipeline`(*results*)
Initialization before data preparation.

Parameters **`results`** (*`dict`*) – Dict before data preprocessing.

- img_fields (list): Image fields.
- bbox3d_fields (list): 3D bounding boxes fields.
- pts_mask_fields (list): Mask fields of points.
- pts_seg_fields (list): Mask fields of point segments.
- bbox_fields (list): Fields of bounding boxes.
- mask_fields (list): Fields of masks.
- seg_fields (list): Segment fields.
- box_type_3d (str): 3D box type.
- box_mode_3d (str): 3D box mode.

prepare_test_data(*index*)
Prepare data for testing.

Parameters **index** (*int*) – Index for accessing the target data.

Returns Testing data dict of the corresponding index.

Return type dict

prepare_train_data(*index*)
Training data preparation.

Parameters **index** (*int*) – Index for accessing the target data.

Returns Training data dict of the corresponding index.

Return type dict

class mmdet3d.datasets.**Custom3DSegDataset**(*data_root*, *ann_file*, *pipeline=None*, *classes=None*, *palette=None*, *modality=None*, *test_mode=False*, *ignore_index=None*, *scene_idxs=None*, *file_client_args={'backend': 'disk'}*)
Customized 3D dataset for semantic segmentation task.

This is the base dataset of ScanNet and S3DIS dataset.

Parameters

- **data_root** (*str*) – Path of dataset root.
- **ann_file** (*str*) – Path of annotation file.
- **pipeline** (*list[dict], optional*) – Pipeline used for data processing. Defaults to None.
- **classes** (*tuple[str], optional*) – Classes used in the dataset. Defaults to None.
- **palette** (*list[list[int]], optional*) – The palette of segmentation map. Defaults to None.
- **modality** (*dict, optional*) – Modality to specify the sensor data used as input. Defaults to None.
- **test_mode** (*bool, optional*) – Whether the dataset is in test mode. Defaults to False.
- **ignore_index** (*int, optional*) – The label index to be ignored, e.g. unannotated points. If None is given, set to len(self.CLASSES) to be consistent with PointSegClassMapping function in pipeline. Defaults to None.

- **scene_idxs** (*np.ndarray | str, optional*) – Precomputed index to load data. For scenes with many points, we may sample it several times. Defaults to None.

evaluate(*results*, *metric=None*, *logger=None*, *show=False*, *out_dir=None*, *pipeline=None*)
Evaluate.

Evaluation in semantic segmentation protocol.

Parameters

- **results** (*list[dict]*) – List of results.
- **metric** (*str | list[str]*) – Metrics to be evaluated.
- **logger** (*logging.Logger | str, optional*) – Logger used for printing related information during evaluation. Defaults to None.
- **show** (*bool, optional*) – Whether to visualize. Defaults to False.
- **out_dir** (*str, optional*) – Path to save the visualization results. Defaults to None.
- **pipeline** (*list[dict], optional*) – raw data loading for showing. Default: None.

Returns Evaluation results.

Return type dict

format_results(*outputs*, *pklfile_prefix=None*, *submission_prefix=None*)
Format the results to pkl file.

Parameters

- **outputs** (*list[dict]*) – Testing results of the dataset.
- **pklfile_prefix** (*str*) – The prefix of pkl files. It includes the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.

Returns

(outputs, tmp_dir), outputs is the detection results, tmp_dir is the temporal directory created for saving json files when `jsonfile_prefix` is not specified.

Return type tuple

get_classes_and_palette(*classes=None*, *palette=None*)
Get class names of current dataset.

This function is taken from MMSegmentation.

Parameters

- **classes** (*Sequence[str] | str*) – If classes is None, use default CLASSES defined by builtin dataset. If classes is a string, take it as a file name. The file contains the name of classes where each line contains one class name. If classes is a tuple or list, override the CLASSES defined by the dataset. Defaults to None.
- **palette** (*Sequence[Sequence[int]]] | np.ndarray*) – The palette of segmentation map. If None is given, random palette will be generated. Defaults to None.

get_data_info(*index*)
Get data info according to the given index.

Parameters **index** (*int*) – Index of the sample data to get.

Returns

Data information that will be passed to the data preprocessing pipelines. It includes the following keys:

- sample_idx (str): Sample index.
- pts_filename (str): Filename of point clouds.
- file_name (str): Filename of point clouds.
- ann_info (dict): Annotation info.

Return type dict

`get_scene_idxs`(*scene_idxs*)
Compute scene_idxs for data sampling.

We sample more times for scenes with more points.

`load_annotations`(*ann_file*)
Load annotations from ann_file.

Parameters **ann_file** (*str*) – Path of the annotation file.

Returns List of annotations.

Return type list[dict]

`pre_pipeline`(*results*)
Initialization before data preparation.

Parameters **results** (*dict*) – Dict before data preprocessing.

- img_fields (list): Image fields.
- pts_mask_fields (list): Mask fields of points.
- pts_seg_fields (list): Mask fields of point segments.
- mask_fields (list): Fields of masks.
- seg_fields (list): Segment fields.

`prepare_test_data`(*index*)
Prepare data for testing.

Parameters **index** (*int*) – Index for accessing the target data.

Returns Testing data dict of the corresponding index.

Return type dict

`prepare_train_data`(*index*)
Training data preparation.

Parameters **index** (*int*) – Index for accessing the target data.

Returns Training data dict of the corresponding index.

Return type dict

class mmdet3d.datasets.**GlobalAlignment**(*rotation_axis*)
Apply global alignment to 3D scene points by rotation and translation.

Parameters **rotation_axis** (*int*) – Rotation axis for points and bboxes rotation.

Note:

We do not record the applied rotation and translation as in GlobalRotScaleTrans. Because usually, we do not need to reverse the alignment step.

For example, ScanNet 3D detection task uses aligned ground-truth bounding boxes for evaluation.

---

class mmdet3d.datasets.**GlobalRotScaleTrans**(*rot_range=[- 0.78539816, 0.78539816]*, *scale_ratio_range=[0.95, 1.05]*, *translation_std=[0, 0, 0]*, *shift_height=False*)

Apply global rotation, scaling and translation to a 3D scene.

Parameters

- **rot_range** (*list[float], optional*) – Range of rotation angle. Defaults to [-0.78539816, 0.78539816] (close to [-pi/4, pi/4]).
- **scale_ratio_range** (*list[float], optional*) – Range of scale ratio. Defaults to [0.95, 1.05].
- **translation_std** (*list[float], optional*) – The standard deviation of translation noise applied to a scene, which is sampled from a gaussian distribution whose standard deviation is set by `translation_std`. Defaults to [0, 0, 0]
- **shift_height** (*bool, optional*) – Whether to shift height. (the fourth dimension of indoor points) when scaling. Defaults to False.

class mmdet3d.datasets.**IndoorPatchPointSample**(*num_points*, *block_size=1.5*, *sample_rate=None*, *ignore_index=None*, *use_normalized_coord=False*, *num_try=10*, *enlarge_size=0.2*, *min_unique_num=None*, *eps=0.01*)

Indoor point sample within a patch. Modified from PointNet++.

Sampling data to a certain number for semantic segmentation.

Parameters

- **num_points** (*int*) – Number of points to be sampled.
- **block_size** (*float, optional*) – Size of a block to sample points from. Defaults to 1.5.
- **sample_rate** (*float, optional*) – Stride used in sliding patch generation. This parameter is unused in *IndoorPatchPointSample* and thus has been deprecated. We plan to remove it in the future. Defaults to None.
- **ignore_index** (*int, optional*) – Label index that won't be used for the segmentation task. This is set in PointSegClassMapping as neg_cls. If not None, will be used as a patch selection criterion. Defaults to None.
- **use_normalized_coord** (*bool, optional*) – Whether to use normalized xyz as additional features. Defaults to False.
- **num_try** (*int, optional*) – Number of times to try if the patch selected is invalid. Defaults to 10.
- **enlarge_size** (*float, optional*) – Enlarge the sampled patch to [-block_size / 2 - enlarge_size, block_size / 2 + enlarge_size] as an augmentation. If None, set it as 0. Defaults to 0.2.
- **min_unique_num** (*int, optional*) – Minimum number of unique points the sampled patch should contain. If None, use PointNet++'s method to judge uniqueness. Defaults to None.

- **eps** (*float, optional*) – A value added to patch boundary to guarantee points coverage. Defaults to 1e-2.

Note:

This transform should only be used in the training process of point cloud segmentation tasks. For the sliding patch generation and inference process in testing, please refer to the *slide_inference* function of *EncoderDecoder3D* class.

---

class mmdet3d.datasets.**IndoorPointSample**(**args*, ***kwargs*)
Indoor point sample.

Sampling data to a certain number. NOTE: IndoorPointSample is deprecated in favor of PointSample

Parameters **num_points** (*int*) – Number of points to be sampled.

class mmdet3d.datasets.**KittiDataset**(*data_root*, *ann_file*, *split*, *pts_prefix='velodyne'*, *pipeline=None*, *classes=None*, *modality=None*, *box_type_3d='LiDAR'*, *filter_empty_gt=True*, *test_mode=False*, *pcd_limit_range=[0, - 40, - 3, 70.4, 40, 0.0]*, ***kwargs*)
KITTI Dataset.

This class serves as the API for experiments on the KITTI Dataset.

Parameters

- **data_root** (*str*) – Path of dataset root.
- **ann_file** (*str*) – Path of annotation file.
- **split** (*str*) – Split of input data.
- **pts_prefix** (*str, optional*) – Prefix of points files. Defaults to 'velodyne'.
- **pipeline** (*list[dict], optional*) – Pipeline used for data processing. Defaults to None.
- **classes** (*tuple[str], optional*) – Classes used in the dataset. Defaults to None.
- **modality** (*dict, optional*) – Modality to specify the sensor data used as input. Defaults to None.
- **box_type_3d** (*str, optional*) – Type of 3D box of this dataset. Based on the *box_type_3d*, the dataset will encapsulate the box to its original format then converted them to *box_type_3d*. Defaults to 'LiDAR' in this dataset. Available options includes
  - 'LiDAR': Box in LiDAR coordinates.
  - 'Depth': Box in depth coordinates, usually for indoor dataset.
  - 'Camera': Box in camera coordinates.
- **filter_empty_gt** (*bool, optional*) – Whether to filter empty GT. Defaults to True.
- **test_mode** (*bool, optional*) – Whether the dataset is in test mode. Defaults to False.
- **pcd_limit_range** (*list, optional*) – The range of point cloud used to filter invalid predicted boxes. Default: [0, -40, -3, 70.4, 40, 0.0].

bbox2result_kitti(*net_outputs*, *class_names*, *pklfile_prefix=None*, *submission_prefix=None*)
Convert 3D detection results to kitti format for evaluation and test submission.

Parameters

- **net_outputs** (*list[np.ndarray]*) – List of array storing the inferenced bounding boxes and scores.
- **class_names** (*list[String]*) – A list of class names.
- **pklfile_prefix** (*str*) – The prefix of pkl file.
- **submission_prefix** (*str*) – The prefix of submission file.

Returns A list of dictionaries with the kitti format.

Return type list[dict]

bbox2result_kitti2d(*net_outputs*, *class_names*, *pklfile_prefix=None*, *submission_prefix=None*)
Convert 2D detection results to kitti format for evaluation and test submission.

Parameters

- **net_outputs** (*list[np.ndarray]*) – List of array storing the inferenced bounding boxes and scores.
- **class_names** (*list[String]*) – A list of class names.
- **pklfile_prefix** (*str*) – The prefix of pkl file.
- **submission_prefix** (*str*) – The prefix of submission file.

Returns A list of dictionaries have the kitti format

Return type list[dict]

convert_valid_bboxes(*box_dict*, *info*)
Convert the predicted boxes into valid ones.

Parameters

- **box_dict** (*dict*) – Box dictionaries to be converted.
  - boxes_3d (LiDARInstance3DBoxes): 3D bounding boxes.
  - scores_3d (torch.Tensor): Scores of boxes.
  - labels_3d (torch.Tensor): Class labels of boxes.
- **info** (*dict*) – Data info.

Returns

Valid predicted boxes.

- bbox (np.ndarray): 2D bounding boxes.
- **box3d_camera (np.ndarray): 3D bounding boxes in** camera coordinate.
- **box3d_lidar (np.ndarray): 3D bounding boxes in** LiDAR coordinate.
- scores (np.ndarray): Scores of boxes.
- label_preds (np.ndarray): Class label predictions.
- sample_idx (int): Sample index.

Return type dict

drop_arrays_by_name(*gt_names*, *used_classes*)
Drop irrelevant ground truths by name.

Parameters

- **gt_names** (*list[str]*) – Names of ground truths.

- **used_classes** (*list[str]*) – Classes of interest.

Returns Indices of ground truths that will be dropped.

Return type np.ndarray

evaluate(*results*, *metric=None*, *logger=None*, *pklfile_prefix=None*, *submission_prefix=None*, *show=False*, *out_dir=None*, *pipeline=None*)

Evaluation in KITTI protocol.

Parameters

- **results** (*list[dict]*) – Testing results of the dataset.
- **metric** (*str | list[str], optional*) – Metrics to be evaluated. Default: None.
- **logger** (*logging.Logger | str, optional*) – Logger used for printing related information during evaluation. Default: None.
- **pklfile_prefix** (*str, optional*) – The prefix of pkl files, including the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.
- **submission_prefix** (*str, optional*) – The prefix of submission data. If not specified, the submission data will not be generated. Default: None.
- **show** (*bool, optional*) – Whether to visualize. Default: False.
- **out_dir** (*str, optional*) – Path to save the visualization results. Default: None.
- **pipeline** (*list[dict], optional*) – raw data loading for showing. Default: None.

Returns Results of each evaluation metric.

Return type dict[str, float]

format_results(*outputs*, *pklfile_prefix=None*, *submission_prefix=None*)

Format the results to pkl file.

Parameters

- **outputs** (*list[dict]*) – Testing results of the dataset.
- **pklfile_prefix** (*str*) – The prefix of pkl files. It includes the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.
- **submission_prefix** (*str*) – The prefix of submitted files. It includes the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.

Returns

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

Return type tuple

get_ann_info(*index*)

Get annotation info according to the given index.

Parameters **index** (*int*) – Index of the annotation data to get.

Returns

annotation information consists of the following keys:

- **gt_bboxes_3d (LiDARInstance3DBoxes):** 3D ground truth bboxes.

- gt_labels_3d (np.ndarray): Labels of ground truths.
- gt_bboxes (np.ndarray): 2D ground truth bboxes.
- gt_labels (np.ndarray): Labels of ground truths.
- gt_names (list[str]): Class names of ground truths.
- **difficulty (int): Difficulty defined by KITTI.** 0, 1, 2 represent xxxxx respectively.

Return type dict

get_data_info(*index*)
Get data info according to the given index.

Parameters **index** (*int*) – Index of the sample data to get.

Returns

Data information that will be passed to the data preprocessing pipelines. It includes the following keys:

- sample_idx (str): Sample index.
- pts_filename (str): Filename of point clouds.
- img_prefix (str): Prefix of image files.
- img_info (dict): Image info.
- **lidar2img (list[np.ndarray], optional): Transformations** from lidar to different cameras.
- ann_info (dict): Annotation info.

Return type dict

keep_arrays_by_name(*gt_names*, *used_classes*)
Keep useful ground truths by name.

Parameters

- **gt_names** (*list[str]*) – Names of ground truths.
- **used_classes** (*list[str]*) – Classes of interest.

Returns Indices of ground truths that will be keeped.

Return type np.ndarray

remove_dontcare(*ann_info*)
Remove annotations that do not need to be cared.

Parameters **ann_info** (*dict*) – Dict of annotation infos. The `'DontCare'` annotations will be removed according to ann_file['name'].

Returns Annotations after filtering.

Return type dict

show(*results*, *out_dir*, *show=True*, *pipeline=None*)
Results visualization.

Parameters

- **results** (*list[dict]*) – List of bounding boxes results.
- **out_dir** (*str*) – Output directory of visualization result.

- **show** (*bool*) – Whether to visualize the results online. Default: False.
- **pipeline** (*list[dict], optional*) – raw data loading for showing. Default: None.

class mmdet3d.datasets.**KittiMonoDataset**(*data_root*, *info_file*, *ann_file*, *pipeline*, *load_interval=1*, *with_velocity=False*, *eval_version=None*, *version=None*, ***kwargs*)

Monocular 3D detection on KITTI Dataset.

Parameters

- **data_root** (*str*) – Path of dataset root.
- **info_file** (*str*) – Path of info file.
- **load_interval** (*int, optional*) – Interval of loading the dataset. It is used to uniformly sample the dataset. Defaults to 1.
- **with_velocity** (*bool, optional*) – Whether include velocity prediction into the experiments. Defaults to False.
- **eval_version** (*str, optional*) – Configuration version of evaluation. Defaults to None.
- **version** (*str, optional*) – Dataset version. Defaults to None.
- **kwargs** (*dict*) – Other arguments are the same of NuScenesMonoDataset.

bbox2result_kitti(*net_outputs*, *class_names*, *pklfile_prefix=None*, *submission_prefix=None*)

Convert 3D detection results to kitti format for evaluation and test submission.

Parameters

- **net_outputs** (*list[np.ndarray]*) – List of array storing the inferenced bounding boxes and scores.
- **class_names** (*list[String]*) – A list of class names.
- **pklfile_prefix** (*str*) – The prefix of pkl file.
- **submission_prefix** (*str*) – The prefix of submission file.

Returns A list of dictionaries with the kitti format.

Return type list[dict]

bbox2result_kitti2d(*net_outputs*, *class_names*, *pklfile_prefix=None*, *submission_prefix=None*)

Convert 2D detection results to kitti format for evaluation and test submission.

Parameters

- **net_outputs** (*list[np.ndarray]*) – List of array storing the inferenced bounding boxes and scores.
- **class_names** (*list[String]*) – A list of class names.
- **pklfile_prefix** (*str*) – The prefix of pkl file.
- **submission_prefix** (*str*) – The prefix of submission file.

Returns A list of dictionaries have the kitti format

Return type list[dict]

convert_valid_bboxes(*box_dict*, *info*)

Convert the predicted boxes into valid ones.

Parameters

- **box_dict** (*dict*) – Box dictionaries to be converted. - boxes_3d (CameraInstance3DBoxes): 3D bounding boxes. - scores_3d (torch.Tensor): Scores of boxes. - labels_3d (torch.Tensor): Class labels of boxes.
- **info** (*dict*) – Data info.

Returns

Valid predicted boxes.

- bbox (np.ndarray): 2D bounding boxes.
- **box3d_camera (np.ndarray): 3D bounding boxes in** camera coordinate.
- scores (np.ndarray): Scores of boxes.
- label_preds (np.ndarray): Class label predictions.
- sample_idx (int): Sample index.

Return type dict

evaluate(*results*, *metric=None*, *logger=None*, *pklfile_prefix=None*, *submission_prefix=None*, *show=False*, *out_dir=None*, *pipeline=None*)

Evaluation in KITTI protocol.

Parameters

- **results** (*list[dict]*) – Testing results of the dataset.
- **metric** (*str | list[str], optional*) – Metrics to be evaluated. Defaults to None.
- **logger** (*logging.Logger | str, optional*) – Logger used for printing related information during evaluation. Default: None.
- **pklfile_prefix** (*str, optional*) – The prefix of pkl files, including the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.
- **submission_prefix** (*str, optional*) – The prefix of submission data. If not specified, the submission data will not be generated.
- **show** (*bool, optional*) – Whether to visualize. Default: False.
- **out_dir** (*str, optional*) – Path to save the visualization results. Default: None.
- **pipeline** (*list[dict], optional*) – raw data loading for showing. Default: None.

Returns Results of each evaluation metric.

Return type dict[str, float]

format_results(*outputs*, *pklfile_prefix=None*, *submission_prefix=None*)

Format the results to pkl file.

Parameters

- **outputs** (*list[dict]*) – Testing results of the dataset.
- **pklfile_prefix** (*str*) – The prefix of pkl files. It includes the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.

- **submission_prefix** (*str*) – The prefix of submitted files. It includes the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.

Returns

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

Return type tuple

class mmdet3d.datasets.**LoadAnnotations3D**(*with_bbox_3d=True*, *with_label_3d=True*, *with_attr_label=False*, *with_mask_3d=False*, *with_seg_3d=False*, *with_bbox=False*, *with_label=False*, *with_mask=False*, *with_seg=False*, *with_bbox_depth=False*, *poly2mask=True*, *seg_3d_dtype=<class 'numpy.int64'>*, *file_client_args={'backend': 'disk'}*)

Load Annotations3D.

Load instance mask and semantic mask of points and encapsulate the items into related fields.

Parameters

- **with_bbox_3d** (*bool, optional*) – Whether to load 3D boxes. Defaults to True.
- **with_label_3d** (*bool, optional*) – Whether to load 3D labels. Defaults to True.
- **with_attr_label** (*bool, optional*) – Whether to load attribute label. Defaults to False.
- **with_mask_3d** (*bool, optional*) – Whether to load 3D instance masks. for points. Defaults to False.
- **with_seg_3d** (*bool, optional*) – Whether to load 3D semantic masks. for points. Defaults to False.
- **with_bbox** (*bool, optional*) – Whether to load 2D boxes. Defaults to False.
- **with_label** (*bool, optional*) – Whether to load 2D labels. Defaults to False.
- **with_mask** (*bool, optional*) – Whether to load 2D instance masks. Defaults to False.
- **with_seg** (*bool, optional*) – Whether to load 2D semantic masks. Defaults to False.
- **with_bbox_depth** (*bool, optional*) – Whether to load 2.5D boxes. Defaults to False.
- **poly2mask** (*bool, optional*) – Whether to convert polygon annotations to bitmasks. Defaults to True.
- **seg_3d_dtype** (*dtype, optional*) – Dtype of 3D semantic masks. Defaults to int64
- **file_client_args** (*dict*) – Config dict of file clients, refer to https://github.com/open-mmlab/mmcv/blob/master/mmcv/fileio/file_client.py for more details.

class mmdet3d.datasets.**LoadPointsFromDict**(*coord_type*, *load_dim=6*, *use_dim=[0, 1, 2]*, *shift_height=False*, *use_color=False*, *file_client_args={'backend': 'disk'}*)

Load Points From Dict.

class mmdet3d.datasets.**LoadPointsFromFile**(*coord_type*, *load_dim=6*, *use_dim=[0, 1, 2]*, *shift_height=False*, *use_color=False*, *file_client_args={'backend': 'disk'}*)

Load Points From File.

Load points from file.

Parameters

- **coord_type** (*str*) – The type of coordinates of points cloud. Available options includes: - 'LIDAR': Points in LiDAR coordinates. - 'DEPTH': Points in depth coordinates, usually for indoor dataset. - 'CAMERA': Points in camera coordinates.
- **load_dim** (*int, optional*) – The dimension of the loaded points. Defaults to 6.
- **use_dim** (*list[int], optional*) – Which dimensions of the points to use. Defaults to [0, 1, 2]. For KITTI dataset, set use_dim=4 or use_dim=[0, 1, 2, 3] to use the intensity dimension.
- **shift_height** (*bool, optional*) – Whether to use shifted height. Defaults to False.
- **use_color** (*bool, optional*) – Whether to use color features. Defaults to False.
- **file_client_args** (*dict, optional*) – Config dict of file clients, refer to https://github.com/open-mmlab/mmcv/blob/master/mmcv/fileio/file_client.py for more details. Defaults to dict(backend='disk').

class mmdet3d.datasets.**LoadPointsFromMultiSweeps**(*sweeps_num=10*, *load_dim=5*, *use_dim=[0, 1, 2, 4]*, *time_dim=4*, *file_client_args={'backend': 'disk'}*, *pad_empty_sweeps=False*, *remove_close=False*, *test_mode=False*)

Load points from multiple sweeps.

This is usually used for nuScenes dataset to utilize previous sweeps.

Parameters

- **sweeps_num** (*int, optional*) – Number of sweeps. Defaults to 10.
- **load_dim** (*int, optional*) – Dimension number of the loaded points. Defaults to 5.
- **use_dim** (*list[int], optional*) – Which dimension to use. Defaults to [0, 1, 2, 4].
- **time_dim** (*int, optional*) – Which dimension to represent the timestamps of each points. Defaults to 4.
- **file_client_args** (*dict, optional*) – Config dict of file clients, refer to https://github.com/open-mmlab/mmcv/blob/master/mmcv/fileio/file_client.py for more details. Defaults to dict(backend='disk').
- **pad_empty_sweeps** (*bool, optional*) – Whether to repeat keyframe when sweeps is empty. Defaults to False.
- **remove_close** (*bool, optional*) – Whether to remove close points. Defaults to False.
- **test_mode** (*bool, optional*) – If *test_mode=True*, it will not randomly sample sweeps but select the nearest N frames. Defaults to False.

class mmdet3d.datasets.**LyftDataset**(*ann_file*, *pipeline=None*, *data_root=None*, *classes=None*, *load_interval=1*, *modality=None*, *box_type_3d='LiDAR'*, *filter_empty_gt=True*, *test_mode=False*, ***kwargs*)

Lyft Dataset.

This class serves as the API for experiments on the Lyft Dataset.

Please refer to https://www.kaggle.com/c/3d-object-detection-for-autonomous-vehicles/data for data downloading.

Parameters

- **ann_file** (*str*) – Path of annotation file.

- **pipeline** (*list[dict], optional*) – Pipeline used for data processing. Defaults to None.
- **data_root** (*str*) – Path of dataset root.
- **classes** (*tuple[str], optional*) – Classes used in the dataset. Defaults to None.
- **load_interval** (*int, optional*) – Interval of loading the dataset. It is used to uniformly sample the dataset. Defaults to 1.
- **modality** (*dict, optional*) – Modality to specify the sensor data used as input. Defaults to None.
- **box_type_3d** (*str, optional*) – Type of 3D box of this dataset. Based on the *box_type_3d*, the dataset will encapsulate the box to its original format then converted them to *box_type_3d*. Defaults to 'LiDAR' in this dataset. Available options includes
  - 'LiDAR': Box in LiDAR coordinates.
  - 'Depth': Box in depth coordinates, usually for indoor dataset.
  - 'Camera': Box in camera coordinates.
- **filter_empty_gt** (*bool, optional*) – Whether to filter empty GT. Defaults to True.
- **test_mode** (*bool, optional*) – Whether the dataset is in test mode. Defaults to False.

evaluate(*results*, *metric='bbox'*, *logger=None*, *jsonfile_prefix=None*, *csv_savepath=None*, *result_names=['pts_bbox']*, *show=False*, *out_dir=None*, *pipeline=None*)

Evaluation in Lyft protocol.

Parameters

- **results** (*list[dict]*) – Testing results of the dataset.
- **metric** (*str | list[str], optional*) – Metrics to be evaluated. Default: 'bbox'.
- **logger** (*logging.Logger | str, optional*) – Logger used for printing related information during evaluation. Default: None.
- **jsonfile_prefix** (*str, optional*) – The prefix of json files including the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.
- **csv_savepath** (*str, optional*) – The path for saving csv files. It includes the file path and the csv filename, e.g., "a/b/filename.csv". If not specified, the result will not be converted to csv file.
- **result_names** (*list[str], optional*) – Result names in the metric prefix. Default: ['pts_bbox'].
- **show** (*bool, optional*) – Whether to visualize. Default: False.
- **out_dir** (*str, optional*) – Path to save the visualization results. Default: None.
- **pipeline** (*list[dict], optional*) – raw data loading for showing. Default: None.

Returns Evaluation results.

Return type dict[str, float]

format_results(*results*, *jsonfile_prefix=None*, *csv_savepath=None*)

Format the results to json (standard format for COCO evaluation).

Parameters

- **results** (*list[dict]*) – Testing results of the dataset.
- **jsonfile_prefix** (*str*) – The prefix of json files. It includes the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.
- **csv_savepath** (*str*) – The path for saving csv files. It includes the file path and the csv filename, e.g., "a/b/filename.csv". If not specified, the result will not be converted to csv file.

Returns

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

Return type tuple

get_ann_info(*index*)

Get annotation info according to the given index.

Parameters **index** (*int*) – Index of the annotation data to get.

Returns

Annotation information consists of the following keys:

- **gt_bboxes_3d (LiDARInstance3DBoxes):** 3D ground truth bboxes.
- gt_labels_3d (np.ndarray): Labels of ground truths.
- gt_names (list[str]): Class names of ground truths.

Return type dict

get_data_info(*index*)

Get data info according to the given index.

Parameters **index** (*int*) – Index of the sample data to get.

Returns

Data information that will be passed to the data preprocessing pipelines. It includes the following keys:

- sample_idx (str): sample index
- pts_filename (str): filename of point clouds
- sweeps (list[dict]): infos of sweeps
- timestamp (float): sample timestamp
- img_filename (str, optional): image filename
- **lidar2img (list[np.ndarray], optional): transformations** from lidar to different cameras
- ann_info (dict): annotation info

Return type dict

json2csv(*json_path*, *csv_savepath*)

Convert the json file to csv format for submission.

Parameters

- **json_path** (*str*) – Path of the result json file.
- **csv_savepath** (*str*) – Path to save the csv file.

load_annotations(*ann_file*)
Load annotations from ann_file.

Parameters **ann_file** (*str*) – Path of the annotation file.

Returns List of annotations sorted by timestamps.

Return type list[dict]

show(*results*, *out_dir*, *show=False*, *pipeline=None*)
Results visualization.

Parameters

- **results** (*list[dict]*) – List of bounding boxes results.
- **out_dir** (*str*) – Output directory of visualization result.
- **show** (*bool*) – Whether to visualize the results online. Default: False.
- **pipeline** (*list[dict], optional*) – raw data loading for showing. Default: None.

class mmdet3d.datasets.**NormalizePointsColor**(*color_mean*)
Normalize color of points.

Parameters **color_mean** (*list[float]*) – Mean color of the point cloud.

class mmdet3d.datasets.**NuScenesDataset**(*ann_file*, *pipeline=None*, *data_root=None*, *classes=None*, *load_interval=1*, *with_velocity=True*, *modality=None*, *box_type_3d='LiDAR'*, *filter_empty_gt=True*, *test_mode=False*, *eval_version='detection_cvpr_2019'*, *use_valid_flag=False*)
NuScenes Dataset.

This class serves as the API for experiments on the NuScenes Dataset.

Please refer to NuScenes Dataset for data downloading.

Parameters

- **ann_file** (*str*) – Path of annotation file.
- **pipeline** (*list[dict], optional*) – Pipeline used for data processing. Defaults to None.
- **data_root** (*str*) – Path of dataset root.
- **classes** (*tuple[str], optional*) – Classes used in the dataset. Defaults to None.
- **load_interval** (*int, optional*) – Interval of loading the dataset. It is used to uniformly sample the dataset. Defaults to 1.
- **with_velocity** (*bool, optional*) – Whether include velocity prediction into the experiments. Defaults to True.
- **modality** (*dict, optional*) – Modality to specify the sensor data used as input. Defaults to None.
- **box_type_3d** (*str, optional*) – Type of 3D box of this dataset. Based on the *box_type_3d*, the dataset will encapsulate the box to its original format then converted them to *box_type_3d*. Defaults to 'LiDAR' in this dataset. Available options includes. -

'LiDAR': Box in LiDAR coordinates. - 'Depth': Box in depth coordinates, usually for indoor dataset. - 'Camera': Box in camera coordinates.

- **filter_empty_gt** (*bool, optional*) – Whether to filter empty GT. Defaults to True.
- **test_mode** (*bool, optional*) – Whether the dataset is in test mode. Defaults to False.
- **eval_version** (*bool, optional*) – Configuration version of evaluation. Defaults to 'detection_cvpr_2019'.
- **use_valid_flag** (*bool, optional*) – Whether to use *use_valid_flag* key in the info file as mask to filter gt_boxes and gt_names. Defaults to False.

evaluate(*results*, *metric='bbox'*, *logger=None*, *jsonfile_prefix=None*, *result_names=['pts_bbox']*, *show=False*, *out_dir=None*, *pipeline=None*)

Evaluation in nuScenes protocol.

Parameters

- **results** (*list[dict]*) – Testing results of the dataset.
- **metric** (*str | list[str], optional*) – Metrics to be evaluated. Default: 'bbox'.
- **logger** (*logging.Logger | str, optional*) – Logger used for printing related information during evaluation. Default: None.
- **jsonfile_prefix** (*str, optional*) – The prefix of json files including the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.
- **show** (*bool, optional*) – Whether to visualize. Default: False.
- **out_dir** (*str, optional*) – Path to save the visualization results. Default: None.
- **pipeline** (*list[dict], optional*) – raw data loading for showing. Default: None.

Returns Results of each evaluation metric.

Return type dict[str, float]

format_results(*results*, *jsonfile_prefix=None*)

Format the results to json (standard format for COCO evaluation).

Parameters

- **results** (*list[dict]*) – Testing results of the dataset.
- **jsonfile_prefix** (*str*) – The prefix of json files. It includes the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.

Returns

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

Return type tuple

get_ann_info(*index*)

Get annotation info according to the given index.

Parameters **index** (*int*) – Index of the annotation data to get.

Returns

Annotation information consists of the following keys:

- **gt_bboxes_3d (LiDARInstance3DBoxes):** 3D ground truth bboxes
- gt_labels_3d (np.ndarray): Labels of ground truths.
- gt_names (list[str]): Class names of ground truths.

Return type dict

`get_cat_ids`(*idx*)

Get category distribution of single scene.

Parameters **idx** (*int*) – Index of the data_info.

Returns

for each category, if the current scene contains such boxes, store a list containing idx, otherwise, store empty list.

Return type dict[list]

`get_data_info`(*index*)

Get data info according to the given index.

Parameters **index** (*int*) – Index of the sample data to get.

Returns

Data information that will be passed to the data preprocessing pipelines. It includes the following keys:

- sample_idx (str): Sample index.
- pts_filename (str): Filename of point clouds.
- sweeps (list[dict]): Infos of sweeps.
- timestamp (float): Sample timestamp.
- img_filename (str, optional): Image filename.
- **lidar2img (list[np.ndarray], optional): Transformations** from lidar to different cameras.
- ann_info (dict): Annotation info.

Return type dict

`load_annotations`(*ann_file*)

Load annotations from ann_file.

Parameters **ann_file** (*str*) – Path of the annotation file.

Returns List of annotations sorted by timestamps.

Return type list[dict]

`show`(*results*, *out_dir*, *show=False*, *pipeline=None*)

Results visualization.

Parameters

- **results** (*list[dict]*) – List of bounding boxes results.
- **out_dir** (*str*) – Output directory of visualization result.

- **show** (*bool*) – Whether to visualize the results online. Default: False.
- **pipeline** (*list[dict], optional*) – raw data loading for showing. Default: None.

class mmdet3d.datasets.**NuScenesMonoDataset**(*data_root*, *ann_file*, *pipeline*, *load_interval=1*, *with_velocity=True*, *modality=None*, *box_type_3d='Camera'*, *eval_version='detection_cvpr_2019'*, *use_valid_flag=False*, *version='v1.0-trainval'*, *classes=None*, *img_prefix=''*, *seg_prefix=None*, *proposal_file=None*, *test_mode=False*, *filter_empty_gt=True*, *file_client_args={'backend': 'disk'}*)

Monocular 3D detection on NuScenes Dataset.

This class serves as the API for experiments on the NuScenes Dataset.

Please refer to NuScenes Dataset for data downloading.

Parameters

- **ann_file** (*str*) – Path of annotation file.
- **data_root** (*str*) – Path of dataset root.
- **load_interval** (*int, optional*) – Interval of loading the dataset. It is used to uniformly sample the dataset. Defaults to 1.
- **with_velocity** (*bool, optional*) – Whether include velocity prediction into the experiments. Defaults to True.
- **modality** (*dict, optional*) – Modality to specify the sensor data used as input. Defaults to None.
- **box_type_3d** (*str, optional*) – Type of 3D box of this dataset. Based on the *box_type_3d*, the dataset will encapsulate the box to its original format then converted them to *box_type_3d*. Defaults to 'Camera' in this class. Available options includes. - 'LiDAR': Box in LiDAR coordinates. - 'Depth': Box in depth coordinates, usually for indoor dataset. - 'Camera': Box in camera coordinates.
- **eval_version** (*str, optional*) – Configuration version of evaluation. Defaults to 'detection_cvpr_2019'.
- **use_valid_flag** (*bool, optional*) – Whether to use *use_valid_flag* key in the info file as mask to filter gt_boxes and gt_names. Defaults to False.
- **version** (*str, optional*) – Dataset version. Defaults to 'v1.0-trainval'.

evaluate(*results*, *metric='bbox'*, *logger=None*, *jsonfile_prefix=None*, *result_names=['img_bbox']*, *show=False*, *out_dir=None*, *pipeline=None*)

Evaluation in nuScenes protocol.

Parameters

- **results** (*list[dict]*) – Testing results of the dataset.
- **metric** (*str | list[str], optional*) – Metrics to be evaluated. Default: 'bbox'.
- **logger** (*logging.Logger | str, optional*) – Logger used for printing related information during evaluation. Default: None.
- **jsonfile_prefix** (*str*) – The prefix of json files. It includes the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.

- **result_names** (*list[str], optional*) – Result names in the metric prefix. Default: ['img_bbox'].
- **show** (*bool, optional*) – Whether to visualize. Default: False.
- **out_dir** (*str, optional*) – Path to save the visualization results. Default: None.
- **pipeline** (*list[dict], optional*) – raw data loading for showing. Default: None.

Returns Results of each evaluation metric.

Return type dict[str, float]

format_results(*results*, *jsonfile_prefix=None*, ***kwargs*)

Format the results to json (standard format for COCO evaluation).

Parameters

- **results** (*list[tuple | numpy.ndarray]*) – Testing results of the dataset.
- **jsonfile_prefix** (*str*) – The prefix of json files. It includes the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.

Returns

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

Return type tuple

get_attr_name(*attr_idx*, *label_name*)

Get attribute from predicted index.

This is a workaround to predict attribute when the predicted velocity is not reliable. We map the predicted attribute index to the one in the attribute set. If it is consistent with the category, we will keep it. Otherwise, we will use the default attribute.

Parameters

- **attr_idx** (*int*) – Attribute index.
- **label_name** (*str*) – Predicted category name.

Returns Predicted attribute name.

Return type str

pre_pipeline(*results*)

Initialization before data preparation.

Parameters **results** (*dict*) – Dict before data preprocessing.

- img_fields (list): Image fields.
- bbox3d_fields (list): 3D bounding boxes fields.
- pts_mask_fields (list): Mask fields of points.
- pts_seg_fields (list): Mask fields of point segments.
- bbox_fields (list): Fields of bounding boxes.
- mask_fields (list): Fields of masks.
- seg_fields (list): Segment fields.

- box_type_3d (str): 3D box type.
- box_mode_3d (str): 3D box mode.

show(*results*, *out_dir*, *show=False*, *pipeline=None*)
Results visualization.

Parameters

- **results** (*list[dict]*) – List of bounding boxes results.
- **out_dir** (*str*) – Output directory of visualization result.
- **show** (*bool*) – Whether to visualize the results online. Default: False.
- **pipeline** (*list[dict], optional*) – raw data loading for showing. Default: None.

class mmdet3d.datasets.**ObjectNameFilter**(*classes*)
Filter GT objects by their names.

Parameters **classes** (*list[str]*) – List of class names to be kept for training.

class mmdet3d.datasets.**ObjectNoise**(*translation_std=[0.25, 0.25, 0.25]*, *global_rot_range=[0.0, 0.0]*, *rot_range=[- 0.15707963267, 0.15707963267]*, *num_try=100*)
Apply noise to each GT objects in the scene.

Parameters

- **translation_std** (*list[float], optional*) – Standard deviation of the distribution where translation noise are sampled from. Defaults to [0.25, 0.25, 0.25].
- **global_rot_range** (*list[float], optional*) – Global rotation to the scene. Defaults to [0.0, 0.0].
- **rot_range** (*list[float], optional*) – Object rotation range. Defaults to [-0.15707963267, 0.15707963267].
- **num_try** (*int, optional*) – Number of times to try if the noise applied is invalid. Defaults to 100.

class mmdet3d.datasets.**ObjectRangeFilter**(*point_cloud_range*)
Filter objects by the range.

Parameters **point_cloud_range** (*list[float]*) – Point cloud range.

class mmdet3d.datasets.**ObjectSample**(*db_sampler*, *sample_2d=False*, *use_ground_plane=False*)
Sample GT objects to the data.

Parameters

- **db_sampler** (*dict*) – Config dict of the database sampler.
- **sample_2d** (*bool*) – Whether to also paste 2D image patch to the images This should be true when applying multi-modality cut-and-paste. Defaults to False.
- **use_ground_plane** (*bool*) – Whether to use gound plane to adjust the 3D labels.

static remove_points_in_boxes(*points*, *boxes*)
Remove the points in the sampled bounding boxes.

Parameters

- **points** (BasePoints) – Input point cloud array.
- **boxes** (*np.ndarray*) – Sampled ground truth boxes.

Returns Points with those in the boxes removed.

Return type np.ndarray

class mmdet3d.datasets.**PointSample**(*num_points*, *sample_range=None*, *replace=False*)
Point sample.

Sampling data to a certain number.

Parameters

- **num_points** (*int*) – Number of points to be sampled.
- **sample_range** (*float, optional*) – The range where to sample points. If not None, the points with depth larger than *sample_range* are prior to be sampled. Defaults to None.
- **replace** (*bool, optional*) – Whether the sampling is with or without replacement. Defaults to False.

class mmdet3d.datasets.**PointShuffle**
Shuffle input points.

class mmdet3d.datasets.**PointsRangeFilter**(*point_cloud_range*)
Filter points by the range.

Parameters **point_cloud_range** (*list[float]*) – Point cloud range.

class mmdet3d.datasets.**RandomDropPointsColor**(*drop_ratio=0.2*)
Randomly set the color of points to all zeros.

Once this transform is executed, all the points' color will be dropped. Refer to PAConv for more details.

Parameters **drop_ratio** (*float, optional*) – The probability of dropping point colors. Defaults to 0.2.

class mmdet3d.datasets.**RandomFlip3D**(*sync_2d=True*, *flip_ratio_bev_horizontal=0.0*, *flip_ratio_bev_vertical=0.0*, ***kwargs*)
Flip the points & bbox.

If the input dict contains the key "flip", then the flag will be used, otherwise it will be randomly decided by a ratio specified in the init method.

Parameters

- **sync_2d** (*bool, optional*) – Whether to apply flip according to the 2D images. If True, it will apply the same flip as that to 2D images. If False, it will decide whether to flip randomly and independently to that of 2D images. Defaults to True.
- **flip_ratio_bev_horizontal** (*float, optional*) – The flipping probability in horizontal direction. Defaults to 0.0.
- **flip_ratio_bev_vertical** (*float, optional*) – The flipping probability in vertical direction. Defaults to 0.0.

random_flip_data_3d(*input_dict*, *direction='horizontal'*)
Flip 3D data randomly.

Parameters

- **input_dict** (*dict*) – Result dict from loading pipeline.
- **direction** (*str, optional*) – Flip direction. Default: 'horizontal'.

Returns

Flipped results, 'points', 'bbox3d_fields' keys are updated in the result dict.

Return type dict

class mmdet3d.datasets.**RandomJitterPoints**(*jitter_std=[0.01, 0.01, 0.01]*, *clip_range=[- 0.05, 0.05]*)

Randomly jitter point coordinates.

Different from the global translation in GlobalRotScaleTrans, here we apply different noises to each point in a scene.

Parameters

- **jitter_std** (*list[float]*) – The standard deviation of jittering noise. This applies random noise to all points in a 3D scene, which is sampled from a gaussian distribution whose standard deviation is set by `jitter_std`. Defaults to [0.01, 0.01, 0.01]
- **clip_range** (*list[float]*) – Clip the randomly generated jitter noise into this range. If None is given, don't perform clipping. Defaults to [-0.05, 0.05]

Note:

This transform should only be used in point cloud segmentation tasks because we don't transform ground-truth bboxes accordingly.

For similar transform in detection task, please refer to *ObjectNoise*.

---

class mmdet3d.datasets.**RandomShiftScale**(*shift_scale*, *aug_prob*)

Random shift scale.

Different from the normal shift and scale function, it doesn't directly shift or scale image. It can record the shift and scale infos into loading pipelines. It's designed to be used with AffineResize together.

Parameters

- **shift_scale** (*tuple[float]*) – Shift and scale range.
- **aug_prob** (*float*) – The shifting and scaling probability.

class mmdet3d.datasets.**S3DISDataset**(*data_root*, *ann_file*, *pipeline=None*, *classes=None*, *modality=None*, *box_type_3d='Depth'*, *filter_empty_gt=True*, *test_mode=False*, **kwargs*)

S3DIS Dataset for Detection Task.

This class is the inner dataset for S3DIS. Since S3DIS has 6 areas, we often train on 5 of them and test on the remaining one. The one for test is Area_5 as suggested in GSDN. To concatenate 5 areas during training *mmdet.datasets.dataset_wrappers.ConcatDataset* should be used.

Parameters

- **data_root** (*str*) – Path of dataset root.
- **ann_file** (*str*) – Path of annotation file.
- **pipeline** (*list[dict], optional*) – Pipeline used for data processing. Defaults to None.
- **classes** (*tuple[str], optional*) – Classes used in the dataset. Defaults to None.
- **modality** (*dict, optional*) – Modality to specify the sensor data used as input. Defaults to None.
- **box_type_3d** (*str, optional*) – Type of 3D box of this dataset. Based on the *box_type_3d*, the dataset will encapsulate the box to its original format then converted them to *box_type_3d*. Defaults to 'Depth' in this dataset. Available options includes

- 'LiDAR': Box in LiDAR coordinates.
- 'Depth': Box in depth coordinates, usually for indoor dataset.
- 'Camera': Box in camera coordinates.

- **filter_empty_gt** (*bool, optional*) – Whether to filter empty GT. Defaults to True.
- **test_mode** (*bool, optional*) – Whether the dataset is in test mode. Defaults to False.

get_ann_info(*index*)

Get annotation info according to the given index.

Parameters **index** (*int*) – Index of the annotation data to get.

Returns

annotation information consists of the following keys:

- **gt_bboxes_3d (DepthInstance3DBoxes):** 3D ground truth bboxes
- gt_labels_3d (np.ndarray): Labels of ground truths.
- pts_instance_mask_path (str): Path of instance masks.
- pts_semantic_mask_path (str): Path of semantic masks.

Return type dict

get_data_info(*index*)

Get data info according to the given index.

Parameters **index** (*int*) – Index of the sample data to get.

Returns

Data information that will be passed to the data preprocessing pipelines. It includes the following keys:

- pts_filename (str): Filename of point clouds.
- file_name (str): Filename of point clouds.
- ann_info (dict): Annotation info.

Return type dict

class mmdet3d.datasets.**S3DISSegDataset**(*data_root*, *ann_files*, *pipeline=None*, *classes=None*, *palette=None*, *modality=None*, *test_mode=False*, *ignore_index=None*, *scene_idxs=None*, ***kwargs*)

S3DIS Dataset for Semantic Segmentation Task.

This class serves as the API for experiments on the S3DIS Dataset. It wraps the provided datasets of different areas. We don't use *mmdet.datasets.dataset_wrappers.ConcatDataset* because we need to concat the *scene_idxs* of different areas.

Please refer to the google form for data downloading.

Parameters

- **data_root** (*str*) – Path of dataset root.
- **ann_files** (*list[str]*) – Path of several annotation files.
- **pipeline** (*list[dict], optional*) – Pipeline used for data processing. Defaults to None.
- **classes** (*tuple[str], optional*) – Classes used in the dataset. Defaults to None.

- **palette** (*list[list[int]], optional*) – The palette of segmentation map. Defaults to None.
- **modality** (*dict, optional*) – Modality to specify the sensor data used as input. Defaults to None.
- **test_mode** (*bool, optional*) – Whether the dataset is in test mode. Defaults to False.
- **ignore_index** (*int, optional*) – The label index to be ignored, e.g. unannotated points. If None is given, set to len(self.CLASSES). Defaults to None.
- **scene_idxs** (*list[np.ndarray] | list[str], optional*) – Precomputed index to load data. For scenes with many points, we may sample it several times. Defaults to None.

concat_data_infos(*data_infos*)
Concat data_infos from several datasets to form self.data_infos.

Parameters **data_infos** (*list[list[dict]]*) –

concat_scene_idxs(*scene_idxs*)
Concat scene_idxs from several datasets to form self.scene_idxs.

Needs to manually add offset to scene_idxs[1, 2, ...].

Parameters **scene_idxs** (*list[np.ndarray]*) –

class mmdet3d.datasets.**SUNRGBDDataset**(*data_root*, *ann_file*, *pipeline=None*, *classes=None*, *modality={'use_camera': True, 'use_lidar': True}*, *box_type_3d='Depth'*, *filter_empty_gt=True*, *test_mode=False*, ***kwargs*)

SUNRGBD Dataset.

This class serves as the API for experiments on the SUNRGBD Dataset.

See the download page for data downloading.

Parameters

- **data_root** (*str*) – Path of dataset root.
- **ann_file** (*str*) – Path of annotation file.
- **pipeline** (*list[dict], optional*) – Pipeline used for data processing. Defaults to None.
- **classes** (*tuple[str], optional*) – Classes used in the dataset. Defaults to None.
- **modality** (*dict, optional*) – Modality to specify the sensor data used as input. Defaults to None.
- **box_type_3d** (*str, optional*) – Type of 3D box of this dataset. Based on the *box_type_3d*, the dataset will encapsulate the box to its original format then converted them to *box_type_3d*. Defaults to 'Depth' in this dataset. Available options includes
  - 'LiDAR': Box in LiDAR coordinates.
  - 'Depth': Box in depth coordinates, usually for indoor dataset.
  - 'Camera': Box in camera coordinates.
- **filter_empty_gt** (*bool, optional*) – Whether to filter empty GT. Defaults to True.
- **test_mode** (*bool, optional*) – Whether the dataset is in test mode. Defaults to False.

evaluate(*results*, *metric=None*, *iou_thr=(0.25, 0.5)*, *iou_thr_2d=(0.5)*, *logger=None*, *show=False*, *out_dir=None*, *pipeline=None*)

Evaluate.

Evaluation in indoor protocol.

Parameters

- **results** (*list[dict]*) – List of results.
- **metric** (*str | list[str], optional*) – Metrics to be evaluated. Default: None.
- **iou_thr** (*list[float], optional*) – AP IoU thresholds for 3D evaluation. Default: (0.25, 0.5).
- **iou_thr_2d** (*list[float], optional*) – AP IoU thresholds for 2D evaluation. Default: (0.5,).
- **show** (*bool, optional*) – Whether to visualize. Default: False.
- **out_dir** (*str, optional*) – Path to save the visualization results. Default: None.
- **pipeline** (*list[dict], optional*) – raw data loading for showing. Default: None.

Returns Evaluation results.

Return type dict

get_ann_info(*index*)

Get annotation info according to the given index.

Parameters **index** (*int*) – Index of the annotation data to get.

Returns

annotation information consists of the following keys:

- **gt_bboxes_3d (DepthInstance3DBoxes):** 3D ground truth bboxes
- gt_labels_3d (np.ndarray): Labels of ground truths.
- pts_instance_mask_path (str): Path of instance masks.
- pts_semantic_mask_path (str): Path of semantic masks.

Return type dict

get_data_info(*index*)

Get data info according to the given index.

Parameters **index** (*int*) – Index of the sample data to get.

Returns

Data information that will be passed to the data preprocessing pipelines. It includes the following keys:

- sample_idx (str): Sample index.
- pts_filename (str, optional): Filename of point clouds.
- file_name (str, optional): Filename of point clouds.
- img_prefix (str, optional): Prefix of image files.
- img_info (dict, optional): Image info.
- calib (dict, optional): Camera calibration info.

- ann_info (dict): Annotation info.

Return type dict

show(*results*, *out_dir*, *show=True*, *pipeline=None*)
Results visualization.

Parameters

- **results** (*list[dict]*) – List of bounding boxes results.
- **out_dir** (*str*) – Output directory of visualization result.
- **show** (*bool*) – Visualize the results online.
- **pipeline** (*list[dict], optional*) – raw data loading for showing. Default: None.

class mmdet3d.datasets.**ScanNetDataset**(*data_root*, *ann_file*, *pipeline=None*, *classes=None*, *modality={'use_camera': False, 'use_depth': True}*, *box_type_3d='Depth'*, *filter_empty_gt=True*, *test_mode=False*, ***kwargs*)
ScanNet Dataset for Detection Task.

This class serves as the API for experiments on the ScanNet Dataset.

Please refer to the github repo for data downloading.

Parameters

- **data_root** (*str*) – Path of dataset root.
- **ann_file** (*str*) – Path of annotation file.
- **pipeline** (*list[dict], optional*) – Pipeline used for data processing. Defaults to None.
- **classes** (*tuple[str], optional*) – Classes used in the dataset. Defaults to None.
- **modality** (*dict, optional*) – Modality to specify the sensor data used as input. Defaults to None.
- **box_type_3d** (*str, optional*) – Type of 3D box of this dataset. Based on the *box_type_3d*, the dataset will encapsulate the box to its original format then converted them to *box_type_3d*. Defaults to 'Depth' in this dataset. Available options includes
  - 'LiDAR': Box in LiDAR coordinates.
  - 'Depth': Box in depth coordinates, usually for indoor dataset.
  - 'Camera': Box in camera coordinates.
- **filter_empty_gt** (*bool, optional*) – Whether to filter empty GT. Defaults to True.
- **test_mode** (*bool, optional*) – Whether the dataset is in test mode. Defaults to False.

get_ann_info(*index*)
Get annotation info according to the given index.

Parameters **index** (*int*) – Index of the annotation data to get.

Returns

annotation information consists of the following keys:

- **gt_bboxes_3d (DepthInstance3DBoxes):** 3D ground truth bboxes
- gt_labels_3d (np.ndarray): Labels of ground truths.

- pts_instance_mask_path (str): Path of instance masks.
- pts_semantic_mask_path (str): Path of semantic masks.
- **axis_align_matrix (np.ndarray): Transformation matrix for** global scene alignment.

Return type dict

`get_data_info`(*index*)
Get data info according to the given index.

Parameters **index** (*int*) – Index of the sample data to get.

Returns

Data information that will be passed to the data preprocessing pipelines. It includes the following keys:

- sample_idx (str): Sample index.
- pts_filename (str): Filename of point clouds.
- file_name (str): Filename of point clouds.
- img_prefix (str, optional): Prefix of image files.
- img_info (dict, optional): Image info.
- ann_info (dict): Annotation info.

Return type dict

`prepare_test_data`(*index*)
Prepare data for testing.

We should take axis_align_matrix from self.data_infos since we need to align point clouds.

Parameters **index** (*int*) – Index for accessing the target data.

Returns Testing data dict of the corresponding index.

Return type dict

`show`(*results*, *out_dir*, *show=True*, *pipeline=None*)
Results visualization.

Parameters

- **results** (*list[dict]*) – List of bounding boxes results.
- **out_dir** (*str*) – Output directory of visualization result.
- **show** (*bool*) – Visualize the results online.
- **pipeline** (*list[dict], optional*) – raw data loading for showing. Default: None.

class mmdet3d.datasets.**ScanNetInstanceSegDataset**(*data_root*, *ann_file*, *pipeline=None*, *classes=None*, *palette=None*, *modality=None*, *test_mode=False*, *ignore_index=None*, *scene_idxs=None*, *file_client_args={'backend': 'disk'}*)

`evaluate`(*results*, *metric=None*, *options=None*, *logger=None*, *show=False*, *out_dir=None*, *pipeline=None*)
Evaluation in instance segmentation protocol.

Parameters

- **results** (*list[dict]*) – List of results.
- **metric** (*str | list[str]*) – Metrics to be evaluated.
- **options** (*dict, optional*) – options for instance_seg_eval.
- **logger** (*logging.Logger | None | str*) – Logger used for printing related information during evaluation. Defaults to None.
- **show** (*bool, optional*) – Whether to visualize. Defaults to False.
- **out_dir** (*str, optional*) – Path to save the visualization results. Defaults to None.
- **pipeline** (*list[dict], optional*) – raw data loading for showing. Default: None.

Returns Evaluation results.

Return type dict

get_ann_info(*index*)

Get annotation info according to the given index.

Parameters **index** (*int*) – Index of the annotation data to get.

Returns

annotation information consists of the following keys:

- pts_semantic_mask_path (str): Path of semantic masks.
- pts_instance_mask_path (str): Path of instance masks.

Return type dict

get_classes_and_palette(*classes=None*, *palette=None*)

Get class names of current dataset. Palette is simply ignored for instance segmentation.

Parameters

- **classes** (*Sequence[str] | str | None*) – If classes is None, use default CLASSES defined by builtin dataset. If classes is a string, take it as a file name. The file contains the name of classes where each line contains one class name. If classes is a tuple or list, override the CLASSES defined by the dataset. Defaults to None.
- **palette** (*Sequence[Sequence[int]] | np.ndarray | None*) – The palette of segmentation map. If None is given, random palette will be generated. Defaults to None.

class mmdet3d.datasets.**ScanNetSegDataset**(*data_root*, *ann_file*, *pipeline=None*, *classes=None*, *palette=None*, *modality=None*, *test_mode=False*, *ignore_index=None*, *scene_idxs=None*, ***kwargs*)

ScanNet Dataset for Semantic Segmentation Task.

This class serves as the API for experiments on the ScanNet Dataset.

Please refer to the github repo for data downloading.

Parameters

- **data_root** (*str*) – Path of dataset root.
- **ann_file** (*str*) – Path of annotation file.

- **pipeline** (*list[dict], optional*) – Pipeline used for data processing. Defaults to None.
- **classes** (*tuple[str], optional*) – Classes used in the dataset. Defaults to None.
- **palette** (*list[list[int]], optional*) – The palette of segmentation map. Defaults to None.
- **modality** (*dict, optional*) – Modality to specify the sensor data used as input. Defaults to None.
- **test_mode** (*bool, optional*) – Whether the dataset is in test mode. Defaults to False.
- **ignore_index** (*int, optional*) – The label index to be ignored, e.g. unannotated points. If None is given, set to len(self.CLASSES). Defaults to None.
- **scene_idxs** (*np.ndarray | str, optional*) – Precomputed index to load data. For scenes with many points, we may sample it several times. Defaults to None.

format_results(*results*, *txtfile_prefix=None*)

Format the results to txt file. Refer to ScanNet documentation.

Parameters

- **outputs** (*list[dict]*) – Testing results of the dataset.
- **txtfile_prefix** (*str*) – The prefix of saved files. It includes the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.

Returns

(outputs, tmp_dir), outputs is the detection results, tmp_dir is the temporal directory created for saving submission files when `submission_prefix` is not specified.

Return type tuple

get_ann_info(*index*)

Get annotation info according to the given index.

Parameters **index** (*int*) – Index of the annotation data to get.

Returns

annotation information consists of the following keys:

- pts_semantic_mask_path (str): Path of semantic masks.

Return type dict

get_scene_idxs(*scene_idxs*)

Compute scene_idxs for data sampling.

We sample more times for scenes with more points.

show(*results*, *out_dir*, *show=True*, *pipeline=None*)

Results visualization.

Parameters

- **results** (*list[dict]*) – List of bounding boxes results.
- **out_dir** (*str*) – Output directory of visualization result.
- **show** (*bool*) – Visualize the results online.

- **pipeline** (*list[dict], optional*) – raw data loading for showing. Default: None.

class mmdet3d.datasets.**SemanticKITTIDataset**(*data_root*, *ann_file*, *pipeline=None*, *classes=None*, *modality=None*, *box_type_3d='Lidar'*, *filter_empty_gt=False*, *test_mode=False*)

SemanticKITTI Dataset.

This class serves as the API for experiments on the SemanticKITTI Dataset Please refer to <http://www.semantic-kitti.org/dataset.html>`_ for data downloading

Parameters

- **data_root** (*str*) – Path of dataset root.
- **ann_file** (*str*) – Path of annotation file.
- **pipeline** (*list[dict], optional*) – Pipeline used for data processing. Defaults to None.
- **classes** (*tuple[str], optional*) – Classes used in the dataset. Defaults to None.
- **modality** (*dict, optional*) – Modality to specify the sensor data used as input. Defaults to None.
- **box_type_3d** (*str, optional*) – NO 3D box for this dataset. You can choose any type Based on the *box_type_3d*, the dataset will encapsulate the box to its original format then converted them to *box_type_3d*. Defaults to 'LiDAR' in this dataset. Available options includes
  - 'LiDAR': Box in LiDAR coordinates.
  - 'Depth': Box in depth coordinates, usually for indoor dataset.
  - 'Camera': Box in camera coordinates.
- **filter_empty_gt** (*bool, optional*) – Whether to filter empty GT. Defaults to True.
- **test_mode** (*bool, optional*) – Whether the dataset is in test mode. Defaults to False.

get_ann_info(*index*)

Get annotation info according to the given index.

Parameters **index** (*int*) – Index of the annotation data to get.

Returns

annotation information consists of the following keys:

- pts_semantic_mask_path (str): Path of semantic masks.

Return type dict

get_data_info(*index*)

Get data info according to the given index. :param index: Index of the sample data to get. :type index: int

Returns

Data information that will be passed to the data preprocessing pipelines. It includes the following keys: - sample_idx (str): Sample index. - pts_filename (str): Filename of point clouds. - file_name (str): Filename of point clouds. - ann_info (dict): Annotation info.

Return type dict

class mmdet3d.datasets.**VoxelBasedPointSampler**(*cur_sweep_cfg*, *prev_sweep_cfg=None*, *time_dim=3*)
Voxel based point sampler.

Apply voxel sampling to multiple sweep points.

Parameters

- **cur_sweep_cfg** (*dict*) – Config for sampling current points.
- **prev_sweep_cfg** (*dict*) – Config for sampling previous points.
- **time_dim** (*int*) – Index that indicate the time dimension for input points.

class mmdet3d.datasets.**WaymoDataset**(*data_root*, *ann_file*, *split*, *pts_prefix='velodyne'*, *pipeline=None*, *classes=None*, *modality=None*, *box_type_3d='LiDAR'*, *filter_empty_gt=True*, *test_mode=False*, *load_interval=1*, *pcd_limit_range=[- 85, - 85, - 5, 85, 85, 5]*, ***kwargs*)
Waymo Dataset.

This class serves as the API for experiments on the Waymo Dataset.

Please refer to `<https://waymo.com/open/download/>`_for data downloading. It is recommended to symlink the dataset root to $MMDETECTION3D/data and organize them as the doc shows.

Parameters

- **data_root** (*str*) – Path of dataset root.
- **ann_file** (*str*) – Path of annotation file.
- **split** (*str*) – Split of input data.
- **pts_prefix** (*str, optional*) – Prefix of points files. Defaults to 'velodyne'.
- **pipeline** (*list[dict], optional*) – Pipeline used for data processing. Defaults to None.
- **classes** (*tuple[str], optional*) – Classes used in the dataset. Defaults to None.
- **modality** (*dict, optional*) – Modality to specify the sensor data used as input. Defaults to None.
- **box_type_3d** (*str, optional*) – Type of 3D box of this dataset. Based on the *box_type_3d*, the dataset will encapsulate the box to its original format then converted them to *box_type_3d*. Defaults to 'LiDAR' in this dataset. Available options includes
  - 'LiDAR': box in LiDAR coordinates
  - 'Depth': box in depth coordinates, usually for indoor dataset
  - 'Camera': box in camera coordinates
- **filter_empty_gt** (*bool, optional*) – Whether to filter empty GT. Defaults to True.
- **test_mode** (*bool, optional*) – Whether the dataset is in test mode. Defaults to False.
- **pcd_limit_range** (*list(float), optional*) – The range of point cloud used to filter invalid predicted boxes. Default: [-85, -85, -5, 85, 85, 5].

bbox2result_kitti(*net_outputs*, *class_names*, *pklfile_prefix=None*, *submission_prefix=None*)
Convert results to kitti format for evaluation and test submission.

Parameters

- **net_outputs** (*List[np.ndarray]*) – list of array storing the bbox and score
- **class_nanes** (*List[String]*) – A list of class names

- **pklfile_prefix** (*str*) – The prefix of pkl file.
- **submission_prefix** (*str*) – The prefix of submission file.

Returns A list of dict have the kitti 3d format

Return type List[dict]

convert_valid_bboxes(*box_dict*, *info*)
Convert the boxes into valid format.

Parameters

- **box_dict** (*dict*) – Bounding boxes to be converted.
  - boxes_3d (:obj:LiDARInstance3DBoxes): 3D bounding boxes.
  - scores_3d (np.ndarray): Scores of predicted boxes.
  - labels_3d (np.ndarray): Class labels of predicted boxes.
- **info** (*dict*) – Dataset information dictionary.

Returns

Valid boxes after conversion.

- bbox (np.ndarray): 2D bounding boxes (in camera 0).
- box3d_camera (np.ndarray): 3D boxes in camera coordinates.
- box3d_lidar (np.ndarray): 3D boxes in lidar coordinates.
- scores (np.ndarray): Scores of predicted boxes.
- label_preds (np.ndarray): Class labels of predicted boxes.
- sample_idx (np.ndarray): Sample index.

Return type dict

evaluate(*results*, *metric='waymo'*, *logger=None*, *pklfile_prefix=None*, *submission_prefix=None*, *show=False*, *out_dir=None*, *pipeline=None*)
Evaluation in KITTI protocol.

Parameters

- **results** (*list[dict]*) – Testing results of the dataset.
- **metric** (*str | list[str], optional*) – Metrics to be evaluated. Default: 'waymo'. Another supported metric is 'kitti'.
- **logger** (*logging.Logger | str, optional*) – Logger used for printing related information during evaluation. Default: None.
- **pklfile_prefix** (*str, optional*) – The prefix of pkl files including the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.
- **submission_prefix** (*str, optional*) – The prefix of submission data. If not specified, the submission data will not be generated.
- **show** (*bool, optional*) – Whether to visualize. Default: False.
- **out_dir** (*str, optional*) – Path to save the visualization results. Default: None.
- **pipeline** (*list[dict], optional*) – raw data loading for showing. Default: None.

Returns float]: results of each evaluation metric

Return type dict[str

format_results(*outputs*, *pklfile_prefix=None*, *submission_prefix=None*, *data_format='waymo'*)

Format the results to pkl file.

Parameters

- **outputs** (*list[dict]*) – Testing results of the dataset.
- **pklfile_prefix** (*str*) – The prefix of pkl files. It includes the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.
- **submission_prefix** (*str*) – The prefix of submitted files. It includes the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.
- **data_format** (*str, optional*) – Output data format. Default: 'waymo'. Another supported choice is 'kitti'.

Returns

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

Return type tuple

get_data_info(*index*)

Get data info according to the given index.

Parameters **index** (*int*) – Index of the sample data to get.

Returns

Standard input_dict consists of the data information.

- sample_idx (str): sample index
- pts_filename (str): filename of point clouds
- img_prefix (str): prefix of image files
- img_info (dict): image info
- **lidar2img (list[np.ndarray], optional): transformations from** lidar to different cameras
- ann_info (dict): annotation info

Return type dict

mmdet3d.datasets.**build_dataloader**(*dataset*, *samples_per_gpu*, *workers_per_gpu*, *num_gpus=1*, *dist=True*, *shuffle=True*, *seed=None*, *runner_type='EpochBasedRunner'*, *persistent_workers=False*, *class_aware_sampler=None*, ***kwargs*)

Build PyTorch DataLoader.

In distributed training, each GPU/process has a dataloader. In non-distributed training, there is only one dataloader for all GPUs.

Parameters

- **dataset** (*Dataset*) – A PyTorch dataset.
- **samples_per_gpu** (*int*) – Number of training samples on each GPU, i.e., batch size of each GPU.

- **workers_per_gpu** (*int*) – How many subprocesses to use for data loading for each GPU.
- **num_gpus** (*int*) – Number of GPUs. Only used in non-distributed training.
- **dist** (*bool*) – Distributed training/test or not. Default: True.
- **shuffle** (*bool*) – Whether to shuffle the data at every epoch. Default: True.
- **seed** (*int, Optional*) – Seed to be used. Default: None.
- **runner_type** (*str*) – Type of runner. Default: *EpochBasedRunner*
- **persistent_workers** (*bool*) – If True, the data loader will not shutdown the worker processes after a dataset has been consumed once. This allows to maintain the workers *Dataset* instances alive. This argument is only valid when PyTorch>=1.7.0. Default: False.
- **class_aware_sampler** (*dict*) – Whether to use *ClassAwareSampler* during training. Default: None.
- **kwargs** – any keyword argument to be used to initialize DataLoader

Returns A PyTorch dataloader.

Return type DataLoader

mmdet3d.datasets.**get_loading_pipeline**(*pipeline*)

Only keep loading image, points and annotations related configuration.

Parameters **pipeline** (list[dict] | list[Pipeline]) – Data pipeline configs or list of pipeline functions.

Returns

The new pipeline list with only keep loading image, points and annotations related configuration.

Return type list[dict] | list[Pipeline])

Examples

```
>>> pipelines = [
...     dict(type='LoadPointsFromFile',
...          coord_type='LIDAR', load_dim=4, use_dim=4),
...     dict(type='LoadImageFromFile'),
...     dict(type='LoadAnnotations3D',
...          with_bbox=True, with_label_3d=True),
...     dict(type='Resize',
...          img_scale=[(640, 192), (2560, 768)], keep_ratio=True),
...     dict(type='RandomFlip3D', flip_ratio_bev_horizontal=0.5),
...     dict(type='PointsRangeFilter',
...          point_cloud_range=point_cloud_range),
...     dict(type='ObjectRangeFilter',
...          point_cloud_range=point_cloud_range),
...     dict(type='PointShuffle'),
...     dict(type='Normalize', **img_norm_cfg),
...     dict(type='Pad', size_divisor=32),
...     dict(type='DefaultFormatBundle3D', class_names=class_names),
...     dict(type='Collect3D',
...          keys=['points', 'img', 'gt_bboxes_3d', 'gt_labels_3d'])
```

(continues on next page)

(continued from previous page)

```
...     ]
>>> expected_pipelines = [
...     dict(type='LoadPointsFromFile',
...          coord_type='LIDAR', load_dim=4, use_dim=4),
...     dict(type='LoadImageFromFile'),
...     dict(type='LoadAnnotations3D',
...          with_bbox=True, with_label_3d=True),
...     dict(type='DefaultFormatBundle3D', class_names=class_names),
...     dict(type='Collect3D',
...          keys=['points', 'img', 'gt_bboxes_3d', 'gt_labels_3d'])
...     ]
>>> assert expected_pipelines ==          ...         get_loading_pipeline(pipelines)
```

CHAPTER

FORTYSEVEN

MMDET3D.MODELS

47.1 detectors

class mmdet3d.models.detectors.**Base3DDetector**(*init_cfg=None*)

Base class for detectors.

forward(*return_loss=True*, ***kwargs*)

Calls either forward_train or forward_test depending on whether return_loss=True.

Note this setting will change the expected inputs. When *return_loss=True*, img and img_metas are single-nested (i.e. torch.Tensor and list[dict]), and when *resturn_loss=False*, img and img_metas should be double nested (i.e. list[torch.Tensor], list[list[dict]]), with the outer list indicating test time augmentations.

forward_test(*points*, *img_metas*, *img=None*, ***kwargs*)

Parameters

- **points** (*list[torch.Tensor]*) – the outer list indicates test-time augmentations and inner torch.Tensor should have a shape NxC, which contains all points in the batch.
- **img_metas** (*list[list[dict]]*) – the outer list indicates test-time augs (multi-scale, flip, etc.) and the inner list indicates images in a batch
- **img** (*list[torch.Tensor], optional*) – the outer list indicates test-time augmentations and inner torch.Tensor should have a shape NxCxHxW, which contains all images in the batch. Defaults to None.

show_results(*data*, *result*, *out_dir*, *show=False*, *score_thr=None*)

Results visualization.

Parameters

- **data** (*list[dict]*) – Input points and the information of the sample.
- **result** (*list[dict]*) – Prediction results.
- **out_dir** (*str*) – Output directory of visualization result.
- **show** (*bool, optional*) – Determines whether you are going to show result by open3d. Defaults to False.
- **score_thr** (*float, optional*) – Score threshold of bounding boxes. Default to None.

class mmdet3d.models.detectors.**CenterPoint**(*pts_voxel_layer=None*, *pts_voxel_encoder=None*, *pts_middle_encoder=None*, *pts_fusion_layer=None*, *img_backbone=None*, *pts_backbone=None*, *img_neck=None*, *pts_neck=None*, *pts_bbox_head=None*, *img_roi_head=None*, *img_rpn_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)

Base class of Multi-modality VoxelNet.

aug_test(*points*, *img_metas*, *imgs=None*, *rescale=False*)

Test function with augmentaiton.

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

Test function of point cloud branch with augmentaiton.

The function implementation process is as follows:

- step 1: map features back for double-flip augmentation.
- step 2: merge all features and generate boxes.
- step 3: map boxes back for scale augmentation.
- step 4: merge results.

Parameters

- **feats** (*list[torch.Tensor]*) – Feature of point cloud.
- **img_metas** (*list[dict]*) – Meta information of samples.
- **rescale** (*bool, optional*) – Whether to rescale bboxes. Default: False.

Returns

Returned bboxes consists of the following keys:

- boxes_3d (LiDARInstance3DBoxes): Predicted bboxes.
- scores_3d (torch.Tensor): Scores of predicted boxes.
- labels_3d (torch.Tensor): Labels of predicted boxes.

Return type dict

extract_pts_feat(*pts*, *img_feats*, *img_metas*)

Extract features of points.

forward_pts_train(*pts_feats*, *gt_bboxes_3d*, *gt_labels_3d*, *img_metas*, *gt_bboxes_ignore=None*)

Forward function for point cloud branch.

Parameters

- **pts_feats** (*list[torch.Tensor]*) – Features of point cloud branch
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth boxes for each sample.
- **gt_labels_3d** (*list[torch.Tensor]*) – Ground truth labels for boxes of each sampole
- **img_metas** (*list[dict]*) – Meta information of samples.
- **gt_bboxes_ignore** (*list[torch.Tensor], optional*) – Ground truth boxes to be ignored. Defaults to None.

Returns Losses of each branch.

Return type dict

simple_test_pts(*x*, *img_metas*, *rescale=False*)
Test function of point cloud branch.

property with_velocity
Whether the head predicts velocity

Type bool

class mmdet3d.models.detectors.**DynamicMVXFasterRCNN**(***kwargs*)
Multi-modality VoxelNet using Faster R-CNN and dynamic voxelization.

extract_pts_feat(*points*, *img_feats*, *img_metas*)
Extract point features.

voxelize(*points*)
Apply dynamic voxelization to points.

Parameters **points** (*list[torch.Tensor]*) – Points of each sample.

Returns Concatenated points and coordinates.

Return type tuple[torch.Tensor]

class mmdet3d.models.detectors.**DynamicVoxelNet**(*voxel_layer*, *voxel_encoder*, *middle_encoder*, *backbone*, *neck=None*, *bbox_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)
VoxelNet using dynamic voxelization.

extract_feat(*points*, *img_metas*)
Extract features from points.

voxelize(*points*)
Apply dynamic voxelization to points.

Parameters **points** (*list[torch.Tensor]*) – Points of each sample.

Returns Concatenated points and coordinates.

Return type tuple[torch.Tensor]

class mmdet3d.models.detectors.**FCOSMono3D**(*backbone*, *neck*, *bbox_head*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*)
FCOS3D for monocular 3D object detection.

Currently please refer to our entry on the leaderboard.

class mmdet3d.models.detectors.**GroupFree3DNet**(*backbone*, *bbox_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*)
Group-Free 3D.

aug_test(*points*, *img_metas*, *imgs=None*, *rescale=False*)
Test with augmentation.

forward_train(*points*, *img_metas*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*, *gt_bboxes_ignore=None*)
Forward of training.

Parameters

- **points** (*list[torch.Tensor]*) – Points of each batch.
- **img_metas** (*list*) – Image metas.
- **gt_bboxes_3d** (BaseInstance3DBoxes) – gt bboxes of each batch.

- **gt_labels_3d** (*list[torch.Tensor]*) – gt class labels of each batch.
- **pts_semantic_mask** (*list[torch.Tensor]*) – point-wise semantic label of each batch.
- **pts_instance_mask** (*list[torch.Tensor]*) – point-wise instance label of each batch.
- **gt_bboxes_ignore** (*list[torch.Tensor]*) – Specify which bounding.

Returns torch.Tensor]: Losses.

Return type dict[str

simple_test(*points*, *img_metas*, *imgs=None*, *rescale=False*)
Forward of testing.

Parameters

- **points** (*list[torch.Tensor]*) – Points of each sample.
- **img_metas** (*list*) – Image metas.
- **rescale** (*bool*) – Whether to rescale results.

Returns Predicted 3d boxes.

Return type list

class mmdet3d.models.detectors.**H3DNet**(*backbone*, *neck=None*, *rpn_head=None*, *roi_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)
H3DNet model.

Please refer to the paper

aug_test(*points*, *img_metas*, *imgs=None*, *rescale=False*)
Test with augmentation.

extract_feats(*points*, *img_metas*)
Extract features of multiple samples.

forward_train(*points*, *img_metas*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*, *gt_bboxes_ignore=None*)
Forward of training.

Parameters

- **points** (*list[torch.Tensor]*) – Points of each batch.
- **img_metas** (*list*) – Image metas.
- **gt_bboxes_3d** (BaseInstance3DBoxes) – gt bboxes of each batch.
- **gt_labels_3d** (*list[torch.Tensor]*) – gt class labels of each batch.
- **pts_semantic_mask** (*list[torch.Tensor]*) – point-wise semantic label of each batch.
- **pts_instance_mask** (*list[torch.Tensor]*) – point-wise instance label of each batch.
- **gt_bboxes_ignore** (*list[torch.Tensor]*) – Specify which bounding.

Returns Losses.

Return type dict

simple_test(*points*, *img_metas*, *imgs=None*, *rescale=False*)
Forward of testing.

Parameters

- **points** (*list[torch.Tensor]*) – Points of each sample.
- **img_metas** (*list*) – Image metas.
- **rescale** (*bool*) – Whether to rescale results.

Returns Predicted 3d boxes.

Return type list

class mmdet3d.models.detectors.**ImVoteNet**(*pts_backbone=None*, *pts_bbox_heads=None*, *pts_neck=None*, *img_backbone=None*, *img_neck=None*, *img_roi_head=None*, *img_rpn_head=None*, *img_mlp=None*, *freeze_img_branch=False*, *fusion_layer=None*, *num_sampled_seed=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)

ImVoteNet for 3D detection.

aug_test(*points=None*, *img_metas=None*, *imgs=None*, *bboxes_2d=None*, *rescale=False*, ***kwargs*)
Test function with augmentation, stage 2.

Parameters

- **points** (*list[list[torch.Tensor]], optional*) – the outer list indicates test-time augmentations and the inner list contains all points in the batch, where each Tensor should have a shape NxC. Defaults to None.
- **img_metas** (*list[list[dict]], optional*) – the outer list indicates test-time augs (multiscale, flip, etc.) and the inner list indicates images in a batch. Defaults to None.
- **imgs** (*list[list[torch.Tensor]], optional*) – the outer list indicates test-time augmentations and inner Tensor should have a shape NxCxHxW, which contains all images in the batch. Defaults to None. Defaults to None.
- **bboxes_2d** (*list[list[torch.Tensor]], optional*) – Provided 2d bboxes, not supported yet. Defaults to None.
- **rescale** (*bool, optional*) – Whether or not rescale bboxes. Defaults to False.

Returns Predicted 3d boxes.

Return type list[dict]

aug_test_img_only(*img*, *img_metas*, *rescale=False*)
Test function with augmentation, image network pretrain. May refer to https://github.com/open-mmlab/mmdetection/blob/master/mmdet/models/detectors/two_stage.py.

Parameters

- **img** (*list[list[torch.Tensor]], optional*) – the outer list indicates test-time augmentations and inner Tensor should have a shape NxCxHxW, which contains all images in the batch. Defaults to None. Defaults to None.
- **img_metas** (*list[list[dict]], optional*) – the outer list indicates test-time augs (multiscale, flip, etc.) and the inner list indicates images in a batch. Defaults to None.

- **rescale** (*bool, optional*) – Whether or not rescale bboxes to the original shape of input image. If rescale is False, then returned bboxes and masks will fit the scale of imgs[0]. Defaults to None.

Returns Predicted 2d boxes.

Return type list[list[torch.Tensor]]

extract_bboxes_2d(*img*, *img_metas*, *train=True*, *bboxes_2d=None*, ***kwargs*)
Extract bounding boxes from 2d detector.

Parameters

- **img** (*torch.Tensor*) – of shape (N, C, H, W) encoding input images. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – Image meta info.
- **train** (*bool*) – train-time or not.
- **bboxes_2d** (*list[torch.Tensor]*) – provided 2d bboxes, not supported yet.

Returns a list of processed 2d bounding boxes.

Return type list[torch.Tensor]

extract_feat(*imgs*)
Just to inherit from abstract method.

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

extract_img_feats(*imgs*)
Extract features from multiple images.

Parameters **imgs** (*list[torch.Tensor]*) – A list of images. The images are augmented from the same image but in different ways.

Returns Features of different images

Return type list[torch.Tensor]

extract_pts_feat(*pts*)
Extract features of points.

extract_pts_feats(*pts*)
Extract features of points from multiple samples.

forward_test(*points=None*, *img_metas=None*, *img=None*, *bboxes_2d=None*, ***kwargs*)
Forwarding of test for image branch pretrain or stage 2 train.

Parameters

- **points** (*list[list[torch.Tensor]], optional*) – the outer list indicates test-time augmentations and the inner list contains all points in the batch, where each Tensor should have a shape NxC. Defaults to None.
- **img_metas** (*list[list[dict]], optional*) – the outer list indicates test-time augs (multiscale, flip, etc.) and the inner list indicates images in a batch. Defaults to None.
- **img** (*list[list[torch.Tensor]], optional*) – the outer list indicates test-time augmentations and inner Tensor should have a shape NxCxHxW, which contains all images in the batch. Defaults to None. Defaults to None.

- **bboxes_2d** (*list[list[torch.Tensor]], optional*) – Provided 2d bboxes, not supported yet. Defaults to None.

Returns Predicted 2d or 3d boxes.

Return type list[list[torch.Tensor]]|list[dict]

forward_train(*points=None*, *img=None*, *img_metas=None*, *gt_bboxes=None*, *gt_labels=None*, *gt_bboxes_ignore=None*, *gt_masks=None*, *proposals=None*, *bboxes_2d=None*, *gt_bboxes_3d=None*, *gt_labels_3d=None*, *pts_semantic_mask=None*, *pts_instance_mask=None*, ***kwargs*)

Forwarding of train for image branch pretrain or stage 2 train.

Parameters

- **points** (*list[torch.Tensor]*) – Points of each batch.
- **img** (*torch.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 and point cloud meta info dict. For example, keys include 'ori_shape', 'img_norm_cfg', and 'transformation_3d_flow'. For details on the values of the keys see *mmdet/datasets/pipelines/formatting.py:Collect*.
- **gt_bboxes** (*list[torch.Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **gt_labels** (*list[torch.Tensor]*) – class indices for each 2d bounding box.
- **gt_bboxes_ignore** (*list[torch.Tensor]*) – specify which 2d bounding boxes can be ignored when computing the loss.
- **gt_masks** (*torch.Tensor*) – true segmentation masks for each 2d bbox, used if the architecture supports a segmentation task.
- **proposals** – override rpn proposals (2d) with custom proposals. Use when *with_rpn* is False.
- **bboxes_2d** (*list[torch.Tensor]*) – provided 2d bboxes, not supported yet.
- **gt_bboxes_3d** (BaseInstance3DBoxes) – 3d gt bboxes.
- **gt_labels_3d** (*list[torch.Tensor]*) – gt class labels for 3d bboxes.
- **pts_semantic_mask** (*list[torch.Tensor]*) – point-wise semantic label of each batch.
- **pts_instance_mask** (*list[torch.Tensor]*) – point-wise instance label of each batch.

Returns a dictionary of loss components.

Return type dict[str, torch.Tensor]

freeze_img_branch_params()

Freeze all image branch parameters.

simple_test(*points=None*, *img_metas=None*, *img=None*, *bboxes_2d=None*, *rescale=False*, ***kwargs*)

Test without augmentation, stage 2.

Parameters

- **points** (*list[torch.Tensor], optional*) – Elements in the list should have a shape NxC, the list indicates all point-clouds in the batch. Defaults to None.

- **img_metas** (*list[dict], optional*) – List indicates images in a batch. Defaults to None.
- **img** (*torch.Tensor, optional*) – Should have a shape NxCxHxW, which contains all images in the batch. Defaults to None.
- **bboxes_2d** (*list[torch.Tensor], optional*) – Provided 2d bboxes, not supported yet. Defaults to None.
- **rescale** (*bool, optional*) – Whether or not rescale bboxes. Defaults to False.

Returns Predicted 3d boxes.

Return type list[dict]

simple_test_img_only(*img*, *img_metas*, *proposals=None*, *rescale=False*)

Test without augmentation, image network pretrain. May refer to https://github.com/open-mmlab/mmdetection/blob/master/mmdet/models/detectors/two_stage.py.

Parameters

- **img** (*torch.Tensor*) – Should have a shape NxCxHxW, which contains all images in the batch.
- **img_metas** (*list[dict]*) –
- **proposals** (*list[Tensor], optional*) – override rpn proposals with custom proposals. Defaults to None.
- **rescale** (*bool, optional*) – Whether or not rescale bboxes to the original shape of input image. Defaults to False.

Returns Predicted 2d boxes.

Return type list[list[torch.Tensor]]

train(*mode=True*)

Overload in order to keep image branch modules in eval mode.

property with_img_backbone

Whether the detector has a 2D image backbone.

Type bool

property with_img_bbox

Whether the detector has a 2D image box head.

Type bool

property with_img_bbox_head

Whether the detector has a 2D image box head (not roi).

Type bool

property with_img_neck

Whether the detector has a neck in image branch.

Type bool

property with_img_roi_head

Whether the detector has a RoI Head in image branch.

Type bool

property with_img_rpn

Whether the detector has a 2D RPN in image detector branch.

Type bool

property with_pts_backbone
Whether the detector has a 3D backbone.

Type bool

property with_pts_bbox
Whether the detector has a 3D box head.

Type bool

property with_pts_neck
Whether the detector has a neck in 3D detector branch.

Type bool

class mmdet3d.models.detectors.**ImVoxelNet**(*backbone*, *neck*, *neck_3d*, *bbox_head*, *prior_generator*, *n_voxels*, *coord_type*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, *pretrained=None*)

ImVoxelNet.

Parameters

- **backbone** (*dict*) – Config of the backbone.
- **neck** (*dict*) – Config of the 2d neck.
- **neck_3d** (*dict*) – Config of the 3d neck.
- **bbox_head** (*dict*) – Config of the head.
- **prior_generator** (*dict*) – Config of the prior generator.
- **n_voxels** (*tuple[int]*) – Number of voxels for x, y, and z axis.
- **coord_type** (*str*) – The type of coordinates of points cloud: 'DEPTH', 'LIDAR', or 'CAMERA'.
- **train_cfg** (*dict, optional*) – Config for train stage. Defaults to None.
- **test_cfg** (*dict, optional*) – Config for test stage. Defaults to None.
- **init_cfg** (*dict, optional*) – Config for weight initialization. Defaults to None.
- **pretrained** (*str, optional*) – Deprecated initialization parameter. Defaults to None.

aug_test(*imgs*, *img_metas*, ***kwargs*)
Test with augmentations.

Parameters

- **imgs** (*list[torch.Tensor]*) – Input images of shape (N, C_in, H, W).
- **img_metas** (*list*) – Image metas.

Returns Predicted 3d boxes.

Return type list[dict]

extract_feat(*img*, *img_metas*)
Extract 3d features from the backbone -> fpn -> 3d projection.

-> 3d neck -> bbox_head.

Parameters

- **img** (*torch.Tensor*) – Input images of shape (N, C_in, H, W).

- **img_metas** (*list*) – Image metas.

Returns

- torch.Tensor: Features of shape (N, C_out, N_x, N_y, N_z).
- torch.Tensor: Valid mask of shape (N, 1, N_x, N_y, N_z).

Return type Tuple

forward_test(*img*, *img_metas*, ***kwargs*)
Forward of testing.

Parameters

- **img** (*torch.Tensor*) – Input images of shape (N, C_in, H, W).
- **img_metas** (*list*) – Image metas.

Returns Predicted 3d boxes.

Return type list[dict]

forward_train(*img*, *img_metas*, *gt_bboxes_3d*, *gt_labels_3d*, ***kwargs*)
Forward of training.

Parameters

- **img** (*torch.Tensor*) – Input images of shape (N, C_in, H, W).
- **img_metas** (*list*) – Image metas.
- **gt_bboxes_3d** (BaseInstance3DBoxes) – gt bboxes of each batch.
- **gt_labels_3d** (*list[torch.Tensor]*) – gt class labels of each batch.

Returns A dictionary of loss components.

Return type dict[str, torch.Tensor]

simple_test(*img*, *img_metas*)
Test without augmentations.

Parameters

- **img** (*torch.Tensor*) – Input images of shape (N, C_in, H, W).
- **img_metas** (*list*) – Image metas.

Returns Predicted 3d boxes.

Return type list[dict]

class mmdet3d.models.detectors.**MVXFasterRCNN**(***kwargs*)
Multi-modality VoxelNet using Faster R-CNN.

class mmdet3d.models.detectors.**MVXTwoStageDetector**(*pts_voxel_layer=None*, *pts_voxel_encoder=None*, *pts_middle_encoder=None*, *pts_fusion_layer=None*, *img_backbone=None*, *pts_backbone=None*, *img_neck=None*, *pts_neck=None*, *pts_bbox_head=None*, *img_roi_head=None*, *img_rpn_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)
Base class of Multi-modality VoxelNet.

aug_test(*points*, *img_metas*, *imgs=None*, *rescale=False*)
Test function with augmentaiton.

aug_test_pts(*feats*, *img_metas*, *rescale=False*)
Test function of point cloud branch with augmentaiton.

extract_feat(*points*, *img*, *img_metas*)
Extract features from images and points.

extract_feats(*points*, *img_metas*, *imgs=None*)
Extract point and image features of multiple samples.

extract_img_feat(*img*, *img_metas*)
Extract features of images.

extract_pts_feat(*pts*, *img_feats*, *img_metas*)
Extract features of points.

forward_img_train(*x*, *img_metas*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore=None*, *proposals=None*, ***kwargs*)
Forward function for image branch.

This function works similar to the forward function of Faster R-CNN.

Parameters

- **x** (*list[torch.Tensor]*) – Image features of shape (B, C, H, W) of multiple levels.
- **img_metas** (*list[dict]*) – Meta information of images.
- **gt_bboxes** (*list[torch.Tensor]*) – Ground truth boxes of each image sample.
- **gt_labels** (*list[torch.Tensor]*) – Ground truth labels of boxes.
- **gt_bboxes_ignore** (*list[torch.Tensor], optional*) – Ground truth boxes to be ignored. Defaults to None.
- **proposals** (*list[torch.Tensor], optional*) – Proposals of each sample. Defaults to None.

Returns Losses of each branch.

Return type dict

forward_pts_train(*pts_feats*, *gt_bboxes_3d*, *gt_labels_3d*, *img_metas*, *gt_bboxes_ignore=None*)
Forward function for point cloud branch.

Parameters

- **pts_feats** (*list[torch.Tensor]*) – Features of point cloud branch
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth boxes for each sample.
- **gt_labels_3d** (*list[torch.Tensor]*) – Ground truth labels for boxes of each sampole
- **img_metas** (*list[dict]*) – Meta information of samples.
- **gt_bboxes_ignore** (*list[torch.Tensor], optional*) – Ground truth boxes to be ignored. Defaults to None.

Returns Losses of each branch.

Return type dict

forward_train(*points=None*, *img_metas=None*, *gt_bboxes_3d=None*, *gt_labels_3d=None*, *gt_labels=None*, *gt_bboxes=None*, *img=None*, *proposals=None*, *gt_bboxes_ignore=None*)

Forward training function.

Parameters

- **points** (*list[torch.Tensor], optional*) – Points of each sample. Defaults to None.
- **img_metas** (*list[dict], optional*) – Meta information of each sample. Defaults to None.
- **gt_bboxes_3d** (list[BaseInstance3DBoxes], optional) – Ground truth 3D boxes. Defaults to None.
- **gt_labels_3d** (*list[torch.Tensor], optional*) – Ground truth labels of 3D boxes. Defaults to None.
- **gt_labels** (*list[torch.Tensor], optional*) – Ground truth labels of 2D boxes in images. Defaults to None.
- **gt_bboxes** (*list[torch.Tensor], optional*) – Ground truth 2D boxes in images. Defaults to None.
- **img** (*torch.Tensor, optional*) – Images of each sample with shape (N, C, H, W). Defaults to None.
- **proposals** (*[list[torch.Tensor], optional*) – Predicted proposals used for training Fast RCNN. Defaults to None.
- **gt_bboxes_ignore** (*list[torch.Tensor], optional*) – Ground truth 2D boxes in images to be ignored. Defaults to None.

Returns Losses of different branches.

Return type dict

show_results(*data*, *result*, *out_dir*)

Results visualization.

Parameters

- **data** (*dict*) – Input points and the information of the sample.
- **result** (*dict*) – Prediction results.
- **out_dir** (*str*) – Output directory of visualization result.

simple_test(*points*, *img_metas*, *img=None*, *rescale=False*)

Test function without augmentaiton.

simple_test_img(*x*, *img_metas*, *proposals=None*, *rescale=False*)

Test without augmentation.

simple_test_pts(*x*, *img_metas*, *rescale=False*)

Test function of point cloud branch.

simple_test_rpn(*x*, *img_metas*, *rpn_test_cfg*)

RPN test function.

voxelize(*points*)

Apply dynamic voxelization to points.

Parameters **points** (*list[torch.Tensor]*) – Points of each sample.

Returns

Concatenated points, number of points per voxel, and coordinates.

Return type tuple[torch.Tensor]

property with_fusion
Whether the detector has a fusion layer.

Type bool

property with_img_backbone
Whether the detector has a 2D image backbone.

Type bool

property with_img_bbox
Whether the detector has a 2D image box head.

Type bool

property with_img_neck
Whether the detector has a neck in image branch.

Type bool

property with_img_roi_head
Whether the detector has a RoI Head in image branch.

Type bool

property with_img_rpn
Whether the detector has a 2D RPN in image detector branch.

Type bool

property with_img_shared_head
Whether the detector has a shared head in image branch.

Type bool

property with_middle_encoder
Whether the detector has a middle encoder.

Type bool

property with_pts_backbone
Whether the detector has a 3D backbone.

Type bool

property with_pts_bbox
Whether the detector has a 3D box head.

Type bool

property with_pts_neck
Whether the detector has a neck in 3D detector branch.

Type bool

property with_voxel_encoder
Whether the detector has a voxel encoder.

Type bool

class mmdet3d.models.detectors.**MinkSingleStage3DDetector**(*backbone*, *head*, *voxel_size*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, *pretrained=None*)

Single stage detector based on MinkowskiEngine GSDN.

Parameters

- **backbone** (*dict*) – Config of the backbone.
- **head** (*dict*) – Config of the head.
- **voxel_size** (*float*) – Voxel size in meters.
- **train_cfg** (*dict, optional*) – Config for train stage. Defaults to None.
- **test_cfg** (*dict, optional*) – Config for test stage. Defaults to None.
- **init_cfg** (*dict, optional*) – Config for weight initialization. Defaults to None.
- **pretrained** (*str, optional*) – Deprecated initialization parameter. Defaults to None.

aug_test(*points*, *img_metas*, ***kwargs*)

Test with augmentations.

Parameters

- **points** (*list[list[torch.Tensor]]*) – Points of each sample.
- **img_metas** (*list[dict]*) – Contains scene meta infos.

Returns Predicted 3d boxes.

Return type list[dict]

extract_feat(*points*)

Extract features from points.

Parameters **points** (*list[Tensor]*) – Raw point clouds.

Returns Voxelized point clouds.

Return type SparseTensor

forward_train(*points*, *gt_bboxes_3d*, *gt_labels_3d*, *img_metas*)

Forward of training.

Parameters

- **points** (*list[Tensor]*) – Raw point clouds.
- **gt_bboxes** (*list[BaseInstance3DBoxes]*) – Ground truth bboxes of each sample.
- **gt_labels** (*list[torch.Tensor]*) – Labels of each sample.
- **img_metas** (*list[dict]*) – Contains scene meta infos.

Returns Centerness, bbox and classification loss values.

Return type dict

simple_test(*points*, *img_metas*, **args*, ***kwargs*)

Test without augmentations.

Parameters

- **points** (*list[torch.Tensor]*) – Points of each sample.
- **img_metas** (*list[dict]*) – Contains scene meta infos.

Returns Predicted 3d boxes.

Return type list[dict]

class mmdet3d.models.detectors.**PartA2**(*voxel_layer*, *voxel_encoder*, *middle_encoder*, *backbone*, *neck=None*, *rpn_head=None*, *roi_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)

Part-A2 detector.

Please refer to the paper

extract_feat(*points*, *img_metas*)
Extract features from points.

forward_train(*points*, *img_metas*, *gt_bboxes_3d*, *gt_labels_3d*, *gt_bboxes_ignore=None*, *proposals=None*)
Training forward function.

Parameters

- **points** (*list[torch.Tensor]*) – Point cloud of each sample.
- **img_metas** (*list[dict]*) – Meta information of each sample
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth boxes for each sample.
- **gt_labels_3d** (*list[torch.Tensor]*) – Ground truth labels for boxes of each sampole
- **gt_bboxes_ignore** (*list[torch.Tensor], optional*) – Ground truth boxes to be ignored. Defaults to None.

Returns Losses of each branch.

Return type dict

simple_test(*points*, *img_metas*, *proposals=None*, *rescale=False*)
Test function without augmentaiton.

voxelize(*points*)
Apply hard voxelization to points.

class mmdet3d.models.detectors.**PointRCNN**(*backbone*, *neck=None*, *rpn_head=None*, *roi_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)

PointRCNN detector.

Please refer to the PointRCNN

Parameters

- **backbone** (*dict*) – Config dict of detector's backbone.
- **neck** (*dict, optional*) – Config dict of neck. Defaults to None.
- **rpn_head** (*dict, optional*) – Config of RPN head. Defaults to None.
- **roi_head** (*dict, optional*) – Config of ROI head. Defaults to None.
- **train_cfg** (*dict, optional*) – Train configs. Defaults to None.
- **test_cfg** (*dict, optional*) – Test configs. Defaults to None.
- **pretrained** (*str, optional*) – Model pretrained path. Defaults to None.
- **init_cfg** (*dict, optional*) – Config of initialization. Defaults to None.

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

Parameters **points** (*torch.Tensor*) – Input points.

Returns Features from the backbone+neck

Return type dict

forward_train(*points*, *img_metas*, *gt_bboxes_3d*, *gt_labels_3d*)
Forward of training.

Parameters

- **points** (*list[torch.Tensor]*) – Points of each batch.
- **img_metas** (*list[dict]*) – Meta information of each sample.
- **gt_bboxes_3d** (BaseInstance3DBoxes) – gt bboxes of each batch.
- **gt_labels_3d** (*list[torch.Tensor]*) – gt class labels of each batch.

Returns Losses.

Return type dict

simple_test(*points*, *img_metas*, *imgs=None*, *rescale=False*)
Forward of testing.

Parameters

- **points** (*list[torch.Tensor]*) – Points of each sample.
- **img_metas** (*list[dict]*) – Image metas.
- **imgs** (*list[torch.Tensor], optional*) – Images of each sample. Defaults to None.
- **rescale** (*bool, optional*) – Whether to rescale results. Defaults to False.

Returns Predicted 3d boxes.

Return type list

class mmdet3d.models.detectors.**SASSD**(*voxel_layer*, *voxel_encoder*, *middle_encoder*, *backbone*, *neck=None*, *bbox_head=None*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, *pretrained=None*)
SASSD <https://github.com/skyhehe123/SA-SSD> _ for 3D detection.

aug_test(*points*, *img_metas*, *imgs=None*, *rescale=False*)
Test function with augmentaiton.

extract_feat(*points*, *img_metas=None*, *test_mode=False*)
Extract features from points.

forward_train(*points*, *img_metas*, *gt_bboxes_3d*, *gt_labels_3d*, *gt_bboxes_ignore=None*)
Training forward function.

Parameters

- **points** (*list[torch.Tensor]*) – Point cloud of each sample.
- **img_metas** (*list[dict]*) – Meta information of each sample
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth boxes for each sample.
- **gt_labels_3d** (*list[torch.Tensor]*) – Ground truth labels for boxes of each sampole

- **gt_bboxes_ignore** (*list[torch.Tensor], optional*) – Ground truth boxes to be ignored. Defaults to None.

Returns Losses of each branch.

Return type dict

simple_test(*points*, *img_metas*, *imgs=None*, *rescale=False*)
Test function without augmentaiton.

voxelize(*points*)
Apply hard voxelization to points.

class mmdet3d.models.detectors.**SMOKEMono3D**(*backbone*, *neck*, *bbox_head*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*)
SMOKE <https://arxiv.org/abs/2002.10111>`_ for monocular 3D object detection.

class mmdet3d.models.detectors.**SSD3DNet**(*backbone*, *bbox_head=None*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, *pretrained=None*)
3DSSDNet model.

https://arxiv.org/abs/2002.10187.pdf

class mmdet3d.models.detectors.**SingleStageMono3DDetector**(*backbone*, *neck=None*, *bbox_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)
Base class for monocular 3D 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.

extract_feats(*imgs*)
Directly extract features from the backbone+neck.

forward_train(*img*, *img_metas*, *gt_bboxes*, *gt_labels*, *gt_bboxes_3d*, *gt_labels_3d*, *centers2d*, *depths*, *attr_labels=None*, *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_3d** (*list[Tensor]*) – Each item are the 3D truth boxes for each image in [x, y, z, x_size, y_size, z_size, yaw, vx, vy] format.
- **gt_labels_3d** (*list[Tensor]*) – 3D class indices corresponding to each box.
- **centers2d** (*list[Tensor]*) – Projected 3D centers onto 2D images.
- **depths** (*list[Tensor]*) – Depth of projected centers on 2D images.

- **attr_labels** (*list[Tensor], optional*) – Attribute indices corresponding to each box
- **gt_bboxes_ignore** (*list[Tensor]*) – Specify which bounding boxes can be ignored when computing the loss.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

show_results(*data*, *result*, *out_dir*, *show=False*, *score_thr=None*)
Results visualization.

Parameters

- **data** (*list[dict]*) – Input images and the information of the sample.
- **result** (*list[dict]*) – Prediction results.
- **out_dir** (*str*) – Output directory of visualization result.
- **show** (*bool, optional*) – Determines whether you are going to show result by open3d. Defaults to False.
- **TODO** – implement score_thr of single_stage_mono3d.
- **score_thr** (*float, optional*) – Score threshold of bounding boxes. Default to None. Not implemented yet, but it is here for unification.

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 mmdet3d.models.detectors.**VoteNet**(*backbone*, *bbox_head=None*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, *pretrained=None*)

VoteNet for 3D detection.

aug_test(*points*, *img_metas*, *imgs=None*, *rescale=False*)
Test with augmentation.

forward_train(*points*, *img_metas*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*, *gt_bboxes_ignore=None*)
Forward of training.

Parameters

- **points** (*list[torch.Tensor]*) – Points of each batch.
- **img_metas** (*list*) – Image metas.
- **gt_bboxes_3d** (BaseInstance3DBoxes) – gt bboxes of each batch.
- **gt_labels_3d** (*list[torch.Tensor]*) – gt class labels of each batch.

- **pts_semantic_mask** (*list[torch.Tensor]*) – point-wise semantic label of each batch.
- **pts_instance_mask** (*list[torch.Tensor]*) – point-wise instance label of each batch.
- **gt_bboxes_ignore** (*list[torch.Tensor]*) – Specify which bounding.

Returns Losses.

Return type dict

simple_test(*points*, *img_metas*, *imgs=None*, *rescale=False*)
Forward of testing.

Parameters

- **points** (*list[torch.Tensor]*) – Points of each sample.
- **img_metas** (*list*) – Image metas.
- **rescale** (*bool*) – Whether to rescale results.

Returns Predicted 3d boxes.

Return type list

class mmdet3d.models.detectors.**VoxelNet**(*voxel_layer*, *voxel_encoder*, *middle_encoder*, *backbone*, *neck=None*, *bbox_head=None*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, *pretrained=None*)
VoxelNet for 3D detection.

aug_test(*points*, *img_metas*, *imgs=None*, *rescale=False*)
Test function with augmentaiton.

extract_feat(*points*, *img_metas=None*)
Extract features from points.

forward_train(*points*, *img_metas*, *gt_bboxes_3d*, *gt_labels_3d*, *gt_bboxes_ignore=None*)
Training forward function.

Parameters

- **points** (*list[torch.Tensor]*) – Point cloud of each sample.
- **img_metas** (*list[dict]*) – Meta information of each sample
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth boxes for each sample.
- **gt_labels_3d** (*list[torch.Tensor]*) – Ground truth labels for boxes of each sampole
- **gt_bboxes_ignore** (*list[torch.Tensor], optional*) – Ground truth boxes to be ignored. Defaults to None.

Returns Losses of each branch.

Return type dict

simple_test(*points*, *img_metas*, *imgs=None*, *rescale=False*)
Test function without augmentaiton.

voxelize(*points*)
Apply hard voxelization to points.

47.2 backbones

class mmdet3d.models.backbones.**DGCNNBackbone**(*in_channels*, *num_samples=(20, 20, 20)*, *knn_modes=('D-KNN', 'F-KNN', 'F-KNN')*, *radius=(None, None, None)*, *gf_channels=((64, 64), (64, 64), (64))*, *fa_channels=(1024)*, *act_cfg={'type': 'ReLU'}*, *init_cfg=None*)

Backbone network for DGCNN.

Parameters

- **in_channels** (*int*) – Input channels of point cloud.
- **num_samples** (*tuple[int], optional*) – The number of samples for knn or ball query in each graph feature (GF) module. Defaults to (20, 20, 20).
- **knn_modes** (*tuple[str], optional*) – Mode of KNN of each knn module. Defaults to ('D-KNN', 'F-KNN', 'F-KNN').
- **radius** (*tuple[float], optional*) – Sampling radii of each GF module. Defaults to (None, None, None).
- **gf_channels** (*tuple[tuple[int]], optional*) – Out channels of each mlp in GF module. Defaults to ((64, 64), (64, 64), (64,)).
- **fa_channels** (*tuple[int], optional*) – Out channels of each mlp in FA module. Defaults to (1024,).
- **act_cfg** (*dict, optional*) – Config of activation layer. Defaults to dict(type='ReLU').
- **init_cfg** (*dict, optional*) – Initialization config. Defaults to None.

forward(*points*)

Forward pass.

Parameters **points** (*torch.Tensor*) – point coordinates with features, with shape (B, N, in_channels).

Returns

Outputs after graph feature (GF) and feature aggregation (FA) modules.

- gf_points (list[torch.Tensor]): Outputs after each GF module.
- fa_points (torch.Tensor): Outputs after FA module.

Return type dict[str, list[torch.Tensor]]

class mmdet3d.models.backbones.**DLANet**(*depth*, *in_channels=3*, *out_indices=(0, 1, 2, 3, 4, 5)*, *frozen_stages=- 1*, *norm_cfg=None*, *conv_cfg=None*, *layer_with_level_root=(False, True, True, True)*, *with_identity_root=False*, *pretrained=None*, *init_cfg=None*)

DLA backbone.

Parameters

- **depth** (*int*) – Depth of DLA. Default: 34.
- **in_channels** (*int, optional*) – Number of input image channels. Default: 3.
- **norm_cfg** (*dict, optional*) – Dictionary to construct and config norm layer. Default: None.

- **conv_cfg** (*dict, optional*) – Dictionary to construct and config conv layer. Default: None.
- **layer_with_level_root** (*list[bool], optional*) – Whether to apply level_root in each DLA layer, this is only used for tree levels. Default: (False, True, True, True).
- **with_identity_root** (*bool, optional*) – Whether to add identity in root layer. Default: False.
- **pretrained** (*str, optional*) – model pretrained path. Default: None.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Default: None

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmdet3d.models.backbones.**HRNet**(*extra*, *in_channels=3*, *conv_cfg=None*, *norm_cfg={'type': 'BN'}*, *norm_eval=True*, *with_cp=False*, *zero_init_residual=False*, *multiscale_output=True*, *pretrained=None*, *init_cfg=None*)

HRNet backbone.

High-Resolution Representations for Labeling Pixels and Regions arXiv:.

Parameters

- **extra** (*dict*) – Detailed configuration for each stage of HRNet. There must be 4 stages, the configuration for each stage must have 5 keys:
  - num_modules(int): The number of HRModule in this stage.
  - num_branches(int): The number of branches in the HRModule.
  - block(str): The type of convolution block.
  - **num_blocks(tuple): The number of blocks in each branch.** The length must be equal to num_branches.
  - **num_channels(tuple): The number of channels in each branch.** The length must be equal to num_branches.
- **in_channels** (*int*) – Number of input image channels. Default: 3.
- **conv_cfg** (*dict*) – Dictionary to construct and config conv layer.
- **norm_cfg** (*dict*) – Dictionary to construct and config norm layer.
- **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: True.
- **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: False.

- **multiscale_output** (*bool*) – Whether to output multi-level features produced by multiple branches. If False, only the first level feature will be output. Default: True.
- **pretrained** (*str, optional*) – Model pretrained path. Default: None.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Default: None.

Example

```
>>> from mmdet.models import HRNet
>>> import torch
>>> extra = dict(
>>>     stage1=dict(
>>>         num_modules=1,
>>>         num_branches=1,
>>>         block='BOTTLENECK',
>>>         num_blocks=(4, ),
>>>         num_channels=(64, )),
>>>     stage2=dict(
>>>         num_modules=1,
>>>         num_branches=2,
>>>         block='BASIC',
>>>         num_blocks=(4, 4),
>>>         num_channels=(32, 64)),
>>>     stage3=dict(
>>>         num_modules=4,
>>>         num_branches=3,
>>>         block='BASIC',
>>>         num_blocks=(4, 4, 4),
>>>         num_channels=(32, 64, 128)),
>>>     stage4=dict(
>>>         num_modules=3,
>>>         num_branches=4,
>>>         block='BASIC',
>>>         num_blocks=(4, 4, 4, 4),
>>>         num_channels=(32, 64, 128, 256)))
>>> self = HRNet(extra, in_channels=1)
>>> self.eval()
>>> inputs = torch.rand(1, 1, 32, 32)
>>> level_outputs = self.forward(inputs)
>>> for level_out in level_outputs:
...     print(tuple(level_out.shape))
(1, 32, 8, 8)
(1, 64, 4, 4)
(1, 128, 2, 2)
(1, 256, 1, 1)
```

forward(*x*)

Forward function.

property norm1

the normalization layer named “norm1”

Type nn.Module

property norm2
the normalization layer named "norm2"

Type nn.Module

train(*mode=True*)
Convert the model into training mode will keeping the normalization layer freezed.

class mmdet3d.models.backbones.**MinkResNet**(*depth*, *in_channels*, *num_stages=4*, *pool=True*)
Minkowski ResNet backbone. See 4D Spatio-Temporal ConvNets for more details.

Parameters

- **depth** (*int*) – Depth of resnet, from {18, 34, 50, 101, 152}.
- **in_channels** (*ont*) – Number of input channels, 3 for RGB.
- **num_stages** (*int, optional*) – Resnet stages. Default: 4.
- **pool** (*bool, optional*) – Add max pooling after first conv if True. Default: True.

forward(*x*)
Forward pass of ResNet.

Parameters **x** (*ME.SparseTensor*) – Input sparse tensor.

Returns Output sparse tensors.

Return type list[ME.SparseTensor]

class mmdet3d.models.backbones.**MultiBackbone**(*num_streams*, *backbones*, *aggregation_mlp_channels=None*, *conv_cfg={'type': 'Conv1d'}*, *norm_cfg={'eps': 1e-05, 'momentum': 0.01, 'type': 'BN1d'}*, *act_cfg={'type': 'ReLU'}*, *suffixes=('net0', 'net1')*, *init_cfg=None*, *pretrained=None*, ***kwargs*)
MultiBackbone with different configs.

Parameters

- **num_streams** (*int*) – The number of backbones.
- **backbones** (*list or dict*) – A list of backbone configs.
- **aggregation_mlp_channels** (*list[int]*) – Specify the mlp layers for feature aggregation.
- **conv_cfg** (*dict*) – Config dict of convolutional layers.
- **norm_cfg** (*dict*) – Config dict of normalization layers.
- **act_cfg** (*dict*) – Config dict of activation layers.
- **suffixes** (*list*) – A list of suffixes to rename the return dict for each backbone.

forward(*points*)
Forward pass.

Parameters **points** (*torch.Tensor*) – point coordinates with features, with shape (B, N, 3 + input_feature_dim).

Returns

Outputs from multiple backbones.

- fp_xyz[suffix] (list[torch.Tensor]): The coordinates of each fp features.

- fp_features[suffix] (list[torch.Tensor]): The features from each Feature Propagate Layers.
- fp_indices[suffix] (list[torch.Tensor]): Indices of the input points.
- hd_feature (torch.Tensor): The aggregation feature from multiple backbones.

Return type dict[str, list[torch.Tensor]]

class mmdet3d.models.backbones.**NoStemRegNet**(*arch*, *init_cfg=None*, ***kwargs*)

RegNet backbone without Stem for 3D detection.

More details can be found in paper .

Parameters

- **arch** (*dict*) – The parameter of RegNets. - w0 (int): Initial width. - wa (float): Slope of width. - wm (float): Quantization parameter to quantize the width. - depth (int): Depth of the backbone. - group_w (int): Width of group. - bot_mul (float): Bottleneck ratio, i.e. expansion of bottleneck.
- **strides** (*Sequence[int]*) – Strides of the first block of each stage.
- **base_channels** (*int*) – Base channels after stem layer.
- **in_channels** (*int*) – Number of input image channels. Normally 3.
- **dilations** (*Sequence[int]*) – Dilation of each stage.
- **out_indices** (*Sequence[int]*) – Output from which stages.
- **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.
- **frozen_stages** (*int*) – Stages to be frozen (all param fixed). -1 means not freezing any parameters.
- **norm_cfg** (*dict*) – Dictionary to construct and config norm layer.
- **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.
- **with_cp** (*bool*) – Use checkpoint or not. Using checkpoint will save some memory while slowing down the training speed.
- **zero_init_residual** (*bool*) – Whether to use zero init for last norm layer in resblocks to let them behave as identity.

Example

```
>>> from mmdet3d.models import NoStemRegNet
>>> import torch
>>> self = NoStemRegNet(
        arch=dict(
            w0=88,
            wa=26.31,
            wm=2.25,
            group_w=48,
            depth=25,
            bot_mul=1.0))
>>> self.eval()
```

(continues on next page)

(continued from previous page)

```
>>> inputs = torch.rand(1, 64, 16, 16)
>>> level_outputs = self.forward(inputs)
>>> for level_out in level_outputs:
...     print(tuple(level_out.shape))
(1, 96, 8, 8)
(1, 192, 4, 4)
(1, 432, 2, 2)
(1, 1008, 1, 1)
```

forward(*x*)

Forward function of backbone.

Parameters **x** (*torch.Tensor*) – Features in shape (N, C, H, W).

Returns Multi-scale features.

Return type tuple[torch.Tensor]

class mmdet3d.models.backbones.**PointNet2SAMSG**(*in_channels*, *num_points=(2048, 1024, 512, 256)*, *radii=((0.2, 0.4, 0.8), (0.4, 0.8, 1.6), (1.6, 3.2, 4.8))*, *num_samples=((32, 32, 64), (32, 32, 64), (32, 32, 32))*, *sa_channels=(((16, 16, 32), (16, 16, 32), (32, 32, 64)), ((64, 64, 128), (64, 64, 128), (64, 96, 128)), ((128, 128, 256), (128, 192, 256), (128, 256, 256)))*, *aggregation_channels=(64, 128, 256)*, *fps_mods=('D-FPS', 'FS', ('F-FPS', 'D-FPS'))*, *fps_sample_range_lists=(- 1, - 1, (512, - 1))*, *dilated_group=(True, True, True)*, *out_indices=(2)*, *norm_cfg={'type': 'BN2d'}*, *sa_cfg={'normalize_xyz': False, 'pool_mod': 'max', 'type': 'PointSAModuleMSG', 'use_xyz': True}*, *init_cfg=None*)

PointNet2 with Multi-scale grouping.

Parameters

- **in_channels** (*int*) – Input channels of point cloud.
- **num_points** (*tuple[int]*) – The number of points which each SA module samples.
- **radii** (*tuple[float]*) – Sampling radii of each SA module.
- **num_samples** (*tuple[int]*) – The number of samples for ball query in each SA module.
- **sa_channels** (*tuple[tuple[int]]*) – Out channels of each mlp in SA module.
- **aggregation_channels** (*tuple[int]*) – Out channels of aggregation multi-scale grouping features.
- **fps_mods** (*tuple[int]*) – Mod of FPS for each SA module.
- **fps_sample_range_lists** (*tuple[tuple[int]]*) – The number of sampling points which each SA module samples.
- **dilated_group** (*tuple[bool]*) – Whether to use dilated ball query for
- **out_indices** (*Sequence[int]*) – Output from which stages.
- **norm_cfg** (*dict*) – Config of normalization layer.
- **sa_cfg** (*dict*) – Config of set abstraction module, which may contain the following keys and values:

- pool_mod (str): Pool method ('max' or 'avg') for SA modules.
- use_xyz (bool): Whether to use xyz as a part of features.
- normalize_xyz (bool): Whether to normalize xyz with radii in each SA module.

forward(*points*)

Forward pass.

Parameters **points** (*torch.Tensor*) – point coordinates with features, with shape (B, N, 3 + input_feature_dim).

Returns

Outputs of the last SA module.

- sa_xyz (torch.Tensor): The coordinates of sa features.
- **sa_features (torch.Tensor): The features from the** last Set Aggregation Layers.
- **sa_indices (torch.Tensor): Indices of the** input points.

Return type dict[str, torch.Tensor]

class mmdet3d.models.backbones.**PointNet2SASSG**(*in_channels*, *num_points=(2048, 1024, 512, 256)*, *radius=(0.2, 0.4, 0.8, 1.2)*, *num_samples=(64, 32, 16, 16)*, *sa_channels=((64, 64, 128), (128, 128, 256), (128, 128, 256), (128, 128, 256))*, *fp_channels=((256, 256), (256, 256))*, *norm_cfg={'type': 'BN2d'}*, *sa_cfg={'normalize_xyz': True, 'pool_mod': 'max', 'type': 'PointSAModule', 'use_xyz': True}*, *init_cfg=None*)

PointNet2 with Single-scale grouping.

Parameters

- **in_channels** (*int*) – Input channels of point cloud.
- **num_points** (*tuple[int]*) – The number of points which each SA module samples.
- **radius** (*tuple[float]*) – Sampling radii of each SA module.
- **num_samples** (*tuple[int]*) – The number of samples for ball query in each SA module.
- **sa_channels** (*tuple[tuple[int]]*) – Out channels of each mlp in SA module.
- **fp_channels** (*tuple[tuple[int]]*) – Out channels of each mlp in FP module.
- **norm_cfg** (*dict*) – Config of normalization layer.
- **sa_cfg** (*dict*) – Config of set abstraction module, which may contain the following keys and values:
  - pool_mod (str): Pool method ('max' or 'avg') for SA modules.
  - use_xyz (bool): Whether to use xyz as a part of features.
  - normalize_xyz (bool): Whether to normalize xyz with radii in each SA module.

forward(*points*)

Forward pass.

Parameters **points** (*torch.Tensor*) – point coordinates with features, with shape (B, N, 3 + input_feature_dim).

Returns

Outputs after SA and FP modules.

- **fp_xyz (list[torch.Tensor]): The coordinates of** each fp features.
- **fp_features (list[torch.Tensor]): The features** from each Feature Propagate Layers.
- **fp_indices (list[torch.Tensor]): Indices of the** input points.

Return type dict[str, list[torch.Tensor]]

class mmdet3d.models.backbones.**ResNeXt**(*groups=1*, *base_width=4*, ***kwargs*)

ResNeXt backbone.

Parameters

- **depth** (*int*) – Depth of resnet, from {18, 34, 50, 101, 152}.
- **in_channels** (*int*) – Number of input image channels. Default: 3.
- **num_stages** (*int*) – Resnet stages. Default: 4.
- **groups** (*int*) – Group of resnext.
- **base_width** (*int*) – Base width of resnext.
- **strides** (*Sequence[int]*) – Strides of the first block of each stage.
- **dilations** (*Sequence[int]*) – Dilation of each stage.
- **out_indices** (*Sequence[int]*) – Output from which stages.
- **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.
- **frozen_stages** (*int*) – Stages to be frozen (all param fixed). -1 means not freezing any parameters.
- **norm_cfg** (*dict*) – dictionary to construct and config norm layer.
- **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.
- **with_cp** (*bool*) – Use checkpoint or not. Using checkpoint will save some memory while slowing down the training speed.
- **zero_init_residual** (*bool*) – whether to use zero init for last norm layer in resblocks to let them behave as identity.

make_res_layer(***kwargs*)

Pack all blocks in a stage into a `ResLayer`

class mmdet3d.models.backbones.**ResNet**(*depth*, *in_channels=3*, *stem_channels=None*, *base_channels=64*, *num_stages=4*, *strides=(1, 2, 2, 2)*, *dilations=(1, 1, 1, 1)*, *out_indices=(0, 1, 2, 3)*, *style='pytorch'*, *deep_stem=False*, *avg_down=False*, *frozen_stages=- 1*, *conv_cfg=None*, *norm_cfg={'requires_grad': True, 'type': 'BN'}*, *norm_eval=True*, *dcn=None*, *stage_with_dcn=(False, False, False, False)*, *plugins=None*, *with_cp=False*, *zero_init_residual=True*, *pretrained=None*, *init_cfg=None*)

ResNet backbone.

Parameters

- **depth** (*int*) – Depth of resnet, from {18, 34, 50, 101, 152}.

- **stem_channels** (*int | None*) – Number of stem channels. If not specified, it will be the same as *base_channels*. Default: None.
- **base_channels** (*int*) – Number of base channels of res layer. Default: 64.
- **in_channels** (*int*) – Number of input image channels. Default: 3.
- **num_stages** (*int*) – Resnet stages. Default: 4.
- **strides** (*Sequence[int]*) – Strides of the first block of each stage.
- **dilations** (*Sequence[int]*) – Dilation of each stage.
- **out_indices** (*Sequence[int]*) – Output from which stages.
- **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
- **avg_down** (*bool*) – Use AvgPool instead of stride conv when downsampling in the bottleneck.
- **frozen_stages** (*int*) – Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters.
- **norm_cfg** (*dict*) – Dictionary to construct and config norm layer.
- **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.
- **plugins** (*list[dict]*) – List of plugins for stages, each dict contains:
  - cfg (dict, required): Cfg dict to build plugin.
  - position (str, required): Position inside block to insert plugin, options are 'after_conv1', 'after_conv2', 'after_conv3'.
  - stages (tuple[bool], optional): Stages to apply plugin, length should be same as 'num_stages'.
- **with_cp** (*bool*) – Use checkpoint or not. Using checkpoint will save some memory while slowing down the training speed.
- **zero_init_residual** (*bool*) – Whether to use zero init for last norm layer in resblocks to let them behave as identity.
- **pretrained** (*str, optional*) – model pretrained path. Default: None
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Default: None

Example

```
>>> from mmdet.models import ResNet
>>> import torch
>>> self = ResNet(depth=18)
>>> self.eval()
>>> inputs = torch.rand(1, 3, 32, 32)
>>> level_outputs = self.forward(inputs)
>>> for level_out in level_outputs:
...     print(tuple(level_out.shape))
```

(continues on next page)

(continued from previous page)

```
(1, 64, 8, 8)
(1, 128, 4, 4)
(1, 256, 2, 2)
(1, 512, 1, 1)
```

forward(*x*)
Forward function.

make_res_layer(***kwargs*)
Pack all blocks in a stage into a `ResLayer`.

make_stage_plugins(*plugins*, *stage_idx*)
Make plugins for ResNet `stage_idx` th stage.

Currently we support to insert `context_block`, `empirical_attention_block`, `nonlocal_block` into the backbone like ResNet/ResNeXt. They could be inserted after conv1/conv2/conv3 of Bottleneck.

An example of plugins format could be:

Examples

```
>>> plugins=[
...     dict(cfg=dict(type='xxx', arg1='xxx'),
...          stages=(False, True, True, True),
...          position='after_conv2'),
...     dict(cfg=dict(type='yyy'),
...          stages=(True, True, True, True),
...          position='after_conv3'),
...     dict(cfg=dict(type='zzz', postfix='1'),
...          stages=(True, True, True, True),
...          position='after_conv3'),
...     dict(cfg=dict(type='zzz', postfix='2'),
...          stages=(True, True, True, True),
...          position='after_conv3')
... ]
>>> self = ResNet(depth=18)
>>> stage_plugins = self.make_stage_plugins(plugins, 0)
>>> assert len(stage_plugins) == 3
```

Suppose `stage_idx=0`, the structure of blocks in the stage would be:

```
conv1-> conv2->conv3->yyy->zzz1->zzz2
```

Suppose 'stage_idx=1', the structure of blocks in the stage would be:

```
conv1-> conv2->xxx->conv3->yyy->zzz1->zzz2
```

If stages is missing, the plugin would be applied to all stages.

Parameters

- **plugins** (*list[dict]*) – List of plugins cfg to build. The postfix is required if multiple same type plugins are inserted.
- **stage_idx** (*int*) – Index of stage to build

Returns Plugins for current stage

Return type list[dict]

property norm1
the normalization layer named "norm1"

Type nn.Module

train(*mode=True*)
Convert the model into training mode while keep normalization layer freezed.

class mmdet3d.models.backbones.**ResNetV1d**(***kwargs*)
ResNetV1d variant described in Bag of Tricks.

Compared with default ResNet(ResNetV1b), ResNetV1d replaces the 7x7 conv in the input stem with three 3x3 convs. And in the downsampling block, a 2x2 avg_pool with stride 2 is added before conv, whose stride is changed to 1.

class mmdet3d.models.backbones.**SECOND**(*in_channels=128*, *out_channels=[128, 128, 256]*, *layer_nums=[3, 5, 5]*, *layer_strides=[2, 2, 2]*, *norm_cfg={'eps': 0.001, 'momentum': 0.01, 'type': 'BN'}*, *conv_cfg={'bias': False, 'type': 'Conv2d'}*, *init_cfg=None*, *pretrained=None*)
Backbone network for SECOND/PointPillars/PartA2/MVXNet.

Parameters

- **in_channels** (*int*) – Input channels.
- **out_channels** (*list[int]*) – Output channels for multi-scale feature maps.
- **layer_nums** (*list[int]*) – Number of layers in each stage.
- **layer_strides** (*list[int]*) – Strides of each stage.
- **norm_cfg** (*dict*) – Config dict of normalization layers.
- **conv_cfg** (*dict*) – Config dict of convolutional layers.

forward(*x*)
Forward function.

Parameters **x** (*torch.Tensor*) – Input with shape (N, C, H, W).

Returns Multi-scale features.

Return type tuple[torch.Tensor]

class mmdet3d.models.backbones.**SSDVGG**(*depth*, *with_last_pool=False*, *ceil_mode=True*, *out_indices=(3, 4)*, *out_feature_indices=(22, 34)*, *pretrained=None*, *init_cfg=None*, *input_size=None*, *l2_norm_scale=None*)
VGG Backbone network for single-shot-detection.

Parameters

- **depth** (*int*) – Depth of vgg, from {11, 13, 16, 19}.
- **with_last_pool** (*bool*) – Whether to add a pooling layer at the last of the model
- **ceil_mode** (*bool*) – When True, will use *ceil* instead of *floor* to compute the output shape.
- **out_indices** (*Sequence[int]*) – Output from which stages.
- **out_feature_indices** (*Sequence[int]*) – Output from which feature map.
- **pretrained** (*str, optional*) – model pretrained path. Default: None

- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Default: None
- **input_size** (*int, optional*) – Deprecated argumment. Width and height of input, from {300, 512}.
- **l2_norm_scale** (*float, optional*) – Deprecated argumment. L2 normalization layer init scale.

Example

```
>>> self = SSDVGG(input_size=300, depth=11)
>>> self.eval()
>>> inputs = torch.rand(1, 3, 300, 300)
>>> level_outputs = self.forward(inputs)
>>> for level_out in level_outputs:
...     print(tuple(level_out.shape))
(1, 1024, 19, 19)
(1, 512, 10, 10)
(1, 256, 5, 5)
(1, 256, 3, 3)
(1, 256, 1, 1)
```

forward(*x*)
Forward function.

init_weights(*pretrained=None*)
Initialize the weights.

47.3 necks

class mmdet3d.models.necks.**DLANeck**(*in_channels=[16, 32, 64, 128, 256, 512]*, *start_level=2*, *end_level=5*, *norm_cfg=None*, *use_dcn=True*, *init_cfg=None*)

DLA Neck.

Parameters

- **in_channels** (*list[int], optional*) – List of input channels of multi-scale feature map.
- **start_level** (*int, optional*) – The scale level where upsampling starts. Default: 2.
- **end_level** (*int, optional*) – The scale level where upsampling ends. Default: 5.
- **norm_cfg** (*dict, optional*) – Config dict for normalization layer. Default: None.
- **use_dcn** (*bool, optional*) – Whether to use dcn in IDAup module. Default: True.

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks

while the latter silently ignores them.

---

init_weights()
Initialize the weights.

class mmdet3d.models.necks.**FPN**(*in_channels*, *out_channels*, *num_outs*, *start_level=0*, *end_level=- 1*, *add_extra_convs=False*, *relu_before_extra_convs=False*, *no_norm_on_lateral=False*, *conv_cfg=None*, *norm_cfg=None*, *act_cfg=None*, *upsample_cfg={'mode': 'nearest'}*, *init_cfg={'distribution': 'uniform', 'layer': 'Conv2d', 'type': 'Xavier'}*)

Feature Pyramid Network.

This is an implementation of paper Feature Pyramid Networks for Object Detection.

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*) – Index of the start input backbone level used to build the feature pyramid. Default: 0.
- **end_level** (*int*) – Index of the end input backbone level (exclusive) to build the feature pyramid. Default: -1, which means the last level.
- **add_extra_convs** (*bool | str*) – If bool, it decides whether to add conv layers on top of the original feature maps. Default to False. If True, it is equivalent to *add_extra_convs='on_input'*. If str, it specifies the source feature map of the extra convs. Only the following options are allowed
  - 'on_input': Last feat map of neck inputs (i.e. backbone feature).
  - 'on_lateral': Last feature map after lateral convs.
  - 'on_output': The last output feature map after fpn convs.
- **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*) – Config dict for convolution layer. Default: None.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Default: None.
- **act_cfg** (*dict*) – Config dict for activation layer in ConvModule. Default: None.
- **upsample_cfg** (*dict*) – Config dict for interpolate layer. Default: dict(mode='nearest').
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

Example

```
>>> import torch
>>> in_channels = [2, 3, 5, 7]
>>> scales = [340, 170, 84, 43]
>>> inputs = [torch.rand(1, c, s, s)
...           for c, s in zip(in_channels, scales)]
>>> self = FPN(in_channels, 11, len(in_channels)).eval()
>>> outputs = self.forward(inputs)
>>> for i in range(len(outputs)):
...     print(f'outputs[{i}].shape = {outputs[i].shape}')
outputs[0].shape = torch.Size([1, 11, 340, 340])
outputs[1].shape = torch.Size([1, 11, 170, 170])
outputs[2].shape = torch.Size([1, 11, 84, 84])
outputs[3].shape = torch.Size([1, 11, 43, 43])
```

forward(*inputs*)
Forward function.

class mmdet3d.models.necks.**IndoorImVoxelNeck**(*in_channels*, *out_channels*, *n_blocks*)
Neck for ImVoxelNet outdoor scenario.

Parameters

- **in_channels** (*int*) – Number of channels in an input tensor.
- **out_channels** (*int*) – Number of channels in all output tensors.
- **n_blocks** (*list[int]*) – Number of blocks for each feature level.

forward(*x*)
Forward function.

Parameters **x** (*torch.Tensor*) – of shape (N, C_in, N_x, N_y, N_z).

Returns of shape (N, C_out, N_xi, N_yi, N_zi).

Return type list[torch.Tensor]

class mmdet3d.models.necks.**OutdoorImVoxelNeck**(*in_channels*, *out_channels*)
Neck for ImVoxelNet outdoor scenario.

Parameters

- **in_channels** (*int*) – Number of channels in an input tensor.
- **out_channels** (*int*) – Number of channels in all output tensors.

forward(*x*)
Forward function.

Parameters **x** (*torch.Tensor*) – of shape (N, C_in, N_x, N_y, N_z).

Returns of shape (N, C_out, N_y, N_x).

Return type list[torch.Tensor]

init_weights()
Initialize weights of neck.

class mmdet3d.models.necks.**PointNetFPNeck**(*fp_channels*, *init_cfg=None*)
PointNet FP Module used in PointRCNN.

Refer to the official code.

```
sa_n ----------------------------------------
                                              |
... ----------------------------------        |
                                    |         |
sa_1 -------------                  |         |
                 |                  |         |
sa_0 -> fp_0 -> fp_module ->fp_1 -> ... -> fp_module -> fp_n
```

sa_n including sa_xyz (torch.Tensor) and sa_features (torch.Tensor) fp_n including fp_xyz (torch.Tensor) and fp_features (torch.Tensor)

Parameters

- **fp_channels** (*tuple[tuple[int]]*) – Tuple of mlp channels in FP modules.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Default: None

forward(*feat_dict*)

Forward pass.

Parameters **feat_dict** (*dict*) – Feature dict from backbone.

Returns

Outputs of the Neck.

- fp_xyz (torch.Tensor): The coordinates of fp features.
- **fp_features (torch.Tensor): The features from the last** feature propagation layers.

Return type dict[str, torch.Tensor]

class mmdet3d.models.necks.**SECONDFPN**(*in_channels=[128, 128, 256]*, *out_channels=[256, 256, 256]*, *upsample_strides=[1, 2, 4]*, *norm_cfg={'eps': 0.001, 'momentum': 0.01, 'type': 'BN'}*, *upsample_cfg={'bias': False, 'type': 'deconv'}*, *conv_cfg={'bias': False, 'type': 'Conv2d'}*, *use_conv_for_no_stride=False*, *init_cfg=None*)

FPN used in SECOND/PointPillars/PartA2/MVXNet.

Parameters

- **in_channels** (*list[int]*) – Input channels of multi-scale feature maps.
- **out_channels** (*list[int]*) – Output channels of feature maps.
- **upsample_strides** (*list[int]*) – Strides used to upsample the feature maps.
- **norm_cfg** (*dict*) – Config dict of normalization layers.
- **upsample_cfg** (*dict*) – Config dict of upsample layers.
- **conv_cfg** (*dict*) – Config dict of conv layers.
- **use_conv_for_no_stride** (*bool*) – Whether to use conv when stride is 1.

forward(*x*)

Forward function.

Parameters **x** (*torch.Tensor*) – 4D Tensor in (N, C, H, W) shape.

Returns Multi-level feature maps.

Return type list[torch.Tensor]

47.4 dense_heads

class mmdet3d.models.dense_heads.**Anchor3DHead**(*num_classes*, *in_channels*, *train_cfg*, *test_cfg*, *feat_channels=256*, *use_direction_classifier=True*, *anchor_generator={'custom_values': [], 'range': [0, -39.68, -1.78, 69.12, 39.68, -1.78], 'reshape_out': False, 'rotations': [0, 1.57], 'sizes': [[3.9, 1.6, 1.56]], 'strides': [2], 'type': 'Anchor3DRangeGenerator'}*, *assigner_per_size=False*, *assign_per_class=False*, *diff_rad_by_sin=True*, *dir_offset=-1.5707963267948966*, *dir_limit_offset=0*, *bbox_coder={'type': 'DeltaXYZWLHRBBoxCoder'}*, *loss_cls={'loss_weight': 1.0, 'type': 'CrossEntropyLoss', 'use_sigmoid': True}*, *loss_bbox={'beta': 0.1111111111111111, 'loss_weight': 2.0, 'type': 'SmoothL1Loss'}*, *loss_dir={'loss_weight': 0.2, 'type': 'CrossEntropyLoss'}*, *init_cfg=None*)

Anchor head for SECOND/PointPillars/MVXNet/PartA2.

Parameters

- **num_classes** (*int*) – Number of classes.
- **in_channels** (*int*) – Number of channels in the input feature map.
- **train_cfg** (*dict*) – Train configs.
- **test_cfg** (*dict*) – Test configs.
- **feat_channels** (*int*) – Number of channels of the feature map.
- **use_direction_classifier** (*bool*) – Whether to add a direction classifier.
- **anchor_generator** (*dict*) – Config dict of anchor generator.
- **assigner_per_size** (*bool*) – Whether to do assignment for each separate anchor size.
- **assign_per_class** (*bool*) – Whether to do assignment for each class.
- **diff_rad_by_sin** (*bool*) – Whether to change the difference into sin difference for box regression loss.
- **dir_offset** (*float | int*) – The offset of BEV rotation angles. (TODO: may be moved into box coder)
- **dir_limit_offset** (*float | int*) – The limited range of BEV rotation angles. (TODO: may be moved into box coder)
- **bbox_coder** (*dict*) – Config dict of box coders.
- **loss_cls** (*dict*) – Config of classification loss.
- **loss_bbox** (*dict*) – Config of localization loss.
- **loss_dir** (*dict*) – Config of direction classifier loss.

static **add_sin_difference**(*boxes1*, *boxes2*)

Convert the rotation difference to difference in sine function.

Parameters

- **boxes1** (*torch.Tensor*) – Original Boxes in shape (NxC), where C>=7 and the 7th dimension is rotation dimension.

- **boxes2** (*torch.Tensor*) – Target boxes in shape (NxC), where C>=7 and the 7th dimension is rotation dimension.

Returns

boxes1 and boxes2 whose 7th dimensions are changed.

Return type tuple[torch.Tensor]

forward(*feats*)
Forward pass.

Parameters **feats** (*list[torch.Tensor]*) – Multi-level features, e.g., features produced by FPN.

Returns

Multi-level class score, bbox and direction predictions.

Return type tuple[list[torch.Tensor]]

forward_single(*x*)
Forward function on a single-scale feature map.

Parameters **x** (*torch.Tensor*) – Input features.

Returns

Contain score of each class, bbox regression and direction classification predictions.

Return type tuple[torch.Tensor]

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

Parameters

- **featmap_sizes** (*list[tuple]*) – Multi-level feature map sizes.
- **input_metas** (*list[dict]*) – contain pcd and img's meta info.
- **device** (*str*) – device of current module.

Returns

Anchors of each image, valid flags of each image.

Return type list[list[torch.Tensor]]

get_bboxes(*cls_scores*, *bbox_preds*, *dir_cls_preds*, *input_metas*, *cfg=None*, *rescale=False*)
Get bboxes of anchor head.

Parameters

- **cls_scores** (*list[torch.Tensor]*) – Multi-level class scores.
- **bbox_preds** (*list[torch.Tensor]*) – Multi-level bbox predictions.
- **dir_cls_preds** (*list[torch.Tensor]*) – Multi-level direction class predictions.
- **input_metas** (*list[dict]*) – Contain pcd and img's meta info.
- **cfg** (ConfigDict) – Training or testing config.
- **rescale** (*list[torch.Tensor]*) – Whether th rescale bbox.

Returns Prediction resultes of batches.

Return type list[tuple]

get_bboxes_single(*cls_scores*, *bbox_preds*, *dir_cls_preds*, *mlvl_anchors*, *input_meta*, *cfg=None*, *rescale=False*)

Get bboxes of single branch.

Parameters

- **cls_scores** (*torch.Tensor*) – Class score in single batch.
- **bbox_preds** (*torch.Tensor*) – Bbox prediction in single batch.
- **dir_cls_preds** (*torch.Tensor*) – Predictions of direction class in single batch.
- **mlvl_anchors** (*List[torch.Tensor]*) – Multi-level anchors in single batch.
- **input_meta** (*list[dict]*) – Contain pcd and img's meta info.
- **cfg** (ConfigDict) – Training or testing config.
- **rescale** (*list[torch.Tensor]*) – whether th rescale bbox.

Returns

Contain predictions of single batch.

- bboxes (BaseInstance3DBoxes): Predicted 3d bboxes.
- scores (torch.Tensor): Class score of each bbox.
- labels (torch.Tensor): Label of each bbox.

Return type tuple

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

Calculate losses.

Parameters

- **cls_scores** (*list[torch.Tensor]*) – Multi-level class scores.
- **bbox_preds** (*list[torch.Tensor]*) – Multi-level bbox predictions.
- **dir_cls_preds** (*list[torch.Tensor]*) – Multi-level direction class predictions.
- **gt_bboxes** (list[BaseInstance3DBoxes]) – Gt bboxes of each sample.
- **gt_labels** (*list[torch.Tensor]*) – Gt labels of each sample.
- **input_metas** (*list[dict]*) – Contain pcd and img's meta info.
- **gt_bboxes_ignore** (*list[torch.Tensor]*) – Specify which bounding boxes to ignore.

Returns

Classification, bbox, and direction losses of each level.

- loss_cls (list[torch.Tensor]): Classification losses.
- loss_bbox (list[torch.Tensor]): Box regression losses.
- **loss_dir (list[torch.Tensor]): Direction classification** losses.

Return type dict[str, list[torch.Tensor]]

loss_single(*cls_score*, *bbox_pred*, *dir_cls_preds*, *labels*, *label_weights*, *bbox_targets*, *bbox_weights*, *dir_targets*, *dir_weights*, *num_total_samples*)

Calculate loss of Single-level results.

Parameters

- **cls_score** (*torch.Tensor*) – Class score in single-level.
- **bbox_pred** (*torch.Tensor*) – Bbox prediction in single-level.
- **dir_cls_preds** (*torch.Tensor*) – Predictions of direction class in single-level.
- **labels** (*torch.Tensor*) – Labels of class.
- **label_weights** (*torch.Tensor*) – Weights of class loss.
- **bbox_targets** (*torch.Tensor*) – Targets of bbox predictions.
- **bbox_weights** (*torch.Tensor*) – Weights of bbox loss.
- **dir_targets** (*torch.Tensor*) – Targets of direction predictions.
- **dir_weights** (*torch.Tensor*) – Weights of direction loss.
- **num_total_samples** (*int*) – The number of valid samples.

Returns

Losses of class, bbox and direction, respectively.

Return type tuple[torch.Tensor]

class mmdet3d.models.dense_heads.**AnchorFreeMono3DHead**(*num_classes*, *in_channels*, *feat_channels=256*, *stacked_convs=4*, *strides=(4, 8, 16, 32, 64)*, *dcn_on_last_conv=False*, *conv_bias='auto'*, *background_label=None*, *use_direction_classifier=True*, *diff_rad_by_sin=True*, *dir_offset=0*, *dir_limit_offset=0*, *loss_cls={'alpha': 0.25, 'gamma': 2.0, 'loss_weight': 1.0, 'type': 'FocalLoss', 'use_sigmoid': True}*, *loss_bbox={'beta': 0.1111111111111111, 'loss_weight': 1.0, 'type': 'SmoothL1Loss'}*, *loss_dir={'loss_weight': 1.0, 'type': 'CrossEntropyLoss', 'use_sigmoid': False}*, *loss_attr={'loss_weight': 1.0, 'type': 'CrossEntropyLoss', 'use_sigmoid': False}*, *bbox_code_size=9*, *pred_attrs=False*, *num_attrs=9*, *pred_velo=False*, *pred_bbox2d=False*, *group_reg_dims=(2, 1, 3, 1, 2)*, *cls_branch=(128, 64)*, *reg_branch=((128, 64), (128, 64), (64), (64), ())*, *dir_branch=(64)*, *attr_branch=(64)*, *conv_cfg=None*, *norm_cfg=None*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*)

Anchor-free head for monocular 3D object detection.

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, optional*) – Number of hidden channels. Used in child classes. Defaults to 256.
- **stacked_convs** (*int, optional*) – Number of stacking convs of the head.

- **strides** (*tuple, optional*) – Downsample factor of each feature map.
- **dcn_on_last_conv** (*bool, optional*) – If true, use dcn in the last layer of towers. Default: False.
- **conv_bias** (*bool | str, optional*) – If specified as *auto*, it will be decided by the norm_cfg. Bias of conv will be set as True if *norm_cfg* is None, otherwise False. Default: 'auto'.
- **background_label** (*int, optional*) – 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.
- **use_direction_classifier** (*bool, optional*) – Whether to add a direction classifier.
- **diff_rad_by_sin** (*bool, optional*) – Whether to change the difference into sin difference for box regression loss. Defaults to True.
- **dir_offset** (*float, optional*) – Parameter used in direction classification. Defaults to 0.
- **dir_limit_offset** (*float, optional*) – Parameter used in direction classification. Defaults to 0.
- **loss_cls** (*dict, optional*) – Config of classification loss.
- **loss_bbox** (*dict, optional*) – Config of localization loss.
- **loss_dir** (*dict, optional*) – Config of direction classifier loss.
- **loss_attr** (*dict, optional*) – Config of attribute classifier loss, which is only active when *pred_attrs=True*.
- **bbox_code_size** (*int, optional*) – Dimensions of predicted bounding boxes.
- **pred_attrs** (*bool, optional*) – Whether to predict attributes. Defaults to False.
- **num_attrs** (*int, optional*) – The number of attributes to be predicted. Default: 9.
- **pred_velo** (*bool, optional*) – Whether to predict velocity. Defaults to False.
- **pred_bbox2d** (*bool, optional*) – Whether to predict 2D boxes. Defaults to False.
- **group_reg_dims** (*tuple[int], optional*) – The dimension of each regression target group. Default: (2, 1, 3, 1, 2).
- **cls_branch** (*tuple[int], optional*) – Channels for classification branch. Default: (128, 64).
- **reg_branch** (*tuple[tuple], optional*) – Channels for regression branch. Default: (

  (128, 64), # offset (128, 64), # depth (64,), # size (64,), # rot () # velo

  ),
- **dir_branch** (*tuple[int], optional*) – Channels for direction classification branch. Default: (64,).
- **attr_branch** (*tuple[int], optional*) – Channels for classification branch. Default: (64,).
- **conv_cfg** (*dict, optional*) – Config dict for convolution layer. Default: None.
- **norm_cfg** (*dict, optional*) – Config dict for normalization layer. Default: None.
- **train_cfg** (*dict, optional*) – Training config of anchor head.

- **test_cfg** (*dict, optional*) – Testing config of anchor head.

forward(*feats*)

Forward features from the upstream network.

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

Returns

Usually contain classification scores, bbox predictions, and direction class predictions. cls_scores (list[Tensor]): Box scores for each scale level,

each is a 4D-tensor, the channel number is num_points * num_classes.

bbox_preds (list[Tensor]): Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_points * bbox_code_size.

dir_cls_preds (list[Tensor]): Box scores for direction class predictions on each scale level, each is a 4D-tensor, the channel number is num_points * 2. (bin = 2)

attr_preds (list[Tensor]): Attribute scores for each scale level, each is a 4D-tensor, the channel number is num_points * num_attrs.

Return type tuple

forward_single(*x*)

Forward features of a single scale level.

Parameters **x** (*Tensor*) – FPN feature maps of the specified stride.

Returns

Scores for each class, bbox predictions, direction class, and attributes, features after classification and regression conv layers, some models needs these features like FCOS.

Return type tuple

abstract get_bboxes(*cls_scores*, *bbox_preds*, *dir_cls_preds*, *attr_preds*, *img_metas*, *cfg=None*, *rescale=None*)

Transform network output for a batch into bbox predictions.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_points * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, num_points * bbox_code_size, H, W)
- **dir_cls_preds** (*list[Tensor]*) – Box scores for direction class predictions on each scale level, each is a 4D-tensor, the channel number is num_points * 2. (bin = 2)
- **attr_preds** (*list[Tensor]*) – Attribute scores for each scale level Has shape (N, num_points * num_attrs, H, W)
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **cfg** (*mmcv.Config*) – Test / postprocessing configuration, if None, test_cfg would be used
- **rescale** (*bool*) – If True, return boxes in original image space

get_points(*featmap_sizes*, *dtype*, *device*, *flatten=False*)
Get points according to feature map sizes.

Parameters

- **featmap_sizes** (*list[tuple]*) – Multi-level feature map sizes.
- **dtype** (*torch.dtype*) – Type of points.
- **device** (*torch.device*) – Device of points.

Returns points of each image.

Return type tuple

abstract get_targets(*points*, *gt_bboxes_list*, *gt_labels_list*, *gt_bboxes_3d_list*, *gt_labels_3d_list*, *centers2d_list*, *depths_list*, *attr_labels_list*)
Compute regression, classification and centerss targets for points in multiple images.

Parameters

- **points** (*list[Tensor]*) – Points of each fpn level, each has shape (num_points, 2).
- **gt_bboxes_list** (*list[Tensor]*) – Ground truth bboxes of each image, each has shape (num_gt, 4).
- **gt_labels_list** (*list[Tensor]*) – Ground truth labels of each box, each has shape (num_gt,).
- **gt_bboxes_3d_list** (*list[Tensor]*) – 3D Ground truth bboxes of each image, each has shape (num_gt, bbox_code_size).
- **gt_labels_3d_list** (*list[Tensor]*) – 3D Ground truth labels of each box, each has shape (num_gt,).
- **centers2d_list** (*list[Tensor]*) – Projected 3D centers onto 2D image, each has shape (num_gt, 2).
- **depths_list** (*list[Tensor]*) – Depth of projected 3D centers onto 2D image, each has shape (num_gt, 1).
- **attr_labels_list** (*list[Tensor]*) – Attribute labels of each box, each has shape (num_gt,).

init_weights()
Initialize weights of the head.

We currently still use the customized defined init_weights because the default init of DCN triggered by the init_cfg will init conv_offset.weight, which mistakenly affects the training stability.

abstract loss(*cls_scores*, *bbox_preds*, *dir_cls_preds*, *attr_preds*, *gt_bboxes*, *gt_labels*, *gt_bboxes_3d*, *gt_labels_3d*, *centers2d*, *depths*, *attr_labels*, *img_metas*, *gt_bboxes_ignore=None*)
Compute loss of the head.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level, each is a 4D-tensor, the channel number is num_points * num_classes.
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_points * bbox_code_size.
- **dir_cls_preds** (*list[Tensor]*) – Box scores for direction class predictions on each scale level, each is a 4D-tensor, the channel number is num_points * 2. (bin = 2)

- **attr_preds** (*list[Tensor]*) – Box scores for each scale level, each is a 4D-tensor, the channel number is num_points * num_attrs.
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box
- **gt_bboxes_3d** (*list[Tensor]*) – 3D Ground truth bboxes for each image with shape (num_gts, bbox_code_size).
- **gt_labels_3d** (*list[Tensor]*) – 3D class indices of each box.
- **centers2d** (*list[Tensor]*) – Projected 3D centers onto 2D images.
- **depths** (*list[Tensor]*) – Depth of projected centers on 2D images.
- **attr_labels** (*list[Tensor], optional*) – Attribute indices corresponding to each box
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **gt_bboxes_ignore** (*list[Tensor]*) – specify which bounding boxes can be ignored when computing the loss.

class mmdet3d.models.dense_heads.**BaseConvBboxHead**(*in_channels=0*, *shared_conv_channels=()*, *cls_conv_channels=()*, *num_cls_out_channels=0*, *reg_conv_channels=()*, *num_reg_out_channels=0*, *conv_cfg={'type': 'Conv1d'}*, *norm_cfg={'type': 'BN1d'}*, *act_cfg={'type': 'ReLU'}*, *bias='auto'*, *init_cfg=None*, **args*, ***kwargs*)

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

```
             /-> cls convs -> cls_score
shared convs
             \-> reg convs -> bbox_pred
```

forward(*feats*)

Forward.

Parameters **feats** (*Tensor*) – Input features

Returns Class scores predictions Tensor: Regression predictions

Return type Tensor

class mmdet3d.models.dense_heads.**BaseMono3DDenseHead**(*init_cfg=None*)

Base class for Monocular 3D DenseHeads.

forward_train(*x*, *img_metas*, *gt_bboxes*, *gt_labels=None*, *gt_bboxes_3d=None*, *gt_labels_3d=None*, *centers2d=None*, *depths=None*, *attr_labels=None*, *gt_bboxes_ignore=None*, *proposal_cfg=None*, ***kwargs*)

Parameters

- **x** (*list[Tensor]*) – Features from FPN.
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes of the image, shape (num_gts, 4).

- **gt_labels** (*list[Tensor]*) – Ground truth labels of each box, shape (num_gts,).
- **gt_bboxes_3d** (*list[Tensor]*) – 3D ground truth bboxes of the image, shape (num_gts, self.bbox_code_size).
- **gt_labels_3d** (*list[Tensor]*) – 3D ground truth labels of each box, shape (num_gts,).
- **centers2d** (*list[Tensor]*) – Projected 3D center of each box, shape (num_gts, 2).
- **depths** (*list[Tensor]*) – Depth of projected 3D center of each box, shape (num_gts,).
- **attr_labels** (*list[Tensor]*) – Attribute labels of each box, shape (num_gts,).
- **gt_bboxes_ignore** (*list[Tensor]*) – Ground truth bboxes to be ignored, shape (num_ignored_gts, 4).
- **proposal_cfg** (*mmcv.Config*) – Test / postprocessing configuration, if None, test_cfg would be used

Returns losses: (dict[str, Tensor]): A dictionary of loss components. proposal_list (list[Tensor]): Proposals of each image.

Return type tuple

abstract get_bboxes(***kwargs*)

Transform network output for a batch into bbox predictions.

abstract loss(***kwargs*)

Compute losses of the head.

class mmdet3d.models.dense_heads.**CenterHead**(*in_channels=[128]*, *tasks=None*, *train_cfg=None*, *test_cfg=None*, *bbox_coder=None*, *common_heads={}*, *loss_cls={'reduction': 'mean', 'type': 'GaussianFocalLoss'}*, *loss_bbox={'loss_weight': 0.25, 'reduction': 'none', 'type': 'L1Loss'}*, *separate_head={'final_kernel': 3, 'init_bias': - 2.19, 'type': 'SeparateHead'}*, *share_conv_channel=64*, *num_heatmap_convs=2*, *conv_cfg={'type': 'Conv2d'}*, *norm_cfg={'type': 'BN2d'}*, *bias='auto'*, *norm_bbox=True*, *init_cfg=None*)

CenterHead for CenterPoint.

Parameters

- **in_channels** (*list[int] | int, optional*) – Channels of the input feature map. Default: [128].
- **tasks** (*list[dict], optional*) – Task information including class number and class names. Default: None.
- **train_cfg** (*dict, optional*) – Train-time configs. Default: None.
- **test_cfg** (*dict, optional*) – Test-time configs. Default: None.
- **bbox_coder** (*dict, optional*) – Bbox coder configs. Default: None.
- **common_heads** (*dict, optional*) – Conv information for common heads. Default: dict().
- **loss_cls** (*dict, optional*) – Config of classification loss function. Default: dict(type='GaussianFocalLoss', reduction='mean').

- **loss_bbox** (*dict, optional*) – Config of regression loss function. Default: dict(type='L1Loss', reduction='none').
- **separate_head** (*dict, optional*) – Config of separate head. Default: dict(type='SeparateHead', init_bias=-2.19, final_kernel=3)
- **share_conv_channel** (*int, optional*) – Output channels for share_conv layer. Default: 64.
- **num_heatmap_convs** (*int, optional*) – Number of conv layers for heatmap conv layer. Default: 2.
- **conv_cfg** (*dict, optional*) – Config of conv layer. Default: dict(type='Conv2d')
- **norm_cfg** (*dict, optional*) – Config of norm layer. Default: dict(type='BN2d').
- **bias** (*str, optional*) – Type of bias. Default: 'auto'.

forward(*feats*)

Forward pass.

Parameters **feats** (*list[torch.Tensor]*) – Multi-level features, e.g., features produced by FPN.

Returns Output results for tasks.

Return type tuple(list[dict])

forward_single(*x*)

Forward function for CenterPoint.

Parameters **x** (*torch.Tensor*) – Input feature map with the shape of [B, 512, 128, 128].

Returns Output results for tasks.

Return type list[dict]

get_bboxes(*preds_dicts*, *img_metas*, *img=None*, *rescale=False*)

Generate bboxes from bbox head predictions.

Parameters

- **preds_dicts** (*tuple[list[dict]]*) – Prediction results.
- **img_metas** (*list[dict]*) – Point cloud and image's meta info.

Returns Decoded bbox, scores and labels after nms.

Return type list[dict]

get_targets(*gt_bboxes_3d*, *gt_labels_3d*)

Generate targets.

How each output is transformed:

Each nested list is transposed so that all same-index elements in each sub-list (1, ..., N) become the new sub-lists.

[[a0, a1, a2, ...], [b0, b1, b2, ...], ...] ==> [[a0, b0, ...], [a1, b1, ...], [a2, b2, ...]]

The new transposed nested list is converted into a list of N tensors generated by concatenating tensors in the new sub-lists.

[tensor0, tensor1, tensor2, ...]

Parameters

- **gt_bboxes_3d** (list[LiDARInstance3DBoxes]) – Ground truth gt boxes.
- **gt_labels_3d** (*list[torch.Tensor]*) – Labels of boxes.

Returns

tuple[list[torch.Tensor]]: Tuple of target including the following results in order.

- list[torch.Tensor]: Heatmap scores.
- list[torch.Tensor]: Ground truth boxes.
- **list[torch.Tensor]: Indexes indicating the** position of the valid boxes.
- **list[torch.Tensor]: Masks indicating which** boxes are valid.

Return type Returns

get_targets_single(*gt_bboxes_3d*, *gt_labels_3d*)

Generate training targets for a single sample.

Parameters

- **gt_bboxes_3d** (LiDARInstance3DBoxes) – Ground truth gt boxes.
- **gt_labels_3d** (*torch.Tensor*) – Labels of boxes.

Returns

Tuple of target including the following results in order.

- list[torch.Tensor]: Heatmap scores.
- list[torch.Tensor]: Ground truth boxes.
- **list[torch.Tensor]: Indexes indicating the position** of the valid boxes.
- **list[torch.Tensor]: Masks indicating which boxes** are valid.

Return type tuple[list[torch.Tensor]]

get_task_detections(*num_class_with_bg*, *batch_cls_preds*, *batch_reg_preds*, *batch_cls_labels*, *img_metas*)

Rotate nms for each task.

Parameters

- **num_class_with_bg** (*int*) – Number of classes for the current task.
- **batch_cls_preds** (*list[torch.Tensor]*) – Prediction score with the shape of [N].
- **batch_reg_preds** (*list[torch.Tensor]*) – Prediction bbox with the shape of [N, 9].
- **batch_cls_labels** (*list[torch.Tensor]*) – Prediction label with the shape of [N].
- **img_metas** (*list[dict]*) – Meta information of each sample.

Returns

torch.Tensor]]: contains the following keys:

-bboxes (torch.Tensor): Prediction bboxes after nms with the shape of [N, 9].

-scores (torch.Tensor): Prediction scores after nms with the shape of [N].

-labels (torch.Tensor): Prediction labels after nms with the shape of [N].

Return type list[dict[str

loss(*gt_bboxes_3d*, *gt_labels_3d*, *preds_dicts*, ***kwargs*)
Loss function for CenterHead.

Parameters

- **gt_bboxes_3d** (list[LiDARInstance3DBoxes]) – Ground truth gt boxes.
- **gt_labels_3d** (*list[torch.Tensor]*) – Labels of boxes.
- **preds_dicts** (*dict*) – Output of forward function.

Returns torch.Tensor]: Loss of heatmap and bbox of each task.

Return type dict[str

class mmdet3d.models.dense_heads.**FCAF3DHead**(*n_classes*, *in_channels*, *out_channels*, *n_reg_outs*, *voxel_size*, *pts_prune_threshold*, *pts_assign_threshold*, *pts_center_threshold*, *center_loss={'type': 'CrossEntropyLoss', 'use_sigmoid': True}*, *bbox_loss={'type': 'AxisAlignedIoULoss'}*, *cls_loss={'type': 'FocalLoss'}*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*)
Bbox head of FCAF3D. Actually here we store both the sparse 3D FPN and a head. The neck and the head can not be simply separated as pruning score on the i-th level of FPN requires classification scores from i+1-th level of the head.

Parameters

- **n_classes** (*int*) – Number of classes.
- **in_channels** (*tuple[int]*) – Number of channels in input tensors.
- **out_channels** (*int*) – Number of channels in the neck output tensors.
- **n_reg_outs** (*int*) – Number of regression layer channels.
- **voxel_size** (*float*) – Voxel size in meters.
- **pts_prune_threshold** (*int*) – Pruning threshold on each feature level.
- **pts_assign_threshold** (*int*) – Box to location assigner parameter. Assigner selects the maximum feature level with more locations inside the box than pts_assign_threshold.
- **pts_center_threshold** (*int*) – Box to location assigner parameter. After feature level for the box is determined, assigner selects pts_center_threshold locations closest to the box center.
- **center_loss** (*dict, optional*) – Config of centerness loss.
- **bbox_loss** (*dict, optional*) – Config of bbox loss.
- **cls_loss** (*dict, optional*) – Config of classification loss.
- **train_cfg** (*dict, optional*) – Config for train stage. Defaults to None.
- **test_cfg** (*dict, optional*) – Config for test stage. Defaults to None.
- **init_cfg** (*dict, optional*) – Config for weight initialization. Defaults to None.

forward(*x*)
Forward pass.

Parameters **x** (*list[Tensor]*) – Features from the backbone.

Returns Predictions of the head.

Return type list[list[Tensor]]

forward_test(*x*, *input_metas*)
Forward pass of the test stage.

Parameters

- **x** (*list[SparseTensor]*) – Features from the backbone.
- **input_metas** (*list[dict]*) – Contains scene meta info for each sample.

Returns bboxes, scores and labels for each sample.

Return type list[list[Tensor]]

forward_train(*x*, *gt_bboxes*, *gt_labels*, *input_metas*)
Forward pass of the train stage.

Parameters

- **x** (*list[SparseTensor]*) – Features from the backbone.
- **gt_bboxes** (list[BaseInstance3DBoxes]) – Ground truth bboxes of each sample.
- **gt_labels** (*list[torch.Tensor]*) – Labels of each sample.
- **input_metas** (*list[dict]*) – Contains scene meta info for each sample.

Returns Centerness, bbox and classification loss values.

Return type dict

init_weights()
Initialize weights.

class mmdet3d.models.dense_heads.**FCOSMono3DHead**(*regress_ranges=((- 1, 48), (48, 96), (96, 192), (192, 384), (384, 100000000.0))*, *center_sampling=True*, *center_sample_radius=1.5*, *norm_on_bbox=True*, *centerness_on_reg=True*, *centerness_alpha=2.5*, *loss_cls={'alpha': 0.25, 'gamma': 2.0, 'loss_weight': 1.0, 'type': 'FocalLoss', 'use_sigmoid': True}*, *loss_bbox={'beta': 0.1111111111111111, 'loss_weight': 1.0, 'type': 'SmoothL1Loss'}*, *loss_dir={'loss_weight': 1.0, 'type': 'CrossEntropyLoss', 'use_sigmoid': False}*, *loss_attr={'loss_weight': 1.0, 'type': 'CrossEntropyLoss', 'use_sigmoid': False}*, *loss_centerness={'loss_weight': 1.0, 'type': 'CrossEntropyLoss', 'use_sigmoid': True}*, *bbox_coder={'code_size': 9, 'type': 'FCOS3DBBoxCoder'}*, *norm_cfg={'num_groups': 32, 'requires_grad': True, 'type': 'GN'}*, *centerness_branch=(64)*, *init_cfg=None*, ***kwargs*)

Anchor-free head used in FCOS3D.

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **in_channels** (*int*) – Number of channels in the input feature map.

- **regress_ranges** (*tuple[tuple[int, int]], optional*) – Regress range of multiple level points.
- **center_sampling** (*bool, optional*) – If true, use center sampling. Default: True.
- **center_sample_radius** (*float, optional*) – Radius of center sampling. Default: 1.5.
- **norm_on_bbox** (*bool, optional*) – If true, normalize the regression targets with FPN strides. Default: True.
- **centerness_on_reg** (*bool, optional*) – If true, position centerness on the regress branch. Please refer to https://github.com/tianzhi0549/FCOS/issues/89#issuecomment-516877042. Default: True.
- **centerness_alpha** (*int, optional*) – Parameter used to adjust the intensity attenuation from the center to the periphery. Default: 2.5.
- **loss_cls** (*dict, optional*) – Config of classification loss.
- **loss_bbox** (*dict, optional*) – Config of localization loss.
- **loss_dir** (*dict, optional*) – Config of direction classification loss.
- **loss_attr** (*dict, optional*) – Config of attribute classification loss.
- **loss_centerness** (*dict, optional*) – Config of centerness loss.
- **norm_cfg** (*dict, optional*) – dictionary to construct and config norm layer. Default: norm_cfg=dict(type='GN', num_groups=32, requires_grad=True).
- **centerness_branch** (*tuple[int], optional*) – Channels for centerness branch. Default: (64,).

static add_sin_difference(*boxes1*, *boxes2*)

Convert the rotation difference to difference in sine function.

Parameters

- **boxes1** (*torch.Tensor*) – Original Boxes in shape (NxC), where C>=7 and the 7th dimension is rotation dimension.
- **boxes2** (*torch.Tensor*) – Target boxes in shape (NxC), where C>=7 and the 7th dimension is rotation dimension.

Returns

boxes1 and boxes2 whose 7th dimensions are changed.

Return type tuple[torch.Tensor]

forward(*feats*)

Forward features from the upstream network.

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

Returns

cls_scores (list[Tensor]): Box scores for each scale level, each is a 4D-tensor, the channel number is num_points * num_classes.

bbox_preds (list[Tensor]): Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_points * bbox_code_size.

dir_cls_preds (list[Tensor]): Box scores for direction class predictions on each scale level, each is a 4D-tensor, the channel number is num_points * 2. (bin = 2).

attr_preds (list[Tensor]): Attribute scores for each scale level, each is a 4D-tensor, the channel number is num_points * num_attrs.

centernesses (list[Tensor]): Centerness for each scale level, each is a 4D-tensor, the channel number is num_points * 1.

Return type tuple

`forward_single`(*x*, *scale*, *stride*)

Forward features of a single scale level.

Parameters

- **x** (*Tensor*) – FPN feature maps of the specified stride.
- **(** (*scale*) – obj: *mmcv.cnn.Scale*): Learnable scale module to resize the bbox prediction.
- **stride** (*int*) – The corresponding stride for feature maps, only used to normalize the bbox prediction when self.norm_on_bbox is True.

Returns

scores for each class, bbox and direction class predictions, centerness predictions of input feature maps.

Return type tuple

`get_bboxes`(*cls_scores*, *bbox_preds*, *dir_cls_preds*, *attr_preds*, *centernesses*, *img_metas*, *cfg=None*, *rescale=None*)

Transform network output for a batch into bbox predictions.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_points * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, num_points * 4, H, W)
- **dir_cls_preds** (*list[Tensor]*) – Box scores for direction class predictions on each scale level, each is a 4D-tensor, the channel number is num_points * 2. (bin = 2)
- **attr_preds** (*list[Tensor]*) – Attribute scores for each scale level Has shape (N, num_points * num_attrs, H, W)
- **centernesses** (*list[Tensor]*) – Centerness for each scale level with shape (N, num_points * 1, H, W)
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **cfg** (*mmcv.Config*) – Test / postprocessing configuration, if None, test_cfg would be used
- **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, 5) tensor, where the first 4 columns are bounding box positions (tl_x, tl_y, br_x, br_y) and the 5-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]]

static get_direction_target(*reg_targets*, *dir_offset=0*, *dir_limit_offset=0.0*, *num_bins=2*, *one_hot=True*)

Encode direction to 0 ~ num_bins-1.

Parameters

- **reg_targets** (*torch.Tensor*) – Bbox regression targets.
- **dir_offset** (*int, optional*) – Direction offset. Default to 0.
- **dir_limit_offset** (*float, optional*) – Offset to set the direction range. Default to 0.0.
- **num_bins** (*int, optional*) – Number of bins to divide 2*PI. Default to 2.
- **one_hot** (*bool, optional*) – Whether to encode as one hot. Default to True.

Returns Encoded direction targets.

Return type torch.Tensor

get_targets(*points*, *gt_bboxes_list*, *gt_labels_list*, *gt_bboxes_3d_list*, *gt_labels_3d_list*, *centers2d_list*, *depths_list*, *attr_labels_list*)

Compute regression, classification and centerss targets for points in multiple images.

Parameters

- **points** (*list[Tensor]*) – Points of each fpn level, each has shape (num_points, 2).
- **gt_bboxes_list** (*list[Tensor]*) – Ground truth bboxes of each image, each has shape (num_gt, 4).
- **gt_labels_list** (*list[Tensor]*) – Ground truth labels of each box, each has shape (num_gt,).
- **gt_bboxes_3d_list** (*list[Tensor]*) – 3D Ground truth bboxes of each image, each has shape (num_gt, bbox_code_size).
- **gt_labels_3d_list** (*list[Tensor]*) – 3D Ground truth labels of each box, each has shape (num_gt,).
- **centers2d_list** (*list[Tensor]*) – Projected 3D centers onto 2D image, each has shape (num_gt, 2).
- **depths_list** (*list[Tensor]*) – Depth of projected 3D centers onto 2D image, each has shape (num_gt, 1).
- **attr_labels_list** (*list[Tensor]*) – Attribute labels of each box, each has shape (num_gt,).

Returns

concat_lvl_labels (list[Tensor]): Labels of each level. concat_lvl_bbox_targets (list[Tensor]): BBox targets of each

level.

Return type tuple

init_weights()
Initialize weights of the head.

We currently still use the customized init_weights because the default init of DCN triggered by the init_cfg will init conv_offset.weight, which mistakenly affects the training stability.

loss(*cls_scores*, *bbox_preds*, *dir_cls_preds*, *attr_preds*, *centernesses*, *gt_bboxes*, *gt_labels*, *gt_bboxes_3d*, *gt_labels_3d*, *centers2d*, *depths*, *attr_labels*, *img_metas*, *gt_bboxes_ignore=None*)
Compute loss of the head.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level, each is a 4D-tensor, the channel number is num_points * num_classes.
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_points * bbox_code_size.
- **dir_cls_preds** (*list[Tensor]*) – Box scores for direction class predictions on each scale level, each is a 4D-tensor, the channel number is num_points * 2. (bin = 2)
- **attr_preds** (*list[Tensor]*) – Attribute scores for each scale level, each is a 4D-tensor, the channel number is num_points * num_attrs.
- **centernesses** (*list[Tensor]*) – Centerness for each scale level, each is a 4D-tensor, the channel number is num_points * 1.
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box
- **gt_bboxes_3d** (*list[Tensor]*) – 3D boxes ground truth with shape of (num_gts, code_size).
- **gt_labels_3d** (*list[Tensor]*) – same as gt_labels
- **centers2d** (*list[Tensor]*) – 2D centers on the image with shape of (num_gts, 2).
- **depths** (*list[Tensor]*) – Depth ground truth with shape of (num_gts,).
- **attr_labels** (*list[Tensor]*) – Attributes indices of each box.
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **gt_bboxes_ignore** (*list[Tensor]*) – specify which bounding boxes can be ignored when computing the loss.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

static pts2Dto3D(*points*, *view*)

Parameters

- **points** (*torch.Tensor*) – points in 2D images, [N, 3], 3 corresponds with x, y in the image and depth.
- **view** (*np.ndarray*) – camera intrinsic, [3, 3]

Returns

points in 3D space. [N, 3], 3 corresponds with x, y, z in 3D space.

Return type torch.Tensor

class mmdet3d.models.dense_heads.**FreeAnchor3DHead**(*pre_anchor_topk=50*, *bbox_thr=0.6*, *gamma=2.0*, *alpha=0.5*, *init_cfg=None*, ***kwargs*)

FreeAnchor head for 3D detection.

Note: This implementation is directly modified from the mmdet implementation. We find it also works on 3D detection with minor modification, i.e., different hyper-parameters and a additional direction classifier.

Parameters

- **pre_anchor_topk** (*int*) – Number of boxes that be token in each bag.
- **bbox_thr** (*float*) – The threshold of the saturated linear function. It is usually the same with the IoU threshold used in NMS.
- **gamma** (*float*) – Gamma parameter in focal loss.
- **alpha** (*float*) – Alpha parameter in focal loss.
- **kwargs** (*dict*) – Other arguments are the same as those in *Anchor3DHead*.

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

Calculate loss of FreeAnchor head.

Parameters

- **cls_scores** (*list[torch.Tensor]*) – Classification scores of different samples.
- **bbox_preds** (*list[torch.Tensor]*) – Box predictions of different samples
- **dir_cls_preds** (*list[torch.Tensor]*) – Direction predictions of different samples
- **gt_bboxes** (list[BaseInstance3DBoxes]) – Ground truth boxes.
- **gt_labels** (*list[torch.Tensor]*) – Ground truth labels.
- **input_metas** (*list[dict]*) – List of input meta information.
- **gt_bboxes_ignore** (list[BaseInstance3DBoxes], optional) – Ground truth boxes that should be ignored. Defaults to None.

Returns

Loss items.

- positive_bag_loss (torch.Tensor): Loss of positive samples.
- negative_bag_loss (torch.Tensor): Loss of negative samples.

Return type dict[str, torch.Tensor]

negative_bag_loss(*cls_prob*, *box_prob*)

Generate negative bag loss.

Parameters

- **cls_prob** (*torch.Tensor*) – Classification probability of negative samples.
- **box_prob** (*torch.Tensor*) – Bounding box probability of negative samples.

Returns Loss of negative samples.

Return type torch.Tensor

positive_bag_loss(*matched_cls_prob*, *matched_box_prob*)
Generate positive bag loss.

Parameters

- **matched_cls_prob** (*torch.Tensor*) – Classification probability of matched positive samples.
- **matched_box_prob** (*torch.Tensor*) – Bounding box probability of matched positive samples.

Returns Loss of positive samples.

Return type torch.Tensor

class mmdet3d.models.dense_heads.**GroupFree3DHead**(*num_classes*, *in_channels*, *bbox_coder*, *num_decoder_layers*, *transformerlayers*, *decoder_self_posembeds={'input_channel': 6, 'num_pos_feats': 288, 'type': 'ConvBNPositionalEncoding'}*, *decoder_cross_posembeds={'input_channel': 3, 'num_pos_feats': 288, 'type': 'ConvBNPositionalEncoding'}*, *train_cfg=None*, *test_cfg=None*, *num_proposal=128*, *pred_layer_cfg=None*, *size_cls_agnostic=True*, *gt_per_seed=3*, *sampling_objectness_loss=None*, *objectness_loss=None*, *center_loss=None*, *dir_class_loss=None*, *dir_res_loss=None*, *size_class_loss=None*, *size_res_loss=None*, *size_reg_loss=None*, *semantic_loss=None*, *init_cfg=None*)

Bbox head of Group-Free 3D.

Parameters

- **num_classes** (*int*) – The number of class.
- **in_channels** (*int*) – The dims of input features from backbone.
- **bbox_coder** (BaseBBoxCoder) – Bbox coder for encoding and decoding boxes.
- **num_decoder_layers** (*int*) – The number of transformer decoder layers.
- **transformerlayers** (*dict*) – Config for transformer decoder.
- **train_cfg** (*dict*) – Config for training.
- **test_cfg** (*dict*) – Config for testing.
- **num_proposal** (*int*) – The number of initial sampling candidates.
- **pred_layer_cfg** (*dict*) – Config of classfication and regression prediction layers.
- **size_cls_agnostic** (*bool*) – Whether the predicted size is class-agnostic.
- **gt_per_seed** (*int*) – the number of candidate instance each point belongs to.
- **sampling_objectness_loss** (*dict*) – Config of initial sampling objectness loss.
- **objectness_loss** (*dict*) – Config of objectness loss.
- **center_loss** (*dict*) – Config of center loss.

- **dir_class_loss** (*dict*) – Config of direction classification loss.
- **dir_res_loss** (*dict*) – Config of direction residual regression loss.
- **size_class_loss** (*dict*) – Config of size classification loss.
- **size_res_loss** (*dict*) – Config of size residual regression loss.
- **size_reg_loss** (*dict*) – Config of class-agnostic size regression loss.
- **semantic_loss** (*dict*) – Config of point-wise semantic segmentation loss.

forward(*feat_dict*, *sample_mod*)

Forward pass.

Note: The forward of GroupFree3DHead is divided into 2 steps:

1. Initial object candidates sampling.
2. Iterative object box prediction by transformer decoder.

Parameters

- **feat_dict** (*dict*) – Feature dict from backbone.
- **sample_mod** (*str*) – sample mode for initial candidates sampling.

Returns Predictions of GroupFree3D head.

Return type results (dict)

get_bboxes(*points*, *bbox_preds*, *input_metas*, *rescale=False*, *use_nms=True*)

Generate bboxes from GroupFree3D head predictions.

Parameters

- **points** (*torch.Tensor*) – Input points.
- **bbox_preds** (*dict*) – Predictions from GroupFree3D head.
- **input_metas** (*list[dict]*) – Point cloud and image's meta info.
- **rescale** (*bool*) – Whether to rescale bboxes.
- **use_nms** (*bool*) – Whether to apply NMS, skip nms postprocessing while using GroupFree3D head in rpn stage.

Returns Bounding boxes, scores and labels.

Return type list[tuple[torch.Tensor]]

get_targets(*points*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*, *bbox_preds=None*, *max_gt_num=64*)

Generate targets of GroupFree3D head.

Parameters

- **points** (*list[torch.Tensor]*) – Points of each batch.
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth bboxes of each batch.
- **gt_labels_3d** (*list[torch.Tensor]*) – Labels of each batch.
- **pts_semantic_mask** (*list[torch.Tensor]*) – Point-wise semantic label of each batch.

- **pts_instance_mask** (*list[torch.Tensor]*) – Point-wise instance label of each batch.
- **bbox_preds** (*torch.Tensor*) – Bounding box predictions of vote head.
- **max_gt_num** (*int*) – Max number of GTs for single batch.

Returns Targets of GroupFree3D head.

Return type tuple[torch.Tensor]

get_targets_single(*points*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*, *max_gt_nums=None*, *seed_points=None*, *seed_indices=None*, *candidate_indices=None*, *seed_points_obj_topk=4*)

Generate targets of GroupFree3D head for single batch.

Parameters

- **points** (*torch.Tensor*) – Points of each batch.
- **gt_bboxes_3d** (BaseInstance3DBoxes) – Ground truth boxes of each batch.
- **gt_labels_3d** (*torch.Tensor*) – Labels of each batch.
- **pts_semantic_mask** (*torch.Tensor*) – Point-wise semantic label of each batch.
- **pts_instance_mask** (*torch.Tensor*) – Point-wise instance label of each batch.
- **max_gt_nums** (*int*) – Max number of GTs for single batch.
- **seed_points** (*torch.Tensor*) – Coordinates of seed points.
- **seed_indices** (*torch.Tensor*) – Indices of seed points.
- **candidate_indices** (*torch.Tensor*) – Indices of object candidates.
- **seed_points_obj_topk** (*int*) – k value of k-Closest Points Sampling.

Returns Targets of GroupFree3D head.

Return type tuple[torch.Tensor]

init_weights()

Initialize weights of transformer decoder in GroupFree3DHead.

loss(*bbox_preds*, *points*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*, *img_metas=None*, *gt_bboxes_ignore=None*, *ret_target=False*)

Compute loss.

Parameters

- **bbox_preds** (*dict*) – Predictions from forward of vote head.
- **points** (*list[torch.Tensor]*) – Input points.
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth bboxes of each sample.
- **gt_labels_3d** (*list[torch.Tensor]*) – Labels of each sample.
- **pts_semantic_mask** (*list[torch.Tensor]*) – Point-wise semantic mask.
- **pts_instance_mask** (*list[torch.Tensor]*) – Point-wise instance mask.
- **img_metas** (*list[dict]*) – Contain pcd and img's meta info.
- **gt_bboxes_ignore** (*list[torch.Tensor]*) – Specify which bounding.
- **ret_target** (*Bool*) – Return targets or not.

Returns Losses of GroupFree3D.

Return type dict

multiclass_nms_single(*obj_scores*, *sem_scores*, *bbox*, *points*, *input_meta*)
Multi-class nms in single batch.

Parameters

- **obj_scores** (*torch.Tensor*) – Objectness score of bounding boxes.
- **sem_scores** (*torch.Tensor*) – semantic class score of bounding boxes.
- **bbox** (*torch.Tensor*) – Predicted bounding boxes.
- **points** (*torch.Tensor*) – Input points.
- **input_meta** (*dict*) – Point cloud and image's meta info.

Returns Bounding boxes, scores and labels.

Return type tuple[torch.Tensor]

class mmdet3d.models.dense_heads.**ImVoxelHead**(*n_classes*, *n_levels*, *n_channels*, *n_reg_outs*, *pts_assign_threshold*, *pts_center_threshold*, *prior_generator*, *center_loss={'type': 'CrossEntropyLoss', 'use_sigmoid': True}*, *bbox_loss={'type': 'RotatedIoU3DLoss'}*, *cls_loss={'type': 'FocalLoss'}*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*)
`ImVoxelNet<https://arxiv.org/abs/2106.01178>`_ head for indoor datasets.

Parameters

- **n_classes** (*int*) – Number of classes.
- **n_levels** (*int*) – Number of feature levels.
- **n_channels** (*int*) – Number of channels in input tensors.
- **n_reg_outs** (*int*) – Number of regression layer channels.
- **pts_assign_threshold** (*int*) – Min number of location per box to be assigned with.
- **pts_center_threshold** (*int*) – Max number of locations per box to be assigned with.
- **center_loss** (*dict, optional*) – Config of centerness loss. Default: dict(type='CrossEntropyLoss', use_sigmoid=True).
- **bbox_loss** (*dict, optional*) – Config of bbox loss. Default: dict(type='RotatedIoU3DLoss').
- **cls_loss** (*dict, optional*) – Config of classification loss. Default: dict(type='FocalLoss').
- **train_cfg** (*dict, optional*) – Config for train stage. Defaults to None.
- **test_cfg** (*dict, optional*) – Config for test stage. Defaults to None.
- **init_cfg** (*dict, optional*) – Config for weight initialization. Defaults to None.

forward(*x*)
Forward function.

Parameters **x** (*list[Tensor]*) – Features from 3d neck.

Returns Centerness, bbox and classification predictions.

Return type tuple[Tensor]

get_bboxes(*center_preds*, *bbox_preds*, *cls_preds*, *valid_pred*, *img_metas*)
Generate boxes for all scenes.

Parameters

- **center_preds** (*list[list[Tensor]]*) – Centerness predictions for all scenes.
- **bbox_preds** (*list[list[Tensor]]*) – Bbox predictions for all scenes.
- **cls_preds** (*list[list[Tensor]]*) – Classification predictions for all scenes.
- **valid_pred** (*Tensor*) – Valid mask prediction for all scenes.
- **img_metas** (*list[dict]*) – Meta infos for all scenes.

Returns

Predicted bboxes, scores, and labels for all scenes.

Return type list[tuple[Tensor]]

init_weights()
Initialize all layer weights.

loss(*center_preds*, *bbox_preds*, *cls_preds*, *valid_pred*, *gt_bboxes*, *gt_labels*, *img_metas*)
Per scene loss function.

Parameters

- **center_preds** (*list[list[Tensor]]*) – Centerness predictions for all scenes.
- **bbox_preds** (*list[list[Tensor]]*) – Bbox predictions for all scenes.
- **cls_preds** (*list[list[Tensor]]*) – Classification predictions for all scenes.
- **valid_pred** (*Tensor*) – Valid mask prediction for all scenes.
- **gt_bboxes** (*list[BaseInstance3DBoxes]*) – Ground truth boxes for all scenes.
- **gt_labels** (*list[Tensor]*) – Ground truth labels for all scenes.
- **img_metas** (*list[dict]*) – Meta infos for all scenes.

Returns Centerness, bbox, and classification loss values.

Return type dict

class mmdet3d.models.dense_heads.**MonoFlexHead**(*num_classes*, *in_channels*, *use_edge_fusion*, *edge_fusion_inds*, *edge_heatmap_ratio*, *filter_outside_objs=True*, *loss_cls={'loss_weight': 1.0, 'type': 'GaussianFocalLoss'}*, *loss_bbox={'loss_weight': 0.1, 'type': 'IoULoss'}*, *loss_dir={'loss_weight': 0.1, 'type': 'MultiBinLoss'}*, *loss_keypoints={'loss_weight': 0.1, 'type': 'L1Loss'}*, *loss_dims={'loss_weight': 0.1, 'type': 'L1Loss'}*, *loss_offsets2d={'loss_weight': 0.1, 'type': 'L1Loss'}*, *loss_direct_depth={'loss_weight': 0.1, 'type': 'L1Loss'}*, *loss_keypoints_depth={'loss_weight': 0.1, 'type': 'L1Loss'}*, *loss_combined_depth={'loss_weight': 0.1, 'type': 'L1Loss'}*, *loss_attr=None*, *bbox_coder={'code_size': 7, 'type': 'MonoFlexCoder'}*, *norm_cfg={'type': 'BN'}*, *init_cfg=None*, *init_bias=- 2.19*, ***kwargs*)
MonoFlex head used in MonoFlex

```
        / --> 3 x 3 conv --> 1 x 1 conv --> [edge fusion] --> cls
        |
        | --> 3 x 3 conv --> 1 x 1 conv --> 2d bbox
        |
        | --> 3 x 3 conv --> 1 x 1 conv --> [edge fusion] --> 2d offsets
        |
        | --> 3 x 3 conv --> 1 x 1 conv -->  keypoints offsets
        |
        | --> 3 x 3 conv --> 1 x 1 conv -->  keypoints uncertainty
feature
        | --> 3 x 3 conv --> 1 x 1 conv -->  keypoints uncertainty
        |
        | --> 3 x 3 conv --> 1 x 1 conv -->   3d dimensions
        |
        |                   |--- 1 x 1 conv -->  ori cls
        | --> 3 x 3 conv --|
        |                   |--- 1 x 1 conv -->  ori offsets
        |
        | --> 3 x 3 conv --> 1 x 1 conv -->  depth
        |
        \ --> 3 x 3 conv --> 1 x 1 conv -->  depth uncertainty
```

Parameters

- **use_edge_fusion** (*bool*) – Whether to use edge fusion module while feature extraction.
- **edge_fusion_inds** (*list[tuple]*) – Indices of feature to use edge fusion.
- **edge_heatmap_ratio** (*float*) – Ratio of generating target heatmap.
- **filter_outside_objs** (*bool, optional*) – Whether to filter the outside objects. Default: True.
- **loss_cls** (*dict, optional*) – Config of classification loss. Default: loss_cls=dict(type='GaussionFocalLoss', loss_weight=1.0).
- **loss_bbox** (*dict, optional*) – Config of localization loss. Default: loss_bbox=dict(type='IOULoss', loss_weight=10.0).
- **loss_dir** (*dict, optional*) – Config of direction classification loss. Default: dict(type='MultibinLoss', loss_weight=0.1).
- **loss_keypoints** (*dict, optional*) – Config of keypoints loss. Default: dict(type='L1Loss', loss_weight=0.1).
- **loss_dims** – (dict, optional): Config of dimensions loss. Default: dict(type='L1Loss', loss_weight=0.1).
- **loss_offsets2d** – (dict, optional): Config of offsets2d loss. Default: dict(type='L1Loss', loss_weight=0.1).
- **loss_direct_depth** – (dict, optional): Config of directly regression depth loss. Default: dict(type='L1Loss', loss_weight=0.1).
- **loss_keypoints_depth** – (dict, optional): Config of keypoints decoded depth loss. Default: dict(type='L1Loss', loss_weight=0.1).
- **loss_combined_depth** – (dict, optional): Config of combined depth loss. Default: dict(type='L1Loss', loss_weight=0.1).

- **loss_attr** (*dict, optional*) – Config of attribute classification loss. In MonoFlex, Default: None.
- **bbox_coder** (*dict, optional*) – Bbox coder for encoding and decoding boxes. Default: dict(type='MonoFlexCoder', code_size=7).
- **norm_cfg** (*dict, optional*) – Dictionary to construct and config norm layer. Default: norm_cfg=dict(type='GN', num_groups=32, requires_grad=True).
- **init_cfg** (*dict*) – Initialization config dict. Default: None.

decode_heatmap(*cls_score*, *reg_pred*, *input_metas*, *cam2imgs*, *topk=100*, *kernel=3*)

Transform outputs into detections raw bbox predictions.

Parameters

- **class_score** (*Tensor*) – Center predict heatmap, shape (B, num_classes, H, W).
- **reg_pred** (*Tensor*) – Box regression map. shape (B, channel, H , W).
- **input_metas** (*List[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **cam2imgs** (*Tensor*) – Camera intrinsic matrix. shape (N, 4, 4)
- **topk** (*int, optional*) – Get top k center keypoints from heatmap. Default 100.
- **kernel** (*int, optional*) – Max pooling kernel for extract local maximum pixels. Default 3.

Returns

Decoded output of SMOKEHead, containing

the following Tensors:

- **batch_bboxes (Tensor): Coords of each 3D box.** shape (B, k, 7)
- **batch_scores (Tensor): Scores of each 3D box.** shape (B, k)
- **batch_topk_labels (Tensor): Categories of each 3D box.** shape (B, k)

Return type tuple[torch.Tensor]

forward(*feats*, *input_metas*)

Forward features from the upstream network.

Parameters

- **feats** (*list[Tensor]*) – Features from the upstream network, each is a 4D-tensor.
- **input_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.

Returns

cls_scores (list[Tensor]): Box scores for each scale level, each is a 4D-tensor, the channel number is num_points * num_classes.

bbox_preds (list[Tensor]): Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_points * bbox_code_size.

Return type tuple

forward_single(*x*, *input_metas*)

Forward features of a single scale level.

Parameters

- **x** (*Tensor*) – Feature maps from a specific FPN feature level.
- **input_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.

Returns Scores for each class, bbox predictions.

Return type tuple

forward_train(*x*, *input_metas*, *gt_bboxes*, *gt_labels*, *gt_bboxes_3d*, *gt_labels_3d*, *centers2d*, *depths*, *attr_labels*, *gt_bboxes_ignore*, *proposal_cfg*, ***kwargs*)

Parameters

- **x** (*list[Tensor]*) – Features from FPN.
- **input_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes of the image, shape (num_gts, 4).
- **gt_labels** (*list[Tensor]*) – Ground truth labels of each box, shape (num_gts,).
- **gt_bboxes_3d** (*list[Tensor]*) – 3D ground truth bboxes of the image, shape (num_gts, self.bbox_code_size).
- **gt_labels_3d** (*list[Tensor]*) – 3D ground truth labels of each box, shape (num_gts,).
- **centers2d** (*list[Tensor]*) – Projected 3D center of each box, shape (num_gts, 2).
- **depths** (*list[Tensor]*) – Depth of projected 3D center of each box, shape (num_gts,).
- **attr_labels** (*list[Tensor]*) – Attribute labels of each box, shape (num_gts,).
- **gt_bboxes_ignore** (*list[Tensor]*) – Ground truth bboxes to be ignored, shape (num_ignored_gts, 4).
- **proposal_cfg** (*mmcv.Config*) – Test / postprocessing configuration, if None, test_cfg would be used

Returns losses: (dict[str, Tensor]): A dictionary of loss components. proposal_list (list[Tensor]): Proposals of each image.

Return type tuple

get_bboxes(*cls_scores*, *bbox_preds*, *input_metas*)

Generate bboxes from bbox head predictions.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level.
- **bbox_preds** (*list[Tensor]*) – Box regression for each scale.
- **input_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **rescale** (*bool*) – If True, return boxes in original image space.

Returns Each item in result_list is 4-tuple.

Return type list[tuple[CameraInstance3DBoxes, Tensor, Tensor, None]]

get_predictions(*pred_reg*, *labels3d*, *centers2d*, *reg_mask*, *batch_indices*, *input_metas*, *downsample_ratio*)
Prepare predictions for computing loss.

Parameters

- **pred_reg** (*Tensor*) – Box regression map. shape (B, channel, H , W).
- **labels3d** (*Tensor*) – Labels of each 3D box. shape (B * max_objs,)
- **centers2d** (*Tensor*) – Coords of each projected 3D box center on image. shape (N, 2)
- **reg_mask** (*Tensor*) – Indexes of the existence of the 3D box. shape (B * max_objs,)
- **batch_indices** (*Tenosr*) – Batch indices of the 3D box. shape (N, 3)
- **input_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **downsample_ratio** (*int*) – The stride of feature map.

Returns The predictions for computing loss.

Return type dict

get_targets(*gt_bboxes_list*, *gt_labels_list*, *gt_bboxes_3d_list*, *gt_labels_3d_list*, *centers2d_list*, *depths_list*, *feat_shape*, *img_shape*, *input_metas*)
Get training targets for batch images. ``

Args:

gt_bboxes_list (list[Tensor]): Ground truth bboxes of each image, shape (num_gt, 4).

gt_labels_list (list[Tensor]): Ground truth labels of each box, shape (num_gt,).

gt_bboxes_3d_list (list[CameraInstance3DBoxes]): 3D Ground truth bboxes of each image, shape (num_gt, bbox_code_size).

gt_labels_3d_list (list[Tensor]): 3D Ground truth labels of each box, shape (num_gt,).

centers2d_list (list[Tensor]): Projected 3D centers onto 2D image, shape (num_gt, 2).

depths_list (list[Tensor]): Depth of projected 3D centers onto 2D image, each has shape (num_gt, 1).

feat_shape (tuple[int]): Feature map shape with value, shape (B, _, H, W).

img_shape (tuple[int]): Image shape in [h, w] format. input_metas (list[dict]): Meta information of each image, e.g.,

image size, scaling factor, etc.

Returns:

tuple[Tensor, dict]: The Tensor value is the targets of

center heatmap, the dict has components below:

- **base_centers2d_target (Tensor): Coords of each projected 3D box** center on image. shape (B * max_objs, 2), [dtype: int]
- **labels3d (Tensor): Labels of each 3D box.** shape (N,)
- **reg_mask (Tensor): Mask of the existence of the 3D box.** shape (B * max_objs,)

- **batch_indices (Tensor): Batch id of the 3D box.** shape (N,)
- **depth_target (Tensor): Depth target of each 3D box.** shape (N,)
- **keypoints2d_target (Tensor): Keypoints of each projected 3D box** on image. shape (N, 10, 2)
- **keypoints_mask (Tensor): Keypoints mask of each projected 3D** box on image. shape (N, 10)
- **keypoints_depth_mask (Tensor): Depths decoded from keypoints** of each 3D box. shape (N, 3)
- **orientations_target (Tensor): Orientation (encoded local yaw)** target of each 3D box. shape (N,)
- **offsets2d_target (Tensor): Offsets target of each projected** 3D box. shape (N, 2)
- **dimensions_target (Tensor): Dimensions target of each 3D box.** shape (N, 3)
- downsample_ratio (int): The stride of feature map.

`init_weights`()
Initialize weights.

`loss`(*cls_scores*, *bbox_preds*, *gt_bboxes*, *gt_labels*, *gt_bboxes_3d*, *gt_labels_3d*, *centers2d*, *depths*, *attr_labels*, *input_metas*, *gt_bboxes_ignore=None*)
Compute loss of the head.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level. shape (num_gt, 4).
- **bbox_preds** (*list[Tensor]*) – Box dims is a 4D-tensor, the channel number is bbox_code_size. shape (B, 7, H, W).
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image. shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **gt_labels** (*list[Tensor]*) – Class indices corresponding to each box. shape (num_gts,).
- **gt_bboxes_3d** (list[CameraInstance3DBoxes]) – 3D boxes ground truth. it is the flipped gt_bboxes
- **gt_labels_3d** (*list[Tensor]*) – Same as gt_labels.
- **centers2d** (*list[Tensor]*) – 2D centers on the image. shape (num_gts, 2).
- **depths** (*list[Tensor]*) – Depth ground truth. shape (num_gts,).
- **attr_labels** (*list[Tensor]*) – Attributes indices of each box. In kitti it's None.
- **input_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]

class mmdet3d.models.dense_heads.**PGDHead**(*use_depth_classifier=True*, *use_onlyreg_proj=False*, *weight_dim=- 1*, *weight_branch=((256))*, *depth_branch=(64)*, *depth_range=(0, 70)*, *depth_unit=10*, *division='uniform'*, *depth_bins=8*, *loss_depth={'beta': 0.1111111111111111, 'loss_weight': 1.0, 'type': 'SmoothL1Loss'}*, *loss_bbox2d={'beta': 0.1111111111111111, 'loss_weight': 1.0, 'type': 'SmoothL1Loss'}*, *loss_consistency={'loss_weight': 1.0, 'type': 'GIoULoss'}*, *pred_bbox2d=True*, *pred_keypoints=False*, *bbox_coder={'base_depths': ((28.01, 16.32)), 'base_dims': ((0.8, 1.73, 0.6), (1.76, 1.73, 0.6), (3.9, 1.56, 1.6)), 'code_size': 7, 'type': 'PGDBBoxCoder'}*, ***kwargs*)

Anchor-free head used in PGD.

Parameters

- **use_depth_classifer** (*bool, optional*) – Whether to use depth classifier. Defaults to True.
- **use_only_reg_proj** (*bool, optional*) – Whether to use only direct regressed depth in the re-projection (to make the network easier to learn). Defaults to False.
- **weight_dim** (*int, optional*) – Dimension of the location-aware weight map. Defaults to -1.
- **weight_branch** (*tuple[tuple[int]], optional*) – Feature map channels of the convolutional branch for weight map. Defaults to ((256,),).
- **depth_branch** (*tuple[int], optional*) – Feature map channels of the branch for probabilistic depth estimation. Defaults to (64,),
- **depth_range** (*tuple[float], optional*) – Range of depth estimation. Defaults to (0, 70),
- **depth_unit** (*int, optional*) – Unit of depth range division. Defaults to 10.
- **division** (*str, optional*) – Depth division method. Options include 'uniform', 'linear', 'log', 'loguniform'. Defaults to 'uniform'.
- **depth_bins** (*int, optional*) – Discrete bins of depth division. Defaults to 8.
- **loss_depth** (*dict, optional*) – Depth loss. Defaults to dict(type='SmoothL1Loss', beta=1.0 / 9.0, loss_weight=1.0).
- **loss_bbox2d** (*dict, optional*) – Loss for 2D box estimation. Defaults to dict(type='SmoothL1Loss', beta=1.0 / 9.0, loss_weight=1.0).
- **loss_consistency** (*dict, optional*) – Consistency loss. Defaults to dict(type='GIoULoss', loss_weight=1.0),
- **pred_velo** (*bool, optional*) – Whether to predict velocity. Defaults to False.
- **pred_bbox2d** (*bool, optional*) – Whether to predict 2D bounding boxes. Defaults to True.
- **pred_keypoints** (*bool, optional*) – Whether to predict keypoints. Defaults to False,
- **bbox_coder** (*dict, optional*) – Bounding box coder. Defaults to dict(type='PGDBBoxCoder', base_depths=((28.01, 16.32),), base_dims=((0.8, 1.73, 0.6), (1.76, 1.73, 0.6), (3.9, 1.56, 1.6)), code_size=7).

forward(*feats*)

Forward features from the upstream network.

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

Returns

cls_scores (list[Tensor]): Box scores for each scale level, each is a 4D-tensor, the channel number is num_points * num_classes.

bbox_preds (list[Tensor]): Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_points * bbox_code_size.

dir_cls_preds (list[Tensor]): Box scores for direction class predictions on each scale level, each is a 4D-tensor, the channel number is num_points * 2. (bin = 2).

weight (list[Tensor]): Location-aware weight maps on each scale level, each is a 4D-tensor, the channel number is num_points * 1.

depth_cls_preds (list[Tensor]): Box scores for depth class predictions on each scale level, each is a 4D-tensor, the channel number is num_points * self.num_depth_cls.

attr_preds (list[Tensor]): Attribute scores for each scale level, each is a 4D-tensor, the channel number is num_points * num_attrs.

centernesses (list[Tensor]): Centerness for each scale level, each is a 4D-tensor, the channel number is num_points * 1.

Return type tuple

forward_single(*x*, *scale*, *stride*)

Forward features of a single scale level.

Parameters

- **x** (*Tensor*) – FPN feature maps of the specified stride.
- **(** (*scale*) – obj: *mmcv.cnn.Scale*): Learnable scale module to resize the bbox prediction.
- **stride** (*int*) – The corresponding stride for feature maps, only used to normalize the bbox prediction when self.norm_on_bbox is True.

Returns

scores for each class, bbox and direction class predictions, depth class predictions, location-aware weights, attribute and centerness predictions of input feature maps.

Return type tuple

get_bboxes(*cls_scores*, *bbox_preds*, *dir_cls_preds*, *depth_cls_preds*, *weights*, *attr_preds*, *centernesses*, *img_metas*, *cfg=None*, *rescale=None*)

Transform network output for a batch into bbox predictions.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_points * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, num_points * 4, H, W)
- **dir_cls_preds** (*list[Tensor]*) – Box scores for direction class predictions on each scale level, each is a 4D-tensor, the channel number is num_points * 2. (bin = 2)

- **depth_cls_preds** (*list[Tensor]*) – Box scores for direction class predictions on each scale level, each is a 4D-tensor, the channel number is num_points * self.num_depth_cls.
- **weights** (*list[Tensor]*) – Location-aware weights for each scale level, each is a 4D-tensor, the channel number is num_points * self.weight_dim.
- **attr_preds** (*list[Tensor]*) – Attribute scores for each scale level Has shape (N, num_points * num_attrs, H, W)
- **centernesses** (*list[Tensor]*) – Centerness for each scale level with shape (N, num_points * 1, H, W)
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **cfg** (*mmcv.Config, optional*) – Test / postprocessing configuration, if None, test_cfg would be used. Defaults to None.
- **rescale** (*bool, optional*) – If True, return boxes in original image space. Defaults to None.

Returns

Each item in result_list is a tuple, which consists of predicted 3D boxes, scores, labels, attributes and 2D boxes (if necessary).

Return type list[tuple[Tensor]]

get_pos_predictions(*bbox_preds*, *dir_cls_preds*, *depth_cls_preds*, *weights*, *attr_preds*, *centernesses*, *pos_inds*, *img_metas*)

Flatten predictions and get positive ones.

Parameters

- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_points * bbox_code_size.
- **dir_cls_preds** (*list[Tensor]*) – Box scores for direction class predictions on each scale level, each is a 4D-tensor, the channel number is num_points * 2. (bin = 2)
- **depth_cls_preds** (*list[Tensor]*) – Box scores for direction class predictions on each scale level, each is a 4D-tensor, the channel number is num_points * self.num_depth_cls.
- **attr_preds** (*list[Tensor]*) – Attribute scores for each scale level, each is a 4D-tensor, the channel number is num_points * num_attrs.
- **centernesses** (*list[Tensor]*) – Centerness for each scale level, each is a 4D-tensor, the channel number is num_points * 1.
- **pos_inds** (*Tensor*) – Index of foreground points from flattened tensors.
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.

Returns

Box predictions, direction classes, probabilistic depth maps, location-aware weight maps, attributes and centerness predictions.

Return type tuple[Tensor]

get_proj_bbox2d(*bbox_preds*, *pos_dir_cls_preds*, *labels_3d*, *bbox_targets_3d*, *pos_points*, *pos_inds*, *img_metas*, *pos_depth_cls_preds=None*, *pos_weights=None*, *pos_cls_scores=None*, *with_kpts=False*)

Decode box predictions and get projected 2D attributes.

Parameters

- **bbox_preds** (*list[Tensor]*) – Box predictions for each scale level, each is a 4D-tensor, the channel number is num_points * bbox_code_size.
- **pos_dir_cls_preds** (*Tensor*) – Box scores for direction class predictions of positive boxes on all the scale levels in shape (num_pos_points, 2).
- **labels_3d** (*list[Tensor]*) – 3D box category labels for each scale level, each is a 4D-tensor.
- **bbox_targets_3d** (*list[Tensor]*) – 3D box targets for each scale level, each is a 4D-tensor, the channel number is num_points * bbox_code_size.
- **pos_points** (*Tensor*) – Foreground points.
- **pos_inds** (*Tensor*) – Index of foreground points from flattened tensors.
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **pos_depth_cls_preds** (*Tensor, optional*) – Probabilistic depth map of positive boxes on all the scale levels in shape (num_pos_points, self.num_depth_cls). Defaults to None.
- **pos_weights** (*Tensor, optional*) – Location-aware weights of positive boxes in shape (num_pos_points, self.weight_dim). Defaults to None.
- **pos_cls_scores** (*Tensor, optional*) – Classification scores of positive boxes in shape (num_pos_points, self.num_classes). Defaults to None.
- **with_kpts** (*bool, optional*) – Whether to output keypoints targets. Defaults to False.

Returns

Exterior 2D boxes from projected 3D boxes, predicted 2D boxes and keypoint targets (if necessary).

Return type tuple[Tensor]

get_targets(*points*, *gt_bboxes_list*, *gt_labels_list*, *gt_bboxes_3d_list*, *gt_labels_3d_list*, *centers2d_list*, *depths_list*, *attr_labels_list*)

Compute regression, classification and centerss targets for points in multiple images.

Parameters

- **points** (*list[Tensor]*) – Points of each fpn level, each has shape (num_points, 2).
- **gt_bboxes_list** (*list[Tensor]*) – Ground truth bboxes of each image, each has shape (num_gt, 4).
- **gt_labels_list** (*list[Tensor]*) – Ground truth labels of each box, each has shape (num_gt,).
- **gt_bboxes_3d_list** (*list[Tensor]*) – 3D Ground truth bboxes of each image, each has shape (num_gt, bbox_code_size).
- **gt_labels_3d_list** (*list[Tensor]*) – 3D Ground truth labels of each box, each has shape (num_gt,).

- **centers2d_list** (*list[Tensor]*) – Projected 3D centers onto 2D image, each has shape (num_gt, 2).
- **depths_list** (*list[Tensor]*) – Depth of projected 3D centers onto 2D image, each has shape (num_gt, 1).
- **attr_labels_list** (*list[Tensor]*) – Attribute labels of each box, each has shape (num_gt,).

Returns concat_lvl_labels (list[Tensor]): Labels of each level. concat_lvl_bbox_targets (list[Tensor]): BBox targets of each level.

Return type tuple

init_weights()
Initialize weights of the head.

We currently still use the customized defined init_weights because the default init of DCN triggered by the init_cfg will init conv_offset.weight, which mistakenly affects the training stability.

loss(*cls_scores*, *bbox_preds*, *dir_cls_preds*, *depth_cls_preds*, *weights*, *attr_preds*, *centernesses*, *gt_bboxes*, *gt_labels*, *gt_bboxes_3d*, *gt_labels_3d*, *centers2d*, *depths*, *attr_labels*, *img_metas*, *gt_bboxes_ignore=None*)
Compute loss of the head.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level, each is a 4D-tensor, the channel number is num_points * num_classes.
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_points * bbox_code_size.
- **dir_cls_preds** (*list[Tensor]*) – Box scores for direction class predictions on each scale level, each is a 4D-tensor, the channel number is num_points * 2. (bin = 2)
- **depth_cls_preds** (*list[Tensor]*) – Box scores for direction class predictions on each scale level, each is a 4D-tensor, the channel number is num_points * self.num_depth_cls.
- **weights** (*list[Tensor]*) – Location-aware weights for each scale level, each is a 4D-tensor, the channel number is num_points * self.weight_dim.
- **attr_preds** (*list[Tensor]*) – Attribute scores for each scale level, each is a 4D-tensor, the channel number is num_points * num_attrs.
- **centernesses** (*list[Tensor]*) – Centerness for each scale level, each is a 4D-tensor, the channel number is num_points * 1.
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box
- **gt_bboxes_3d** (*list[Tensor]*) – 3D boxes ground truth with shape of (num_gts, code_size).
- **gt_labels_3d** (*list[Tensor]*) – same as gt_labels
- **centers2d** (*list[Tensor]*) – 2D centers on the image with shape of (num_gts, 2).
- **depths** (*list[Tensor]*) – Depth ground truth with shape of (num_gts,).
- **attr_labels** (*list[Tensor]*) – Attributes indices of each box.

- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **gt_bboxes_ignore** (*list[Tensor]*) – specify which bounding boxes can be ignored when computing the loss. Defaults to None.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

class mmdet3d.models.dense_heads.**PartA2RPNHead**(*num_classes*, *in_channels*, *train_cfg*, *test_cfg*, *feat_channels=256*, *use_direction_classifier=True*, *anchor_generator={'custom_values': [], 'range': [0, -39.68, - 1.78, 69.12, 39.68, - 1.78], 'reshape_out': False, 'rotations': [0, 1.57], 'sizes': [[3.9, 1.6, 1.56]], 'strides': [2], 'type': 'Anchor3DRangeGenerator'}*, *assigner_per_size=False*, *assign_per_class=False*, *diff_rad_by_sin=True*, *dir_offset=-1.5707963267948966*, *dir_limit_offset=0*, *bbox_coder={'type': 'DeltaXYZWLHRBBoxCoder'}*, *loss_cls={'loss_weight': 1.0, 'type': 'CrossEntropyLoss', 'use_sigmoid': True}*, *loss_bbox={'beta': 0.1111111111111111, 'loss_weight': 2.0, 'type': 'SmoothL1Loss'}*, *loss_dir={'loss_weight': 0.2, 'type': 'CrossEntropyLoss'}*, *init_cfg=None*)

RPN head for PartA2.

Note: The main difference between the PartA2 RPN head and the Anchor3DHead lies in their output during inference. PartA2 RPN head further returns the original classification score for the second stage since the bbox head in RoI head does not do classification task.

Different from RPN heads in 2D detectors, this RPN head does multi-class classification task and uses FocalLoss like the SECOND and PointPillars do. But this head uses class agnostic nms rather than multi-class nms.

Parameters

- **num_classes** (*int*) – Number of classes.
- **in_channels** (*int*) – Number of channels in the input feature map.
- **train_cfg** (*dict*) – Train configs.
- **test_cfg** (*dict*) – Test configs.
- **feat_channels** (*int*) – Number of channels of the feature map.
- **use_direction_classifier** (*bool*) – Whether to add a direction classifier.
- **anchor_generator** (*dict*) – Config dict of anchor generator.
- **assigner_per_size** (*bool*) – Whether to do assignment for each separate anchor size.
- **assign_per_class** (*bool*) – Whether to do assignment for each class.
- **diff_rad_by_sin** (*bool*) – Whether to change the difference into sin difference for box regression loss.
- **dir_offset** (*float | int*) – The offset of BEV rotation angles (TODO: may be moved into box coder)

- **dir_limit_offset** (*float | int*) – The limited range of BEV rotation angles. (TODO: may be moved into box coder)
- **bbox_coder** (*dict*) – Config dict of box coders.
- **loss_cls** (*dict*) – Config of classification loss.
- **loss_bbox** (*dict*) – Config of localization loss.
- **loss_dir** (*dict*) – Config of direction classifier loss.

class_agnostic_nms(*mlvl_bboxes*, *mlvl_bboxes_for_nms*, *mlvl_max_scores*, *mlvl_label_pred*, *mlvl_cls_score*, *mlvl_dir_scores*, *score_thr*, *max_num*, *cfg*, *input_meta*)

Class agnostic nms for single batch.

Parameters

- **mlvl_bboxes** (*torch.Tensor*) – Bboxes from Multi-level.
- **mlvl_bboxes_for_nms** (*torch.Tensor*) – Bboxes for nms (bev or minmax boxes) from Multi-level.
- **mlvl_max_scores** (*torch.Tensor*) – Max scores of Multi-level bbox.
- **mlvl_label_pred** (*torch.Tensor*) – Class predictions of Multi-level bbox.
- **mlvl_cls_score** (*torch.Tensor*) – Class scores of Multi-level bbox.
- **mlvl_dir_scores** (*torch.Tensor*) – Direction scores of Multi-level bbox.
- **score_thr** (*int*) – Score threshold.
- **max_num** (*int*) – Max number of bboxes after nms.
- **cfg** (ConfigDict) – Training or testing config.
- **input_meta** (*dict*) – Contain pcd and img's meta info.

Returns

Predictions of single batch. Contain the keys:

- boxes_3d (BaseInstance3DBoxes): Predicted 3d bboxes.
- scores_3d (torch.Tensor): Score of each bbox.
- labels_3d (torch.Tensor): Label of each bbox.
- cls_preds (torch.Tensor): Class score of each bbox.

Return type dict

get_bboxes_single(*cls_scores*, *bbox_preds*, *dir_cls_preds*, *mlvl_anchors*, *input_meta*, *cfg*, *rescale=False*)

Get bboxes of single branch.

Parameters

- **cls_scores** (*torch.Tensor*) – Class score in single batch.
- **bbox_preds** (*torch.Tensor*) – Bbox prediction in single batch.
- **dir_cls_preds** (*torch.Tensor*) – Predictions of direction class in single batch.
- **mlvl_anchors** (*List[torch.Tensor]*) – Multi-level anchors in single batch.
- **input_meta** (*list[dict]*) – Contain pcd and img's meta info.
- **cfg** (ConfigDict) – Training or testing config.

- **rescale** (*list[torch.Tensor]*) – whether th rescale bbox.

Returns

Predictions of single batch containing the following keys:

- boxes_3d (BaseInstance3DBoxes): Predicted 3d bboxes.
- scores_3d (torch.Tensor): Score of each bbox.
- labels_3d (torch.Tensor): Label of each bbox.
- cls_preds (torch.Tensor): Class score of each bbox.

Return type dict

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

Calculate losses.

Parameters

- **cls_scores** (*list[torch.Tensor]*) – Multi-level class scores.
- **bbox_preds** (*list[torch.Tensor]*) – Multi-level bbox predictions.
- **dir_cls_preds** (*list[torch.Tensor]*) – Multi-level direction class predictions.
- **gt_bboxes** (list[BaseInstance3DBoxes]) – Ground truth boxes of each sample.
- **gt_labels** (*list[torch.Tensor]*) – Labels of each sample.
- **input_metas** (*list[dict]*) – Point cloud and image's meta info.
- **gt_bboxes_ignore** (*list[torch.Tensor]*) – Specify which bounding.

Returns

Classification, bbox, and direction losses of each level.

- loss_rpn_cls (list[torch.Tensor]): Classification losses.
- loss_rpn_bbox (list[torch.Tensor]): Box regression losses.
- **loss_rpn_dir (list[torch.Tensor]): Direction classification** losses.

Return type dict[str, list[torch.Tensor]]

class mmdet3d.models.dense_heads.**PointRPNHead**(*num_classes*, *train_cfg*, *test_cfg*, *pred_layer_cfg=None*, *enlarge_width=0.1*, *cls_loss=None*, *bbox_loss=None*, *bbox_coder=None*, *init_cfg=None*)

RPN module for PointRCNN.

Parameters

- **num_classes** (*int*) – Number of classes.
- **train_cfg** (*dict*) – Train configs.
- **test_cfg** (*dict*) – Test configs.
- **pred_layer_cfg** (*dict, optional*) – Config of classification and regression prediction layers. Defaults to None.
- **enlarge_width** (*float, optional*) – Enlarge bbox for each side to ignore close points. Defaults to 0.1.
- **cls_loss** (*dict, optional*) – Config of direction classification loss. Defaults to None.
- **bbox_loss** (*dict, optional*) – Config of localization loss. Defaults to None.

- **bbox_coder** (*dict, optional*) – Config dict of box coders. Defaults to None.
- **init_cfg** (*dict, optional*) – Config of initialization. Defaults to None.

class_agnostic_nms(*obj_scores*, *sem_scores*, *bbox*, *points*, *input_meta*)
Class agnostic nms.

Parameters

- **obj_scores** (*torch.Tensor*) – Objectness score of bounding boxes.
- **sem_scores** (*torch.Tensor*) – Semantic class score of bounding boxes.
- **bbox** (*torch.Tensor*) – Predicted bounding boxes.

Returns Bounding boxes, scores and labels.

Return type tuple[torch.Tensor]

forward(*feat_dict*)
Forward pass.

Parameters **feat_dict** (*dict*) – Feature dict from backbone.

Returns

Predicted boxes and classification scores.

Return type tuple[list[torch.Tensor]]

get_bboxes(*points*, *bbox_preds*, *cls_preds*, *input_metas*, *rescale=False*)
Generate bboxes from RPN head predictions.

Parameters

- **points** (*torch.Tensor*) – Input points.
- **bbox_preds** (*dict*) – Regression predictions from PointRCNN head.
- **cls_preds** (*dict*) – Class scores predictions from PointRCNN head.
- **input_metas** (*list[dict]*) – Point cloud and image's meta info.
- **rescale** (*bool, optional*) – Whether to rescale bboxes. Defaults to False.

Returns Bounding boxes, scores and labels.

Return type list[tuple[torch.Tensor]]

get_targets(*points*, *gt_bboxes_3d*, *gt_labels_3d*)
Generate targets of PointRCNN RPN head.

Parameters

- **points** (*list[torch.Tensor]*) – Points of each batch.
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth bboxes of each batch.
- **gt_labels_3d** (*list[torch.Tensor]*) – Labels of each batch.

Returns Targets of PointRCNN RPN head.

Return type tuple[torch.Tensor]

get_targets_single(*points*, *gt_bboxes_3d*, *gt_labels_3d*)
Generate targets of PointRCNN RPN head for single batch.

Parameters

- **points** (*torch.Tensor*) – Points of each batch.

- **gt_bboxes_3d** (BaseInstance3DBoxes) – Ground truth boxes of each batch.
- **gt_labels_3d** (*torch.Tensor*) – Labels of each batch.

Returns Targets of ssd3d head.

Return type tuple[torch.Tensor]

loss(*bbox_preds*, *cls_preds*, *points*, *gt_bboxes_3d*, *gt_labels_3d*, *img_metas=None*)
Compute loss.

Parameters

- **bbox_preds** (*dict*) – Predictions from forward of PointRCNN RPN_Head.
- **cls_preds** (*dict*) – Classification from forward of PointRCNN RPN_Head.
- **points** (*list[torch.Tensor]*) – Input points.
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth bboxes of each sample.
- **gt_labels_3d** (*list[torch.Tensor]*) – Labels of each sample.
- **img_metas** (*list[dict], Optional*) – Contain pcd and img's meta info. Defaults to None.

Returns Losses of PointRCNN RPN module.

Return type dict

class mmdet3d.models.dense_heads.**SMOKEMono3DHead**(*num_classes*, *in_channels*, *dim_channel*, *ori_channel*, *bbox_coder*, *loss_cls={'loss_weight': 1.0, 'type': 'GaussionFocalLoss'}*, *loss_bbox={'loss_weight': 0.1, 'type': 'L1Loss'}*, *loss_dir=None*, *loss_attr=None*, *norm_cfg={'num_groups': 32, 'requires_grad': True, 'type': 'GN'}*, *init_cfg=None*, ***kwargs*)

Anchor-free head used in SMOKE

```
        /-----> 3*3 conv -----> 1*1 conv -----> cls
feature
        \-----> 3*3 conv -----> 1*1 conv -----> reg
```

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **in_channels** (*int*) – Number of channels in the input feature map.
- **dim_channel** (*list[int]*) – indices of dimension offset preds in regression heatmap channels.
- **ori_channel** (*list[int]*) – indices of orientation offset pred in regression heatmap channels.
- **bbox_coder** (CameraInstance3DBoxes) – Bbox coder for encoding and decoding boxes.
- **loss_cls** (*dict, optional*) – Config of classification loss. Default: loss_cls=dict(type='GaussionFocalLoss', loss_weight=1.0).
- **loss_bbox** (*dict, optional*) – Config of localization loss. Default: loss_bbox=dict(type='L1Loss', loss_weight=10.0).

- **loss_dir** (*dict, optional*) – Config of direction classification loss. In SMOKE, Default: None.
- **loss_attr** (*dict, optional*) – Config of attribute classification loss. In SMOKE, Default: None.
- **loss_centerness** (*dict*) – Config of centerness loss.
- **norm_cfg** (*dict*) – Dictionary to construct and config norm layer. Default: norm_cfg=dict(type='GN', num_groups=32, requires_grad=True).
- **init_cfg** (*dict*) – Initialization config dict. Default: None.

decode_heatmap(*cls_score*, *reg_pred*, *img_metas*, *cam2imgs*, *trans_mats*, *topk=100*, *kernel=3*)

Transform outputs into detections raw bbox predictions.

Parameters

- **class_score** (*Tensor*) – Center predict heatmap, shape (B, num_classes, H, W).
- **reg_pred** (*Tensor*) – Box regression map. shape (B, channel, H , W).
- **img_metas** (*List[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **cam2imgs** (*Tensor*) – Camera intrinsic matrixs. shape (B, 4, 4)
- **trans_mats** (*Tensor*) – Transformation matrix from original image to feature map. shape: (batch, 3, 3)
- **topk** (*int*) – Get top k center keypoints from heatmap. Default 100.
- **kernel** (*int*) – Max pooling kernel for extract local maximum pixels. Default 3.

Returns

Decoded output of SMOKEHead, containing

the following Tensors:

- **batch_bboxes (Tensor): Coords of each 3D box.** shape (B, k, 7)
- **batch_scores (Tensor): Scores of each 3D box.** shape (B, k)
- **batch_topk_labels (Tensor): Categories of each 3D box.** shape (B, k)

Return type tuple[torch.Tensor]

forward(*feats*)

Forward features from the upstream network.

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

Returns

cls_scores (list[Tensor]): Box scores for each scale level, each is a 4D-tensor, the channel number is num_points * num_classes.

bbox_preds (list[Tensor]): Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_points * bbox_code_size.

Return type tuple

forward_single(*x*)

Forward features of a single scale level.

Parameters **x** (*Tensor*) – Input feature map.

Returns Scores for each class, bbox of input feature maps.

Return type tuple

get_bboxes(*cls_scores*, *bbox_preds*, *img_metas*, *rescale=None*)
Generate bboxes from bbox head predictions.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level.
- **bbox_preds** (*list[Tensor]*) – Box regression for each scale.
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **rescale** (*bool*) – If True, return boxes in original image space.

Returns Each item in result_list is 4-tuple.

Return type list[tuple[CameraInstance3DBoxes, Tensor, Tensor, None]]

get_predictions(*labels3d*, *centers2d*, *gt_locations*, *gt_dimensions*, *gt_orientations*, *indices*, *img_metas*, *pred_reg*)
Prepare predictions for computing loss.

Parameters

- **labels3d** (*Tensor*) – Labels of each 3D box. shape (B, max_objs,)
- **centers2d** (*Tensor*) – Coords of each projected 3D box center on image. shape (B * max_objs, 2)
- **gt_locations** (*Tensor*) – Coords of each 3D box's location. shape (B * max_objs, 3)
- **gt_dimensions** (*Tensor*) – Dimensions of each 3D box. shape (N, 3)
- **gt_orientations** (*Tensor*) – Orientation(yaw) of each 3D box. shape (N, 1)
- **indices** (*Tensor*) – Indices of the existence of the 3D box. shape (B * max_objs,)
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **pre_reg** (*Tensor*) – Box regression map. shape (B, channel, H , W).

Returns

the dict has components below: - bbox3d_yaws (CameraInstance3DBoxes):

bbox calculated using pred orientations.

- **bbox3d_dims (CameraInstance3DBoxes):** bbox calculated using pred dimensions.
- **bbox3d_locs (CameraInstance3DBoxes):** bbox calculated using pred locations.

Return type dict

get_targets(*gt_bboxes*, *gt_labels*, *gt_bboxes_3d*, *gt_labels_3d*, *centers2d*, *feat_shape*, *img_shape*, *img_metas*)
Get training targets for batch images.

Parameters

- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes of each image, shape (num_gt, 4).
- **gt_labels** (*list[Tensor]*) – Ground truth labels of each box, shape (num_gt,).
- **gt_bboxes_3d** (list[CameraInstance3DBoxes]) – 3D Ground truth bboxes of each image, shape (num_gt, bbox_code_size).
- **gt_labels_3d** (*list[Tensor]*) – 3D Ground truth labels of each box, shape (num_gt,).
- **centers2d** (*list[Tensor]*) – Projected 3D centers onto 2D image, shape (num_gt, 2).
- **feat_shape** (*tuple[int]*) – Feature map shape with value, shape (B, _, H, W).
- **img_shape** (*tuple[int]*) – Image shape in [h, w] format.
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.

Returns

The Tensor value is the targets of

center heatmap, the dict has components below:

- **gt_centers2d (Tensor): Coords of each projected 3D box** center on image. shape (B * max_objs, 2)
- **gt_labels3d (Tensor): Labels of each 3D box.** shape (B, max_objs,)
- **indices (Tensor): Indices of the existence of the 3D box.** shape (B * max_objs,)
- **affine_indices (Tensor): Indices of the affine of the 3D box.** shape (N,)
- **gt_locs (Tensor): Coords of each 3D box's location.** shape (N, 3)
- **gt_dims (Tensor): Dimensions of each 3D box.** shape (N, 3)
- **gt_yaws (Tensor): Orientation(yaw) of each 3D box.** shape (N, 1)
- **gt_cors (Tensor): Coords of the corners of each 3D box.** shape (N, 8, 3)

Return type tuple[Tensor, dict]

loss(*cls_scores*, *bbox_preds*, *gt_bboxes*, *gt_labels*, *gt_bboxes_3d*, *gt_labels_3d*, *centers2d*, *depths*, *attr_labels*, *img_metas*, *gt_bboxes_ignore=None*)

Compute loss of the head.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level. shape (num_gt, 4).
- **bbox_preds** (*list[Tensor]*) – Box dims is a 4D-tensor, the channel number is bbox_code_size. shape (B, 7, H, W).
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image. shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **gt_labels** (*list[Tensor]*) – Class indices corresponding to each box. shape (num_gts,).
- **gt_bboxes_3d** (list[CameraInstance3DBoxes]) – 3D boxes ground truth. it is the flipped gt_bboxes

- **gt_labels_3d** (*list[Tensor]*) – Same as gt_labels.
- **centers2d** (*list[Tensor]*) – 2D centers on the image. shape (num_gts, 2).
- **depths** (*list[Tensor]*) – Depth ground truth. shape (num_gts,).
- **attr_labels** (*list[Tensor]*) – Attributes indices of each box. In kitti it's None.
- **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]

class mmdet3d.models.dense_heads.**SSD3DHead**(*num_classes*, *bbox_coder*, *in_channels=256*, *train_cfg=None*, *test_cfg=None*, *vote_module_cfg=None*, *vote_aggregation_cfg=None*, *pred_layer_cfg=None*, *conv_cfg={'type': 'Conv1d'}*, *norm_cfg={'type': 'BN1d'}*, *act_cfg={'type': 'ReLU'}*, *objectness_loss=None*, *center_loss=None*, *dir_class_loss=None*, *dir_res_loss=None*, *size_res_loss=None*, *corner_loss=None*, *vote_loss=None*, *init_cfg=None*)

Bbox head of 3DSSD.

Parameters

- **num_classes** (*int*) – The number of class.
- **bbox_coder** (BaseBBoxCoder) – Bbox coder for encoding and decoding boxes.
- **in_channels** (*int*) – The number of input feature channel.
- **train_cfg** (*dict*) – Config for training.
- **test_cfg** (*dict*) – Config for testing.
- **vote_module_cfg** (*dict*) – Config of VoteModule for point-wise votes.
- **vote_aggregation_cfg** (*dict*) – Config of vote aggregation layer.
- **pred_layer_cfg** (*dict*) – Config of classfication and regression prediction layers.
- **conv_cfg** (*dict*) – Config of convolution in prediction layer.
- **norm_cfg** (*dict*) – Config of BN in prediction layer.
- **act_cfg** (*dict*) – Config of activation in prediction layer.
- **objectness_loss** (*dict*) – Config of objectness loss.
- **center_loss** (*dict*) – Config of center loss.
- **dir_class_loss** (*dict*) – Config of direction classification loss.
- **dir_res_loss** (*dict*) – Config of direction residual regression loss.
- **size_res_loss** (*dict*) – Config of size residual regression loss.
- **corner_loss** (*dict*) – Config of bbox corners regression loss.
- **vote_loss** (*dict*) – Config of candidate points regression loss.

get_bboxes(*points*, *bbox_preds*, *input_metas*, *rescale=False*)

Generate bboxes from 3DSSD head predictions.

Parameters

- **points** (*torch.Tensor*) – Input points.
- **bbox_preds** (*dict*) – Predictions from sdd3d head.
- **input_metas** (*list[dict]*) – Point cloud and image's meta info.
- **rescale** (*bool*) – Whether to rescale bboxes.

Returns Bounding boxes, scores and labels.

Return type list[tuple[torch.Tensor]]

get_targets(*points*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*, *bbox_preds=None*)

Generate targets of ssd3d head.

Parameters

- **points** (*list[torch.Tensor]*) – Points of each batch.
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth bboxes of each batch.
- **gt_labels_3d** (*list[torch.Tensor]*) – Labels of each batch.
- **pts_semantic_mask** (*list[torch.Tensor]*) – Point-wise semantic label of each batch.
- **pts_instance_mask** (*list[torch.Tensor]*) – Point-wise instance label of each batch.
- **bbox_preds** (*torch.Tensor*) – Bounding box predictions of ssd3d head.

Returns Targets of ssd3d head.

Return type tuple[torch.Tensor]

get_targets_single(*points*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*, *aggregated_points=None*, *seed_points=None*)

Generate targets of ssd3d head for single batch.

Parameters

- **points** (*torch.Tensor*) – Points of each batch.
- **gt_bboxes_3d** (BaseInstance3DBoxes) – Ground truth boxes of each batch.
- **gt_labels_3d** (*torch.Tensor*) – Labels of each batch.
- **pts_semantic_mask** (*torch.Tensor*) – Point-wise semantic label of each batch.
- **pts_instance_mask** (*torch.Tensor*) – Point-wise instance label of each batch.
- **aggregated_points** (*torch.Tensor*) – Aggregated points from candidate points layer.
- **seed_points** (*torch.Tensor*) – Seed points of candidate points.

Returns Targets of ssd3d head.

Return type tuple[torch.Tensor]

loss(*bbox_preds*, *points*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*, *img_metas=None*, *gt_bboxes_ignore=None*)

Compute loss.

Parameters

- **bbox_preds** (*dict*) – Predictions from forward of SSD3DHead.
- **points** (*list[torch.Tensor]*) – Input points.
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth bboxes of each sample.
- **gt_labels_3d** (*list[torch.Tensor]*) – Labels of each sample.
- **pts_semantic_mask** (*list[torch.Tensor]*) – Point-wise semantic mask.
- **pts_instance_mask** (*list[torch.Tensor]*) – Point-wise instance mask.
- **img_metas** (*list[dict]*) – Contain pcd and img's meta info.
- **gt_bboxes_ignore** (*list[torch.Tensor]*) – Specify which bounding.

Returns Losses of 3DSSD.

Return type dict

multiclass_nms_single(*obj_scores*, *sem_scores*, *bbox*, *points*, *input_meta*)
Multi-class nms in single batch.

Parameters

- **obj_scores** (*torch.Tensor*) – Objectness score of bounding boxes.
- **sem_scores** (*torch.Tensor*) – Semantic class score of bounding boxes.
- **bbox** (*torch.Tensor*) – Predicted bounding boxes.
- **points** (*torch.Tensor*) – Input points.
- **input_meta** (*dict*) – Point cloud and image's meta info.

Returns Bounding boxes, scores and labels.

Return type tuple[torch.Tensor]

class mmdet3d.models.dense_heads.**ShapeAwareHead**(*tasks*, *assign_per_class=True*, *init_cfg=None*, ***kwargs*)
Shape-aware grouping head for SSN.

Parameters

- **tasks** (*dict*) – Shape-aware groups of multi-class objects.
- **assign_per_class** (*bool, optional*) – Whether to do assignment for each class. Default: True.
- **kwargs** (*dict*) – Other arguments are the same as those in *Anchor3DHead*.

forward_single(*x*)
Forward function on a single-scale feature map.

Parameters **x** (*torch.Tensor*) – Input features.

Returns

Contain score of each class, bbox regression and direction classification predictions.

Return type tuple[torch.Tensor]

get_bboxes(*cls_scores*, *bbox_preds*, *dir_cls_preds*, *input_metas*, *cfg=None*, *rescale=False*)
Get bboxes of anchor head.

Parameters

- **cls_scores** (*list[torch.Tensor]*) – Multi-level class scores.
- **bbox_preds** (*list[torch.Tensor]*) – Multi-level bbox predictions.
- **dir_cls_preds** (*list[torch.Tensor]*) – Multi-level direction class predictions.
- **input_metas** (*list[dict]*) – Contain pcd and img's meta info.
- **cfg** (ConfigDict, optional) – Training or testing config. Default: None.
- **rescale** (*list[torch.Tensor], optional*) – Whether to rescale bbox. Default: False.

Returns Prediction resultes of batches.

Return type list[tuple]

get_bboxes_single(*cls_scores*, *bbox_preds*, *dir_cls_preds*, *mlvl_anchors*, *input_meta*, *cfg=None*, *rescale=False*)

Get bboxes of single branch.

Parameters

- **cls_scores** (*torch.Tensor*) – Class score in single batch.
- **bbox_preds** (*torch.Tensor*) – Bbox prediction in single batch.
- **dir_cls_preds** (*torch.Tensor*) – Predictions of direction class in single batch.
- **mlvl_anchors** (*List[torch.Tensor]*) – Multi-level anchors in single batch.
- **input_meta** (*list[dict]*) – Contain pcd and img's meta info.
- **cfg** (ConfigDict) – Training or testing config.
- **rescale** (*list[torch.Tensor], optional*) – whether to rescale bbox. Default: False.

Returns

Contain predictions of single batch.

- bboxes (BaseInstance3DBoxes): Predicted 3d bboxes.
- scores (torch.Tensor): Class score of each bbox.
- labels (torch.Tensor): Label of each bbox.

Return type tuple

init_weights()

Initialize the weights.

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

Calculate losses.

Parameters

- **cls_scores** (*list[torch.Tensor]*) – Multi-level class scores.
- **bbox_preds** (*list[torch.Tensor]*) – Multi-level bbox predictions.
- **dir_cls_preds** (*list[torch.Tensor]*) – Multi-level direction class predictions.
- **gt_bboxes** (list[BaseInstance3DBoxes]) – Gt bboxes of each sample.
- **gt_labels** (*list[torch.Tensor]*) – Gt labels of each sample.
- **input_metas** (*list[dict]*) – Contain pcd and img's meta info.

- **gt_bboxes_ignore** (*list[torch.Tensor]*) – Specify which bounding.

Returns

Classification, bbox, and direction losses of each level.

- loss_cls (list[torch.Tensor]): Classification losses.
- loss_bbox (list[torch.Tensor]): Box regression losses.
- **loss_dir (list[torch.Tensor]): Direction classification** losses.

Return type dict[str, list[torch.Tensor]]

loss_single(*cls_score*, *bbox_pred*, *dir_cls_preds*, *labels*, *label_weights*, *bbox_targets*, *bbox_weights*, *dir_targets*, *dir_weights*, *num_total_samples*)

Calculate loss of Single-level results.

Parameters

- **cls_score** (*torch.Tensor*) – Class score in single-level.
- **bbox_pred** (*torch.Tensor*) – Bbox prediction in single-level.
- **dir_cls_preds** (*torch.Tensor*) – Predictions of direction class in single-level.
- **labels** (*torch.Tensor*) – Labels of class.
- **label_weights** (*torch.Tensor*) – Weights of class loss.
- **bbox_targets** (*torch.Tensor*) – Targets of bbox predictions.
- **bbox_weights** (*torch.Tensor*) – Weights of bbox loss.
- **dir_targets** (*torch.Tensor*) – Targets of direction predictions.
- **dir_weights** (*torch.Tensor*) – Weights of direction loss.
- **num_total_samples** (*int*) – The number of valid samples.

Returns

Losses of class, bbox and direction, respectively.

Return type tuple[torch.Tensor]

class mmdet3d.models.dense_heads.**VoteHead**(*num_classes*, *bbox_coder*, *train_cfg=None*, *test_cfg=None*, *vote_module_cfg=None*, *vote_aggregation_cfg=None*, *pred_layer_cfg=None*, *conv_cfg={'type': 'Conv1d'}*, *norm_cfg={'type': 'BN1d'}*, *objectness_loss=None*, *center_loss=None*, *dir_class_loss=None*, *dir_res_loss=None*, *size_class_loss=None*, *size_res_loss=None*, *semantic_loss=None*, *iou_loss=None*, *init_cfg=None*)

Bbox head of Votenet.

Parameters

- **num_classes** (*int*) – The number of class.
- **bbox_coder** (BaseBBoxCoder) – Bbox coder for encoding and decoding boxes.
- **train_cfg** (*dict*) – Config for training.
- **test_cfg** (*dict*) – Config for testing.
- **vote_module_cfg** (*dict*) – Config of VoteModule for point-wise votes.
- **vote_aggregation_cfg** (*dict*) – Config of vote aggregation layer.

- **pred_layer_cfg** (*dict*) – Config of classfication and regression prediction layers.
- **conv_cfg** (*dict*) – Config of convolution in prediction layer.
- **norm_cfg** (*dict*) – Config of BN in prediction layer.
- **objectness_loss** (*dict*) – Config of objectness loss.
- **center_loss** (*dict*) – Config of center loss.
- **dir_class_loss** (*dict*) – Config of direction classification loss.
- **dir_res_loss** (*dict*) – Config of direction residual regression loss.
- **size_class_loss** (*dict*) – Config of size classification loss.
- **size_res_loss** (*dict*) – Config of size residual regression loss.
- **semantic_loss** (*dict*) – Config of point-wise semantic segmentation loss.

forward(*feat_dict*, *sample_mod*)

Forward pass.

Note: The forward of VoteHead is divided into 4 steps:

1. Generate vote_points from seed_points.
2. Aggregate vote_points.
3. Predict bbox and score.
4. Decode predictions.

Parameters

- **feat_dict** (*dict*) – Feature dict from backbone.
- **sample_mod** (*str*) – Sample mode for vote aggregation layer. valid modes are "vote", "seed", "random" and "spec".

Returns Predictions of vote head.

Return type dict

get_bboxes(*points*, *bbox_preds*, *input_metas*, *rescale=False*, *use_nms=True*)

Generate bboxes from vote head predictions.

Parameters

- **points** (*torch.Tensor*) – Input points.
- **bbox_preds** (*dict*) – Predictions from vote head.
- **input_metas** (*list[dict]*) – Point cloud and image's meta info.
- **rescale** (*bool*) – Whether to rescale bboxes.
- **use_nms** (*bool*) – Whether to apply NMS, skip nms postprocessing while using vote head in rpn stage.

Returns Bounding boxes, scores and labels.

Return type list[tuple[torch.Tensor]]

get_targets(*points*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*, *bbox_preds=None*)
Generate targets of vote head.

Parameters

- **points** (*list[torch.Tensor]*) – Points of each batch.
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth bboxes of each batch.
- **gt_labels_3d** (*list[torch.Tensor]*) – Labels of each batch.
- **pts_semantic_mask** (*list[torch.Tensor]*) – Point-wise semantic label of each batch.
- **pts_instance_mask** (*list[torch.Tensor]*) – Point-wise instance label of each batch.
- **bbox_preds** (*torch.Tensor*) – Bounding box predictions of vote head.

Returns Targets of vote head.

Return type tuple[torch.Tensor]

get_targets_single(*points*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*, *aggregated_points=None*)
Generate targets of vote head for single batch.

Parameters

- **points** (*torch.Tensor*) – Points of each batch.
- **gt_bboxes_3d** (BaseInstance3DBoxes) – Ground truth boxes of each batch.
- **gt_labels_3d** (*torch.Tensor*) – Labels of each batch.
- **pts_semantic_mask** (*torch.Tensor*) – Point-wise semantic label of each batch.
- **pts_instance_mask** (*torch.Tensor*) – Point-wise instance label of each batch.
- **aggregated_points** (*torch.Tensor*) – Aggregated points from vote aggregation layer.

Returns Targets of vote head.

Return type tuple[torch.Tensor]

loss(*bbox_preds*, *points*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*, *img_metas=None*, *gt_bboxes_ignore=None*, *ret_target=False*)
Compute loss.

Parameters

- **bbox_preds** (*dict*) – Predictions from forward of vote head.
- **points** (*list[torch.Tensor]*) – Input points.
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth bboxes of each sample.
- **gt_labels_3d** (*list[torch.Tensor]*) – Labels of each sample.
- **pts_semantic_mask** (*list[torch.Tensor]*) – Point-wise semantic mask.
- **pts_instance_mask** (*list[torch.Tensor]*) – Point-wise instance mask.
- **img_metas** (*list[dict]*) – Contain pcd and img's meta info.

- **gt_bboxes_ignore** (*list[torch.Tensor]*) – Specify which bounding.
- **ret_target** (*Bool*) – Return targets or not.

Returns Losses of Votenet.

Return type dict

multiclass_nms_single(*obj_scores*, *sem_scores*, *bbox*, *points*, *input_meta*)
Multi-class nms in single batch.

Parameters

- **obj_scores** (*torch.Tensor*) – Objectness score of bounding boxes.
- **sem_scores** (*torch.Tensor*) – semantic class score of bounding boxes.
- **bbox** (*torch.Tensor*) – Predicted bounding boxes.
- **points** (*torch.Tensor*) – Input points.
- **input_meta** (*dict*) – Point cloud and image's meta info.

Returns Bounding boxes, scores and labels.

Return type tuple[torch.Tensor]

47.5 roi_heads

class mmdet3d.models.roi_heads.**Base3DRoIHead**(*bbox_head=None*, *mask_roi_extractor=None*, *mask_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)
Base class for 3d RoIHeads.

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

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

abstract forward_train(*x*, *img_metas*, *proposal_list*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore=None*, ***kwargs*)
Forward function during training.

Parameters

- **x** (*dict*) – Contains features from the first stage.
- **img_metas** (*list[dict]*) – Meta info of each image.
- **proposal_list** (*list[dict]*) – Proposal information from rpn.
- **gt_bboxes** (list[BaseInstance3DBoxes]) – GT bboxes of each sample. The bboxes are encapsulated by 3D box structures.
- **gt_labels** (*list[torch.LongTensor]*) – GT labels of each sample.
- **gt_bboxes_ignore** (*list[torch.Tensor], optional*) – Ground truth boxes to be ignored.

Returns Losses from each head.

Return type dict[str, torch.Tensor]

abstract init_assigner_sampler()
Initialize assigner and sampler.

abstract init_bbox_head()
Initialize the box head.

abstract init_mask_head()
Initialize maek head.

simple_test(*x*, *proposal_list*, *img_metas*, *proposals=None*, *rescale=False*, ***kwargs*)
Test without augmentation.

property with_bbox
whether the RoIHead has box head

Type bool

property with_mask
whether the RoIHead has mask head

Type bool

class mmdet3d.models.roi_heads.**H3DBboxHead**(*num_classes*, *suface_matching_cfg*, *line_matching_cfg*, *bbox_coder*, *train_cfg=None*, *test_cfg=None*, *gt_per_seed=1*, *num_proposal=256*, *feat_channels=(128, 128)*, *primitive_feat_refine_streams=2*, *primitive_refine_channels=[128, 128, 128]*, *upper_thresh=100.0*, *surface_thresh=0.5*, *line_thresh=0.5*, *conv_cfg={'type': 'Conv1d'}*, *norm_cfg={'type': 'BN1d'}*, *objectness_loss=None*, *center_loss=None*, *dir_class_loss=None*, *dir_res_loss=None*, *size_class_loss=None*, *size_res_loss=None*, *semantic_loss=None*, *cues_objectness_loss=None*, *cues_semantic_loss=None*, *proposal_objectness_loss=None*, *primitive_center_loss=None*, *init_cfg=None*)

Bbox head of H3DNet.

Parameters

- **num_classes** (*int*) – The number of classes.
- **surface_matching_cfg** (*dict*) – Config for surface primitive matching.
- **line_matching_cfg** (*dict*) – Config for line primitive matching.
- **bbox_coder** (BaseBBoxCoder) – Bbox coder for encoding and decoding boxes.
- **train_cfg** (*dict*) – Config for training.
- **test_cfg** (*dict*) – Config for testing.
- **gt_per_seed** (*int*) – Number of ground truth votes generated from each seed point.
- **num_proposal** (*int*) – Number of proposal votes generated.
- **feat_channels** (*tuple[int]*) – Convolution channels of prediction layer.
- **primitive_feat_refine_streams** (*int*) – The number of mlps to refine primitive feature.
- **primitive_refine_channels** (*tuple[int]*) – Convolution channels of prediction layer.
- **upper_thresh** (*float*) – Threshold for line matching.
- **surface_thresh** (*float*) – Threshold for surface matching.
- **line_thresh** (*float*) – Threshold for line matching.

- **conv_cfg** (*dict*) – Config of convolution in prediction layer.
- **norm_cfg** (*dict*) – Config of BN in prediction layer.
- **objectness_loss** (*dict*) – Config of objectness loss.
- **center_loss** (*dict*) – Config of center loss.
- **dir_class_loss** (*dict*) – Config of direction classification loss.
- **dir_res_loss** (*dict*) – Config of direction residual regression loss.
- **size_class_loss** (*dict*) – Config of size classification loss.
- **size_res_loss** (*dict*) – Config of size residual regression loss.
- **semantic_loss** (*dict*) – Config of point-wise semantic segmentation loss.
- **cues_objectness_loss** (*dict*) – Config of cues objectness loss.
- **cues_semantic_loss** (*dict*) – Config of cues semantic loss.
- **proposal_objectness_loss** (*dict*) – Config of proposal objectness loss.
- **primitive_center_loss** (*dict*) – Config of primitive center regression loss.

forward(*feats_dict*, *sample_mod*)

Forward pass.

Parameters

- **feats_dict** (*dict*) – Feature dict from backbone.
- **sample_mod** (*str*) – Sample mode for vote aggregation layer. valid modes are "vote", "seed" and "random".

Returns Predictions of vote head.

Return type dict

get_bboxes(*points*, *bbox_preds*, *input_metas*, *rescale=False*, *suffix=''*)

Generate bboxes from vote head predictions.

Parameters

- **points** (*torch.Tensor*) – Input points.
- **bbox_preds** (*dict*) – Predictions from vote head.
- **input_metas** (*list[dict]*) – Point cloud and image's meta info.
- **rescale** (*bool*) – Whether to rescale bboxes.

Returns Bounding boxes, scores and labels.

Return type list[tuple[torch.Tensor]]

get_proposal_stage_loss(*bbox_preds*, *size_class_targets*, *size_res_targets*, *dir_class_targets*, *dir_res_targets*, *center_targets*, *mask_targets*, *objectness_targets*, *objectness_weights*, *box_loss_weights*, *valid_gt_weights*, *suffix=''*)

Compute loss for the aggregation module.

Parameters

- **bbox_preds** (*dict*) – Predictions from forward of vote head.
- **size_class_targets** (*torch.Tensor*) – Ground truth size class of each prediction bounding box.

- **size_res_targets** (*torch.Tensor*) – Ground truth size residual of each prediction bounding box.
- **dir_class_targets** (*torch.Tensor*) – Ground truth direction class of each prediction bounding box.
- **dir_res_targets** (*torch.Tensor*) – Ground truth direction residual of each prediction bounding box.
- **center_targets** (*torch.Tensor*) – Ground truth center of each prediction bounding box.
- **mask_targets** (*torch.Tensor*) – Validation of each prediction bounding box.
- **objectness_targets** (*torch.Tensor*) – Ground truth objectness label of each prediction bounding box.
- **objectness_weights** (*torch.Tensor*) – Weights of objectness loss for each prediction bounding box.
- **box_loss_weights** (*torch.Tensor*) – Weights of regression loss for each prediction bounding box.
- **valid_gt_weights** (*torch.Tensor*) – Validation of each ground truth bounding box.

Returns Losses of aggregation module.

Return type dict

get_targets(*points*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*, *bbox_preds=None*)

Generate targets of proposal module.

Parameters

- **points** (*list[torch.Tensor]*) – Points of each batch.
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth bboxes of each batch.
- **gt_labels_3d** (*list[torch.Tensor]*) – Labels of each batch.
- **pts_semantic_mask** (*list[torch.Tensor]*) – Point-wise semantic label of each batch.
- **pts_instance_mask** (*list[torch.Tensor]*) – Point-wise instance label of each batch.
- **bbox_preds** (*torch.Tensor*) – Bounding box predictions of vote head.

Returns Targets of proposal module.

Return type tuple[torch.Tensor]

get_targets_single(*points*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*, *aggregated_points=None*, *pred_surface_center=None*, *pred_line_center=None*, *pred_obj_surface_center=None*, *pred_obj_line_center=None*, *pred_surface_sem=None*, *pred_line_sem=None*)

Generate targets for primitive cues for single batch.

Parameters

- **points** (*torch.Tensor*) – Points of each batch.
- **gt_bboxes_3d** (BaseInstance3DBoxes) – Ground truth boxes of each batch.

- **gt_labels_3d** (*torch.Tensor*) – Labels of each batch.
- **pts_semantic_mask** (*torch.Tensor*) – Point-wise semantic label of each batch.
- **pts_instance_mask** (*torch.Tensor*) – Point-wise instance label of each batch.
- **aggregated_points** (*torch.Tensor*) – Aggregated points from vote aggregation layer.
- **pred_surface_center** (*torch.Tensor*) – Prediction of surface center.
- **pred_line_center** (*torch.Tensor*) – Prediction of line center.
- **pred_obj_surface_center** (*torch.Tensor*) – Objectness prediction of surface center.
- **pred_obj_line_center** (*torch.Tensor*) – Objectness prediction of line center.
- **pred_surface_sem** (*torch.Tensor*) – Semantic prediction of surface center.
- **pred_line_sem** (*torch.Tensor*) – Semantic prediction of line center.

Returns Targets for primitive cues.

Return type tuple[torch.Tensor]

loss(*bbox_preds*, *points*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*, *img_metas=None*, *rpn_targets=None*, *gt_bboxes_ignore=None*)
Compute loss.

Parameters

- **bbox_preds** (*dict*) – Predictions from forward of h3d bbox head.
- **points** (*list[torch.Tensor]*) – Input points.
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth bboxes of each sample.
- **gt_labels_3d** (*list[torch.Tensor]*) – Labels of each sample.
- **pts_semantic_mask** (*list[torch.Tensor]*) – Point-wise semantic mask.
- **pts_instance_mask** (*list[torch.Tensor]*) – Point-wise instance mask.
- **img_metas** (*list[dict]*) – Contain pcd and img's meta info.
- **rpn_targets** (*Tuple*) – Targets generated by rpn head.
- **gt_bboxes_ignore** (*list[torch.Tensor]*) – Specify which bounding.

Returns Losses of H3dnet.

Return type dict

multiclass_nms_single(*obj_scores*, *sem_scores*, *bbox*, *points*, *input_meta*)
Multi-class nms in single batch.

Parameters

- **obj_scores** (*torch.Tensor*) – Objectness score of bounding boxes.
- **sem_scores** (*torch.Tensor*) – semantic class score of bounding boxes.
- **bbox** (*torch.Tensor*) – Predicted bounding boxes.
- **points** (*torch.Tensor*) – Input points.
- **input_meta** (*dict*) – Point cloud and image's meta info.

Returns Bounding boxes, scores and labels.

Return type tuple[torch.Tensor]

class mmdet3d.models.roi_heads.**H3DRoIHead**(*primitive_list*, *bbox_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)

H3D roi head for H3DNet.

Parameters

- **primitive_list** (*List*) – Configs of primitive heads.
- **bbox_head** (*ConfigDict*) – Config of bbox_head.
- **train_cfg** (*ConfigDict*) – Training config.
- **test_cfg** (*ConfigDict*) – Testing config.

forward_train(*feats_dict*, *img_metas*, *points*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask*, *pts_instance_mask*, *gt_bboxes_ignore=None*)

Training forward function of PartAggregationROIHead.

Parameters

- **feats_dict** (*dict*) – Contains features from the first stage.
- **img_metas** (*list[dict]*) – Contain pcd and img's meta info.
- **points** (*list[torch.Tensor]*) – Input points.
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth bboxes of each sample.
- **gt_labels_3d** (*list[torch.Tensor]*) – Labels of each sample.
- **pts_semantic_mask** (*list[torch.Tensor]*) – Point-wise semantic mask.
- **pts_instance_mask** (*list[torch.Tensor]*) – Point-wise instance mask.
- **gt_bboxes_ignore** (*list[torch.Tensor]*) – Specify which bounding boxes to ignore.

Returns losses from each head.

Return type dict

init_assigner_sampler()

Initialize assigner and sampler.

init_bbox_head(*bbox_head*)

Initialize box head.

init_mask_head()

Initialize mask head, skip since H3DROIHead does not have one.

simple_test(*feats_dict*, *img_metas*, *points*, *rescale=False*)

Simple testing forward function of PartAggregationROIHead.

Note: This function assumes that the batch size is 1

Parameters

- **feats_dict** (*dict*) – Contains features from the first stage.
- **img_metas** (*list[dict]*) – Contain pcd and img's meta info.

- **points** (*torch.Tensor*) – Input points.
- **rescale** (*bool*) – Whether to rescale results.

Returns Bbox results of one frame.

Return type dict

class mmdet3d.models.roi_heads.**PartA2BboxHead**(*num_classes*, *seg_in_channels*, *part_in_channels*, *seg_conv_channels=None*, *part_conv_channels=None*, *merge_conv_channels=None*, *down_conv_channels=None*, *shared_fc_channels=None*, *cls_channels=None*, *reg_channels=None*, *dropout_ratio=0.1*, *roi_feat_size=14*, *with_corner_loss=True*, *bbox_coder={'type': 'DeltaXYZWLHRBBoxCoder'}*, *conv_cfg={'type': 'Conv1d'}*, *norm_cfg={'eps': 0.001, 'momentum': 0.01, 'type': 'BN1d'}*, *loss_bbox={'beta': 0.1111111111111111, 'loss_weight': 2.0, 'type': 'SmoothL1Loss'}*, *loss_cls={'loss_weight': 1.0, 'reduction': 'none', 'type': 'CrossEntropyLoss', 'use_sigmoid': True}*, *init_cfg=None*)

PartA2 RoI head.

Parameters

- **num_classes** (*int*) – The number of classes to prediction.
- **seg_in_channels** (*int*) – Input channels of segmentation convolution layer.
- **part_in_channels** (*int*) – Input channels of part convolution layer.
- **seg_conv_channels** (*list(int)*) – Out channels of each segmentation convolution layer.
- **part_conv_channels** (*list(int)*) – Out channels of each part convolution layer.
- **merge_conv_channels** (*list(int)*) – Out channels of each feature merged convolution layer.
- **down_conv_channels** (*list(int)*) – Out channels of each downsampled convolution layer.
- **shared_fc_channels** (*list(int)*) – Out channels of each shared fc layer.
- **cls_channels** (*list(int)*) – Out channels of each classification layer.
- **reg_channels** (*list(int)*) – Out channels of each regression layer.
- **dropout_ratio** (*float*) – Dropout ratio of classification and regression layers.
- **roi_feat_size** (*int*) – The size of pooled roi features.
- **with_corner_loss** (*bool*) – Whether to use corner loss or not.
- **bbox_coder** (BaseBBoxCoder) – Bbox coder for box head.
- **conv_cfg** (*dict*) – Config dict of convolutional layers
- **norm_cfg** (*dict*) – Config dict of normalization layers
- **loss_bbox** (*dict*) – Config dict of box regression loss.
- **loss_cls** (*dict*) – Config dict of classifacation loss.

forward(*seg_feats*, *part_feats*)
Forward pass.

Parameters

- **seg_feats** (*torch.Tensor*) – Point-wise semantic features.
- **part_feats** (*torch.Tensor*) – Point-wise part prediction features.

Returns Score of class and bbox predictions.

Return type tuple[torch.Tensor]

get_bboxes(*rois*, *cls_score*, *bbox_pred*, *class_labels*, *class_pred*, *img_metas*, *cfg=None*)
Generate bboxes from bbox head predictions.

Parameters

- **rois** (*torch.Tensor*) – Roi bounding boxes.
- **cls_score** (*torch.Tensor*) – Scores of bounding boxes.
- **bbox_pred** (*torch.Tensor*) – Bounding boxes predictions
- **class_labels** (*torch.Tensor*) – Label of classes
- **class_pred** (*torch.Tensor*) – Score for nms.
- **img_metas** (*list[dict]*) – Point cloud and image's meta info.
- **cfg** (ConfigDict) – Testing config.

Returns Decoded bbox, scores and labels after nms.

Return type list[tuple]

get_corner_loss_lidar(*pred_bbox3d*, *gt_bbox3d*, *delta=1.0*)
Calculate corner loss of given boxes.

Parameters

- **pred_bbox3d** (*torch.FloatTensor*) – Predicted boxes in shape (N, 7).
- **gt_bbox3d** (*torch.FloatTensor*) – Ground truth boxes in shape (N, 7).
- **delta** (*float, optional*) – huber loss threshold. Defaults to 1.0

Returns Calculated corner loss in shape (N).

Return type torch.FloatTensor

get_targets(*sampling_results*, *rcnn_train_cfg*, *concat=True*)
Generate targets.

Parameters

- **sampling_results** (list[SamplingResult]) – Sampled results from rois.
- **rcnn_train_cfg** (ConfigDict) – Training config of rcnn.
- **concat** (*bool*) – Whether to concatenate targets between batches.

Returns Targets of boxes and class prediction.

Return type tuple[torch.Tensor]

init_weights()
Initialize the weights.

loss(*cls_score*, *bbox_pred*, *rois*, *labels*, *bbox_targets*, *pos_gt_bboxes*, *reg_mask*, *label_weights*, *bbox_weights*)
Computing losses.

Parameters

- **cls_score** (*torch.Tensor*) – Scores of each roi.
- **bbox_pred** (*torch.Tensor*) – Predictions of bboxes.
- **rois** (*torch.Tensor*) – Roi bboxes.
- **labels** (*torch.Tensor*) – Labels of class.
- **bbox_targets** (*torch.Tensor*) – Target of positive bboxes.
- **pos_gt_bboxes** (*torch.Tensor*) – Ground truths of positive bboxes.
- **reg_mask** (*torch.Tensor*) – Mask for positive bboxes.
- **label_weights** (*torch.Tensor*) – Weights of class loss.
- **bbox_weights** (*torch.Tensor*) – Weights of bbox loss.

Returns

Computed losses.

- loss_cls (torch.Tensor): Loss of classes.
- loss_bbox (torch.Tensor): Loss of bboxes.
- loss_corner (torch.Tensor): Loss of corners.

Return type dict

multi_class_nms(*box_probs*, *box_preds*, *score_thr*, *nms_thr*, *input_meta*, *use_rotate_nms=True*)
Multi-class NMS for box head.

Note: This function has large overlap with the *box3d_multiclass_nms* implemented in *mmdet3d.core.post_processing*. We are considering merging these two functions in the future.

Parameters

- **box_probs** (*torch.Tensor*) – Predicted boxes probabitilies in shape (N,).
- **box_preds** (*torch.Tensor*) – Predicted boxes in shape (N, 7+C).
- **score_thr** (*float*) – Threshold of scores.
- **nms_thr** (*float*) – Threshold for NMS.
- **input_meta** (*dict*) – Meta information of the current sample.
- **use_rotate_nms** (*bool, optional*) – Whether to use rotated nms. Defaults to True.

Returns Selected indices.

Return type torch.Tensor

class mmdet3d.models.roi_heads.**PartAggregationROIHead**(*semantic_head*, *num_classes=3*, *seg_roi_extractor=None*, *part_roi_extractor=None*, *bbox_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)

Part aggregation roi head for PartA2.

Parameters

- **semantic_head** (*ConfigDict*) – Config of semantic head.
- **num_classes** (*int*) – The number of classes.
- **seg_roi_extractor** (*ConfigDict*) – Config of seg_roi_extractor.
- **part_roi_extractor** (*ConfigDict*) – Config of part_roi_extractor.
- **bbox_head** (*ConfigDict*) – Config of bbox_head.
- **train_cfg** (*ConfigDict*) – Training config.
- **test_cfg** (*ConfigDict*) – Testing config.

forward_train(*feats_dict*, *voxels_dict*, *img_metas*, *proposal_list*, *gt_bboxes_3d*, *gt_labels_3d*)

Training forward function of PartAggregationROIHead.

Parameters

- **feats_dict** (*dict*) – Contains features from the first stage.
- **voxels_dict** (*dict*) – Contains information of voxels.
- **img_metas** (*list[dict]*) – Meta info of each image.
- **proposal_list** (*list[dict]*) – Proposal information from rpn. The dictionary should contain the following keys:
  - boxes_3d (BaseInstance3DBoxes): Proposal bboxes
  - labels_3d (torch.Tensor): Labels of proposals
  - cls_preds (torch.Tensor): Original scores of proposals
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – GT bboxes of each sample. The bboxes are encapsulated by 3D box structures.
- **gt_labels_3d** (*list[LongTensor]*) – GT labels of each sample.

Returns

losses from each head.

- loss_semantic (torch.Tensor): loss of semantic head
- loss_bbox (torch.Tensor): loss of bboxes

Return type dict

init_assigner_sampler()

Initialize assigner and sampler.

init_bbox_head(*bbox_head*)

Initialize box head.

init_mask_head()

Initialize mask head, skip since PartAggregationROIHead does not have one.

simple_test(*feats_dict*, *voxels_dict*, *img_metas*, *proposal_list*, ***kwargs*)
Simple testing forward function of PartAggregationROIHead.

Note: This function assumes that the batch size is 1

Parameters

- **feats_dict** (*dict*) – Contains features from the first stage.
- **voxels_dict** (*dict*) – Contains information of voxels.
- **img_metas** (*list[dict]*) – Meta info of each image.
- **proposal_list** (*list[dict]*) – Proposal information from rpn.

Returns Bbox results of one frame.

Return type dict

property with_semantic
whether the head has semantic branch

Type bool

class mmdet3d.models.roi_heads.**PointRCNNBboxHead**(*num_classes*, *in_channels*, *mlp_channels*, *pred_layer_cfg=None*, *num_points=(128, 32, - 1)*, *radius=(0.2, 0.4, 100)*, *num_samples=(64, 64, 64)*, *sa_channels=((128, 128, 128), (128, 128, 256), (256, 256, 512))*, *bbox_coder={'type': 'DeltaXYZWLHRBBoxCoder'}*, *sa_cfg={'pool_mod': 'max', 'type': 'PointSAModule', 'use_xyz': True}*, *conv_cfg={'type': 'Conv1d'}*, *norm_cfg={'type': 'BN1d'}*, *act_cfg={'type': 'ReLU'}*, *bias='auto'*, *loss_bbox={'beta': 0.1111111111111111, 'loss_weight': 1.0, 'reduction': 'sum', 'type': 'SmoothL1Loss'}*, *loss_cls={'loss_weight': 1.0, 'reduction': 'sum', 'type': 'CrossEntropyLoss', 'use_sigmoid': True}*, *with_corner_loss=True*, *init_cfg=None*)

PointRCNN RoI Bbox head.

Parameters

- **num_classes** (*int*) – The number of classes to prediction.
- **in_channels** (*int*) –
- **mlp_channels** (*list[int]*) – the number of mlp channels
- **pred_layer_cfg** (*dict, optional*) – Config of classfication and regression prediction layers. Defaults to None.
- **num_points** (*tuple, optional*) – The number of points which each SA module samples. Defaults to (128, 32, -1).
- **radius** (*tuple, optional*) – Sampling radius of each SA module. Defaults to (0.2, 0.4, 100).
- **num_samples** (*tuple, optional*) – The number of samples for ball query in each SA module. Defaults to (64, 64, 64).

- **sa_channels** (*tuple, optional*) – Out channels of each mlp in SA module. Defaults to ((128, 128, 128), (128, 128, 256), (256, 256, 512)).
- **bbox_coder** (*dict, optional*) – Config dict of box coders. Defaults to dict(type='DeltaXYZWLHRBBoxCoder').
- **sa_cfg** (*dict, optional*) – Config of set abstraction module, which may contain the following keys and values:
  - pool_mod (str): Pool method ('max' or 'avg') for SA modules.
  - use_xyz (bool): Whether to use xyz as a part of features.
  - normalize_xyz (bool): Whether to normalize xyz with radii in each SA module.

  Defaults to dict(type='PointSAModule', pool_mod='max', use_xyz=True).
- **conv_cfg** (*dict, optional*) – Config dict of convolutional layers. Defaults to dict(type='Conv1d').
- **norm_cfg** (*dict, optional*) – Config dict of normalization layers. Defaults to dict(type='BN1d').
- **act_cfg** (*dict, optional*) – Config dict of activation layers. Defaults to dict(type='ReLU').
- **bias** (*str, optional*) – Type of bias. Defaults to 'auto'.
- **loss_bbox** (*dict, optional*) – Config of regression loss function. Defaults to dict(type='SmoothL1Loss', beta=1.0 / 9.0,

  reduction='sum', loss_weight=1.0).
- **loss_cls** (*dict, optional*) – Config of classification loss function. Defaults to dict(type='CrossEntropyLoss', use_sigmoid=True,

  reduction='sum', loss_weight=1.0).
- **with_corner_loss** (*bool, optional*) – Whether using corner loss. Defaults to True.
- **init_cfg** (*dict, optional*) – Config of initialization. Defaults to None.

forward(*feats*)

Forward pass.

Parameters **feats** (*torch.Torch*) – Features from RCNN modules.

Returns Score of class and bbox predictions.

Return type tuple[torch.Tensor]

get_bboxes(*rois*, *cls_score*, *bbox_pred*, *class_labels*, *img_metas*, *cfg=None*)

Generate bboxes from bbox head predictions.

Parameters

- **rois** (*torch.Tensor*) – RoI bounding boxes.
- **cls_score** (*torch.Tensor*) – Scores of bounding boxes.
- **bbox_pred** (*torch.Tensor*) – Bounding boxes predictions
- **class_labels** (*torch.Tensor*) – Label of classes
- **img_metas** (*list[dict]*) – Point cloud and image's meta info.
- **cfg** (ConfigDict, optional) – Testing config. Defaults to None.

Returns Decoded bbox, scores and labels after nms.

Return type list[tuple]

get_corner_loss_lidar(*pred_bbox3d*, *gt_bbox3d*, *delta=1.0*)
Calculate corner loss of given boxes.

Parameters

- **pred_bbox3d** (*torch.FloatTensor*) – Predicted boxes in shape (N, 7).
- **gt_bbox3d** (*torch.FloatTensor*) – Ground truth boxes in shape (N, 7).
- **delta** (*float, optional*) – huber loss threshold. Defaults to 1.0

Returns Calculated corner loss in shape (N).

Return type torch.FloatTensor

get_targets(*sampling_results*, *rcnn_train_cfg*, *concat=True*)
Generate targets.

Parameters

- **sampling_results** (list[SamplingResult]) – Sampled results from rois.
- **rcnn_train_cfg** (ConfigDict) – Training config of rcnn.
- **concat** (*bool, optional*) – Whether to concatenate targets between batches. Defaults to True.

Returns Targets of boxes and class prediction.

Return type tuple[torch.Tensor]

init_weights()
Initialize weights of the head.

loss(*cls_score*, *bbox_pred*, *rois*, *labels*, *bbox_targets*, *pos_gt_bboxes*, *reg_mask*, *label_weights*, *bbox_weights*)
Computing losses.

Parameters

- **cls_score** (*torch.Tensor*) – Scores of each RoI.
- **bbox_pred** (*torch.Tensor*) – Predictions of bboxes.
- **rois** (*torch.Tensor*) – RoI bboxes.
- **labels** (*torch.Tensor*) – Labels of class.
- **bbox_targets** (*torch.Tensor*) – Target of positive bboxes.
- **pos_gt_bboxes** (*torch.Tensor*) – Ground truths of positive bboxes.
- **reg_mask** (*torch.Tensor*) – Mask for positive bboxes.
- **label_weights** (*torch.Tensor*) – Weights of class loss.
- **bbox_weights** (*torch.Tensor*) – Weights of bbox loss.

Returns

Computed losses.

- loss_cls (torch.Tensor): Loss of classes.
- loss_bbox (torch.Tensor): Loss of bboxes.

- loss_corner (torch.Tensor): Loss of corners.

Return type dict

multi_class_nms(*box_probs*, *box_preds*, *score_thr*, *nms_thr*, *input_meta*, *use_rotate_nms=True*)
Multi-class NMS for box head.

Note: This function has large overlap with the *box3d_multiclass_nms* implemented in *mmdet3d.core.post_processing*. We are considering merging these two functions in the future.

Parameters

- **box_probs** (*torch.Tensor*) – Predicted boxes probabilities in shape (N,).
- **box_preds** (*torch.Tensor*) – Predicted boxes in shape (N, 7+C).
- **score_thr** (*float*) – Threshold of scores.
- **nms_thr** (*float*) – Threshold for NMS.
- **input_meta** (*dict*) – Meta information of the current sample.
- **use_rotate_nms** (*bool, optional*) – Whether to use rotated nms. Defaults to True.

Returns Selected indices.

Return type torch.Tensor

class mmdet3d.models.roi_heads.**PointRCNNRoIHead**(*bbox_head*, *point_roi_extractor*, *train_cfg*, *test_cfg*, *depth_normalizer=70.0*, *pretrained=None*, *init_cfg=None*)
RoI head for PointRCNN.

Parameters

- **bbox_head** (*dict*) – Config of bbox_head.
- **point_roi_extractor** (*dict*) – Config of RoI extractor.
- **train_cfg** (*dict*) – Train configs.
- **test_cfg** (*dict*) – Test configs.
- **depth_normalizer** (*float, optional*) – Normalize depth feature. Defaults to 70.0.
- **init_cfg** (*dict, optional*) – Config of initialization. Defaults to None.

forward_train(*feats_dict*, *input_metas*, *proposal_list*, *gt_bboxes_3d*, *gt_labels_3d*)
Training forward function of PointRCNNRoIHead.

Parameters

- **feats_dict** (*dict*) – Contains features from the first stage.
- **imput_metas** (*list[dict]*) – Meta info of each input.
- **proposal_list** (*list[dict]*) – Proposal information from rpn. The dictionary should contain the following keys:
  - boxes_3d (BaseInstance3DBoxes): Proposal bboxes
  - labels_3d (torch.Tensor): Labels of proposals

- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – GT bboxes of each sample. The bboxes are encapsulated by 3D box structures.
- **gt_labels_3d** (*list[LongTensor]*) – GT labels of each sample.

Returns

Losses from RoI RCNN head.

- loss_bbox (torch.Tensor): Loss of bboxes

Return type dict

init_assigner_sampler()
Initialize assigner and sampler.

init_bbox_head(*bbox_head*)
Initialize box head.

Parameters **bbox_head** (*dict*) – Config dict of RoI Head.

init_mask_head()
Initialize maek head.

simple_test(*feats_dict*, *img_metas*, *proposal_list*, ***kwargs*)
Simple testing forward function of PointRCNNRoIHead.

Note: This function assumes that the batch size is 1

Parameters

- **feats_dict** (*dict*) – Contains features from the first stage.
- **img_metas** (*list[dict]*) – Meta info of each image.
- **proposal_list** (*list[dict]*) – Proposal information from rpn.

Returns Bbox results of one frame.

Return type dict

class mmdet3d.models.roi_heads.**PointwiseSemanticHead**(*in_channels*, *num_classes=3*, *extra_width=0.2*, *seg_score_thr=0.3*, *init_cfg=None*, *loss_seg={'alpha': 0.25, 'gamma': 2.0, 'loss_weight': 1.0, 'reduction': 'sum', 'type': 'FocalLoss', 'use_sigmoid': True}*, *loss_part={'loss_weight': 1.0, 'type': 'CrossEntropyLoss', 'use_sigmoid': True}*)
Semantic segmentation head for point-wise segmentation.

Predict point-wise segmentation and part regression results for PartA2. See paper for more details.

Parameters

- **in_channels** (*int*) – The number of input channel.
- **num_classes** (*int*) – The number of class.
- **extra_width** (*float*) – Boxes enlarge width.
- **loss_seg** (*dict*) – Config of segmentation loss.
- **loss_part** (*dict*) – Config of part prediction loss.

forward(*x*)
Forward pass.

Parameters **x** (*torch.Tensor*) – Features from the first stage.

Returns

Part features, segmentation and part predictions.

- seg_preds (torch.Tensor): Segment predictions.
- part_preds (torch.Tensor): Part predictions.
- part_feats (torch.Tensor): Feature predictions.

Return type dict

get_targets(*voxels_dict*, *gt_bboxes_3d*, *gt_labels_3d*)
generate segmentation and part prediction targets.

Parameters

- **voxel_centers** (*torch.Tensor*) – The center of voxels in shape (voxel_num, 3).
- **gt_bboxes_3d** (BaseInstance3DBoxes) – Ground truth boxes in shape (box_num, 7).
- **gt_labels_3d** (*torch.Tensor*) – Class labels of ground truths in shape (box_num).

Returns

Prediction targets

- **seg_targets (torch.Tensor): Segmentation targets** with shape [voxel_num].
- **part_targets (torch.Tensor): Part prediction targets** with shape [voxel_num, 3].

Return type dict

get_targets_single(*voxel_centers*, *gt_bboxes_3d*, *gt_labels_3d*)
generate segmentation and part prediction targets for a single sample.

Parameters

- **voxel_centers** (*torch.Tensor*) – The center of voxels in shape (voxel_num, 3).
- **gt_bboxes_3d** (BaseInstance3DBoxes) – Ground truth boxes in shape (box_num, 7).
- **gt_labels_3d** (*torch.Tensor*) – Class labels of ground truths in shape (box_num).

Returns

Segmentation targets with shape [voxel_num] part prediction targets with shape [voxel_num, 3]

Return type tuple[torch.Tensor]

loss(*semantic_results*, *semantic_targets*)
Calculate point-wise segmentation and part prediction losses.

Parameters

- **semantic_results** (*dict*) – Results from semantic head.
  - seg_preds: Segmentation predictions.
  - part_preds: Part predictions.

- **semantic_targets** (*dict*) – Targets of semantic results.
  - seg_preds: Segmentation targets.
  - part_preds: Part targets.

Returns

Loss of segmentation and part prediction.

- loss_seg (torch.Tensor): Segmentation prediction loss.
- loss_part (torch.Tensor): Part prediction loss.

Return type dict

class mmdet3d.models.roi_heads.**PrimitiveHead**(*num_dims*, *num_classes*, *primitive_mode*, *train_cfg=None*, *test_cfg=None*, *vote_module_cfg=None*, *vote_aggregation_cfg=None*, *feat_channels=(128, 128)*, *upper_thresh=100.0*, *surface_thresh=0.5*, *conv_cfg={'type': 'Conv1d'}*, *norm_cfg={'type': 'BN1d'}*, *objectness_loss=None*, *center_loss=None*, *semantic_reg_loss=None*, *semantic_cls_loss=None*, *init_cfg=None*)

Primitive head of H3DNet.

Parameters

- **num_dims** (*int*) – The dimension of primitive semantic information.
- **num_classes** (*int*) – The number of class.
- **primitive_mode** (*str*) – The mode of primitive module, available mode ['z', 'xy', 'line'].
- **bbox_coder** (BaseBBoxCoder) – Bbox coder for encoding and decoding boxes.
- **train_cfg** (*dict*) – Config for training.
- **test_cfg** (*dict*) – Config for testing.
- **vote_module_cfg** (*dict*) – Config of VoteModule for point-wise votes.
- **vote_aggregation_cfg** (*dict*) – Config of vote aggregation layer.
- **feat_channels** (*tuple[int]*) – Convolution channels of prediction layer.
- **upper_thresh** (*float*) – Threshold for line matching.
- **surface_thresh** (*float*) – Threshold for surface matching.
- **conv_cfg** (*dict*) – Config of convolution in prediction layer.
- **norm_cfg** (*dict*) – Config of BN in prediction layer.
- **objectness_loss** (*dict*) – Config of objectness loss.
- **center_loss** (*dict*) – Config of center loss.
- **semantic_loss** (*dict*) – Config of point-wise semantic segmentation loss.

check_dist(*plane_equ*, *points*)

Whether the mean of points to plane distance is lower than thresh.

Parameters

- **plane_equ** (*torch.Tensor*) – Plane to be checked.

- **points** (*torch.Tensor*) – Points to be checked.

Returns Flag of result.

Return type Tuple

check_horizon(*points*)
Check whether is a horizontal plane.

Parameters **points** (*torch.Tensor*) – Points of input.

Returns Flag of result.

Return type Bool

compute_primitive_loss(*primitive_center*, *primitive_semantic*, *semantic_scores*, *num_proposal*, *gt_primitive_center*, *gt_primitive_semantic*, *gt_sem_cls_label*, *gt_primitive_mask*)
Compute loss of primitive module.

Parameters

- **primitive_center** (*torch.Tensor*) – Pridictions of primitive center.
- **primitive_semantic** (*torch.Tensor*) – Pridictions of primitive semantic.
- **semantic_scores** (*torch.Tensor*) – Pridictions of primitive semantic scores.
- **num_proposal** (*int*) – The number of primitive proposal.
- **gt_primitive_center** (*torch.Tensor*) – Ground truth of primitive center.
- **gt_votes_sem** (*torch.Tensor*) – Ground truth of primitive semantic.
- **gt_sem_cls_label** (*torch.Tensor*) – Ground truth of primitive semantic class.
- **gt_primitive_mask** (*torch.Tensor*) – Ground truth of primitive mask.

Returns Loss of primitive module.

Return type Tuple

forward(*feats_dict*, *sample_mod*)
Forward pass.

Parameters

- **feats_dict** (*dict*) – Feature dict from backbone.
- **sample_mod** (*str*) – Sample mode for vote aggregation layer. valid modes are "vote", "seed" and "random".

Returns Predictions of primitive head.

Return type dict

get_primitive_center(*pred_flag*, *center*)
Generate primitive center from predictions.

Parameters

- **pred_flag** (*torch.Tensor*) – Scores of primitive center.
- **center** (*torch.Tensor*) – Pridictions of primitive center.

Returns Primitive center and the prediction indices.

Return type Tuple

get_targets(*points*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*, *bbox_preds=None*)
Generate targets of primitive head.

Parameters

- **points** (*list[torch.Tensor]*) – Points of each batch.
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth bboxes of each batch.
- **gt_labels_3d** (*list[torch.Tensor]*) – Labels of each batch.
- **pts_semantic_mask** (*list[torch.Tensor]*) – Point-wise semantic label of each batch.
- **pts_instance_mask** (*list[torch.Tensor]*) – Point-wise instance label of each batch.
- **bbox_preds** (*dict*) – Predictions from forward of primitive head.

Returns Targets of primitive head.

Return type tuple[torch.Tensor]

get_targets_single(*points*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*)
Generate targets of primitive head for single batch.

Parameters

- **points** (*torch.Tensor*) – Points of each batch.
- **gt_bboxes_3d** (BaseInstance3DBoxes) – Ground truth boxes of each batch.
- **gt_labels_3d** (*torch.Tensor*) – Labels of each batch.
- **pts_semantic_mask** (*torch.Tensor*) – Point-wise semantic label of each batch.
- **pts_instance_mask** (*torch.Tensor*) – Point-wise instance label of each batch.

Returns Targets of primitive head.

Return type tuple[torch.Tensor]

loss(*bbox_preds*, *points*, *gt_bboxes_3d*, *gt_labels_3d*, *pts_semantic_mask=None*, *pts_instance_mask=None*, *img_metas=None*, *gt_bboxes_ignore=None*)
Compute loss.

Parameters

- **bbox_preds** (*dict*) – Predictions from forward of primitive head.
- **points** (*list[torch.Tensor]*) – Input points.
- **gt_bboxes_3d** (list[BaseInstance3DBoxes]) – Ground truth bboxes of each sample.
- **gt_labels_3d** (*list[torch.Tensor]*) – Labels of each sample.
- **pts_semantic_mask** (*list[torch.Tensor]*) – Point-wise semantic mask.
- **pts_instance_mask** (*list[torch.Tensor]*) – Point-wise instance mask.
- **img_metas** (*list[dict]*) – Contain pcd and img's meta info.
- **gt_bboxes_ignore** (*list[torch.Tensor]*) – Specify which bounding.

Returns Losses of Primitive Head.

Return type dict

match_point2line(*points*, *corners*, *with_yaw*, *mode='bottom'*)
Match points to corresponding line.

Parameters

- **points** (*torch.Tensor*) – Points of input.
- **corners** (*torch.Tensor*) – Eight corners of a bounding box.
- **with_yaw** (*Bool*) – Whether the boundind box is with rotation.
- **mode** (*str, optional*) – Specify which line should be matched, available mode are ('bottom', 'top', 'left', 'right'). Defaults to 'bottom'.

Returns Flag of matching correspondence.

Return type Tuple

match_point2plane(*plane*, *points*)
Match points to plane.

Parameters

- **plane** (*torch.Tensor*) – Equation of the plane.
- **points** (*torch.Tensor*) – Points of input.

Returns

Distance of each point to the plane and flag of matching correspondence.

Return type Tuple

point2line_dist(*points*, *pts_a*, *pts_b*)
Calculate the distance from point to line.

Parameters

- **points** (*torch.Tensor*) – Points of input.
- **pts_a** (*torch.Tensor*) – Point on the specific line.
- **pts_b** (*torch.Tensor*) – Point on the specific line.

Returns Distance between each point to line.

Return type torch.Tensor

primitive_decode_scores(*predictions*, *aggregated_points*)
Decode predicted parts to primitive head.

Parameters

- **predictions** (*torch.Tensor*) – primitive pridictions of each batch.
- **aggregated_points** (*torch.Tensor*) – The aggregated points of vote stage.

Returns

Predictions of primitive head, including center, semantic size and semantic scores.

Return type Dict

class mmdet3d.models.roi_heads.**Single3DRoIAwareExtractor**(*roi_layer=None*, *init_cfg=None*)
Point-wise roi-aware Extractor.

Extract Point-wise roi features.

Parameters **roi_layer** (*dict*) – The config of roi layer.

build_roi_layers(*layer_cfg*)
Build roi layers using *layer_cfg*

forward(*feats*, *coordinate*, *batch_inds*, *rois*)
Extract point-wise roi features.

Parameters

- **feats** (*torch.FloatTensor*) – Point-wise features with shape (batch, npoints, channels) for pooling.
- **coordinate** (*torch.FloatTensor*) – Coordinate of each point.
- **batch_inds** (*torch.LongTensor*) – Indicate the batch of each point.
- **rois** (*torch.FloatTensor*) – Roi boxes with batch indices.

Returns Pooled features

Return type torch.FloatTensor

class mmdet3d.models.roi_heads.**Single3DRoIPointExtractor**(*roi_layer=None*)
Point-wise roi-aware Extractor.

Extract Point-wise roi features.

Parameters **roi_layer** (*dict*) – The config of roi layer.

build_roi_layers(*layer_cfg*)
Build roi layers using *layer_cfg*

forward(*feats*, *coordinate*, *batch_inds*, *rois*)
Extract point-wise roi features.

Parameters

- **feats** (*torch.FloatTensor*) – Point-wise features with shape (batch, npoints, channels) for pooling.
- **coordinate** (*torch.FloatTensor*) – Coordinate of each point.
- **batch_inds** (*torch.LongTensor*) – Indicate the batch of each point.
- **rois** (*torch.FloatTensor*) – Roi boxes with batch indices.

Returns Pooled features

Return type torch.FloatTensor

class mmdet3d.models.roi_heads.**SingleRoIExtractor**(*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

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

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** (*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

47.6 fusion_layers

class mmdet3d.models.fusion_layers.**PointFusion**(*img_channels*, *pts_channels*, *mid_channels*, *out_channels*, *img_levels=3*, *coord_type='LIDAR'*, *conv_cfg=None*, *norm_cfg=None*, *act_cfg=None*, *init_cfg=None*, *activate_out=True*, *fuse_out=False*, *dropout_ratio=0*, *aligned=True*, *align_corners=True*, *padding_mode='zeros'*, *lateral_conv=True*)
Fuse image features from multi-scale features.

Parameters

- **img_channels** (*list[int] | int*) – Channels of image features. It could be a list if the input is multi-scale image features.
- **pts_channels** (*int*) – Channels of point features
- **mid_channels** (*int*) – Channels of middle layers
- **out_channels** (*int*) – Channels of output fused features
- **img_levels** (*int, optional*) – Number of image levels. Defaults to 3.
- **coord_type** (*str*) – 'DEPTH' or 'CAMERA' or 'LIDAR'. Defaults to 'LIDAR'.
- **conv_cfg** (*dict, optional*) – Dict config of conv layers of middle layers. Defaults to None.
- **norm_cfg** (*dict, optional*) – Dict config of norm layers of middle layers. Defaults to None.
- **act_cfg** (*dict, optional*) – Dict config of activatation layers. Defaults to None.
- **activate_out** (*bool, optional*) – Whether to apply relu activation to output features. Defaults to True.

- **fuse_out** (*bool, optional*) – Whether apply conv layer to the fused features. Defaults to False.
- **dropout_ratio** (*int, float, optional*) – Dropout ratio of image features to prevent overfitting. Defaults to 0.
- **aligned** (*bool, optional*) – Whether apply aligned feature fusion. Defaults to True.
- **align_corners** (*bool, optional*) – Whether to align corner when sampling features according to points. Defaults to True.
- **padding_mode** (*str, optional*) – Mode used to pad the features of points that do not have corresponding image features. Defaults to 'zeros'.
- **lateral_conv** (*bool, optional*) – Whether to apply lateral convs to image features. Defaults to True.

forward(*img_feats*, *pts*, *pts_feats*, *img_metas*)
Forward function.

Parameters

- **img_feats** (*list[torch.Tensor]*) – Image features.
- **pts** – [list[torch.Tensor]]: A batch of points with shape N x 3.
- **pts_feats** (*torch.Tensor*) – A tensor consist of point features of the total batch.
- **img_metas** (*list[dict]*) – Meta information of images.

Returns Fused features of each point.

Return type torch.Tensor

obtain_mlvl_feats(*img_feats*, *pts*, *img_metas*)
Obtain multi-level features for each point.

Parameters

- **img_feats** (*list(torch.Tensor)*) – Multi-scale image features produced by image backbone in shape (N, C, H, W).
- **pts** (*list[torch.Tensor]*) – Points of each sample.
- **img_metas** (*list[dict]*) – Meta information for each sample.

Returns Corresponding image features of each point.

Return type torch.Tensor

sample_single(*img_feats*, *pts*, *img_meta*)
Sample features from single level image feature map.

Parameters

- **img_feats** (*torch.Tensor*) – Image feature map in shape (1, C, H, W).
- **pts** (*torch.Tensor*) – Points of a single sample.
- **img_meta** (*dict*) – Meta information of the single sample.

Returns Single level image features of each point.

Return type torch.Tensor

class mmdet3d.models.fusion_layers.**VoteFusion**(*num_classes=10*, *max_imvote_per_pixel=3*)
Fuse 2d features from 3d seeds.

Parameters

- **num_classes** (*int*) – number of classes.
- **max_imvote_per_pixel** (*int*) – max number of imvotes.

forward(*imgs*, *bboxes_2d_rescaled*, *seeds_3d_depth*, *img_metas*)
Forward function.

Parameters

- **imgs** (*list[torch.Tensor]*) – Image features.
- **bboxes_2d_rescaled** (*list[torch.Tensor]*) – 2D bboxes.
- **seeds_3d_depth** (*torch.Tensor*) – 3D seeds.
- **img_metas** (*list[dict]*) – Meta information of images.

Returns Concatenated cues of each point. torch.Tensor: Validity mask of each feature.

Return type torch.Tensor

mmdet3d.models.fusion_layers.**apply_3d_transformation**(*pcd*, *coord_type*, *img_meta*, *reverse=False*)
Apply transformation to input point cloud.

Parameters

- **pcd** (*torch.Tensor*) – The point cloud to be transformed.
- **coord_type** (*str*) – 'DEPTH' or 'CAMERA' or 'LIDAR'.
- **img_meta** (*dict*) – Meta info regarding data transformation.
- **reverse** (*bool*) – Reversed transformation or not.

Note: The elements in img_meta['transformation_3d_flow']: "T" stands for translation; "S" stands for scale; "R" stands for rotation; "HF" stands for horizontal flip; "VF" stands for vertical flip.

Returns The transformed point cloud.

Return type torch.Tensor

mmdet3d.models.fusion_layers.**bbox_2d_transform**(*img_meta*, *bbox_2d*, *ori2new*)
Transform 2d bbox according to img_meta.

Parameters

- **img_meta** (*dict*) – Meta info regarding data transformation.
- **bbox_2d** (*torch.Tensor*) – Shape (..., >4) The input 2d bboxes to transform.
- **ori2new** (*bool*) – Origin img coord system to new or not.

Returns The transformed 2d bboxes.

Return type torch.Tensor

mmdet3d.models.fusion_layers.**coord_2d_transform**(*img_meta*, *coord_2d*, *ori2new*)
Transform 2d pixel coordinates according to img_meta.

Parameters

- **img_meta** (*dict*) – Meta info regarding data transformation.
- **coord_2d** (*torch.Tensor*) – Shape (..., 2) The input 2d coords to transform.

- **ori2new** (*bool*) – Origin img coord system to new or not.

Returns The transformed 2d coordinates.

Return type torch.Tensor

47.7 losses

class mmdet3d.models.losses.**AxisAlignedIoULoss**(*reduction='mean'*, *loss_weight=1.0*)

Calculate the IoU loss (1-IoU) of axis aligned bounding boxes.

Parameters

- **reduction** (*str*) – Method to reduce losses. The valid reduction method are none, sum or mean.
- **loss_weight** (*float, optional*) – Weight of loss. Defaults to 1.0.

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

Forward function of loss calculation.

Parameters

- **pred** (*torch.Tensor*) – Bbox predictions with shape [..., 6] (x1, y1, z1, x2, y2, z2).
- **target** (*torch.Tensor*) – Bbox targets (gt) with shape [..., 6] (x1, y1, z1, x2, y2, z2).
- **weight** (*torch.Tensor | float, optional*) – Weight of loss. Defaults to None.
- **avg_factor** (*int, optional*) – Average factor that is used to average the loss. Defaults to None.
- **reduction_override** (*str, optional*) – Method to reduce losses. The valid reduction method are 'none', 'sum' or 'mean'. Defaults to None.

Returns IoU loss between predictions and targets.

Return type torch.Tensor

class mmdet3d.models.losses.**ChamferDistance**(*mode='l2'*, *reduction='mean'*, *loss_src_weight=1.0*, *loss_dst_weight=1.0*)

Calculate Chamfer Distance of two sets.

Parameters

- **mode** (*str*) – Criterion mode to calculate distance. The valid modes are smooth_l1, l1 or l2.
- **reduction** (*str*) – Method to reduce losses. The valid reduction method are none, sum or mean.
- **loss_src_weight** (*float*) – Weight of loss_source.
- **loss_dst_weight** (*float*) – Weight of loss_target.

forward(*source*, *target*, *src_weight=1.0*, *dst_weight=1.0*, *reduction_override=None*, *return_indices=False*, ***kwargs*)

Forward function of loss calculation.

Parameters

- **source** (*torch.Tensor*) – Source set with shape [B, N, C] to calculate Chamfer Distance.

- **target** (*torch.Tensor*) – Destination set with shape [B, M, C] to calculate Chamfer Distance.
- **src_weight** (*torch.Tensor | float, optional*) – Weight of source loss. Defaults to 1.0.
- **dst_weight** (*torch.Tensor | float, optional*) – Weight of destination loss. Defaults to 1.0.
- **reduction_override** (*str, optional*) – Method to reduce losses. The valid reduction method are 'none', 'sum' or 'mean'. Defaults to None.
- **return_indices** (*bool, optional*) – Whether to return indices. Defaults to False.

Returns

If return_indices=True, return losses of source and target with their corresponding indices in the order of `(loss_source, loss_target, indices1, indices2)`. If `return_indices=False`, return `(loss_source, loss_target)`.

Return type tuple[torch.Tensor]

class mmdet3d.models.losses.**FocalLoss**(*use_sigmoid=True*, *gamma=2.0*, *alpha=0.25*, *reduction='mean'*, *loss_weight=1.0*, *activated=False*)

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

class mmdet3d.models.losses.**MultiBinLoss**(*reduction='none'*, *loss_weight=1.0*)

Multi-Bin Loss for orientation.

Parameters

- **reduction** (*str, optional*) – The method to reduce the loss. Options are 'none', 'mean' and 'sum'. Defaults to 'none'.
- **loss_weight** (*float, optional*) – The weight of loss. Defaults to 1.0.

forward(*pred*, *target*, *num_dir_bins*, *reduction_override=None*)

Forward function.

Parameters

- **pred** (*torch.Tensor*) – The prediction.
- **target** (*torch.Tensor*) – The learning target of the prediction.

- **num_dir_bins** (*int*) – Number of bins to encode direction angle.
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None.

class mmdet3d.models.losses.**PAConvRegularizationLoss**(*reduction='mean'*, *loss_weight=1.0*)

Calculate correlation loss of kernel weights in PAConv's weight bank.

This is used as a regularization term in PAConv model training.

Parameters

- **reduction** (*str*) – Method to reduce losses. The reduction is performed among all PAConv modules instead of prediction tensors. The valid reduction method are none, sum or mean.
- **loss_weight** (*float, optional*) – Weight of loss. Defaults to 1.0.

forward(*modules*, *reduction_override=None*, ***kwargs*)

Forward function of loss calculation.

Parameters

- **modules** (List[nn.Module] | generator) – A list or a python generator of torch.nn.Modules.
- **reduction_override** (*str, optional*) – Method to reduce losses. The valid reduction method are 'none', 'sum' or 'mean'. Defaults to None.

Returns Correlation loss of kernel weights.

Return type torch.Tensor

class mmdet3d.models.losses.**RotatedIoU3DLoss**(*reduction='mean'*, *loss_weight=1.0*)

Calculate the IoU loss (1-IoU) of rotated bounding boxes.

Parameters

- **reduction** (*str*) – Method to reduce losses. The valid reduction method are none, sum or mean.
- **loss_weight** (*float, optional*) – Weight of loss. Defaults to 1.0.

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

Forward function of loss calculation.

Parameters

- **pred** (*torch.Tensor*) – Bbox predictions with shape [..., 7] (x, y, z, w, l, h, alpha).
- **target** (*torch.Tensor*) – Bbox targets (gt) with shape [..., 7] (x, y, z, w, l, h, alpha).
- **weight** (*torch.Tensor | float, optional*) – Weight of loss. Defaults to None.
- **avg_factor** (*int, optional*) – Average factor that is used to average the loss. Defaults to None.
- **reduction_override** (*str, optional*) – Method to reduce losses. The valid reduction method are 'none', 'sum' or 'mean'. Defaults to None.

Returns IoU loss between predictions and targets.

Return type torch.Tensor

class mmdet3d.models.losses.**SmoothL1Loss**(*beta=1.0*, *reduction='mean'*, *loss_weight=1.0*)

Smooth L1 loss.

Parameters

- **beta** (*float, optional*) – The threshold in the piecewise function. Defaults to 1.0.
- **reduction** (*str, optional*) – The method to reduce the loss. Options are "none", "mean" and "sum". Defaults to "mean".
- **loss_weight** (*float, optional*) – The weight of loss.

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

Parameters

- **pred** (*torch.Tensor*) – The prediction.
- **target** (*torch.Tensor*) – The learning target 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. Defaults to None.

class mmdet3d.models.losses.**UncertainL1Loss**(*alpha=1.0*, *reduction='mean'*, *loss_weight=1.0*)
L1 loss with uncertainty.

Parameters

- **alpha** (*float, optional*) – The coefficient of log(sigma). Defaults to 1.0.
- **reduction** (*str, optional*) – The method to reduce the loss. Options are 'none', 'mean' and 'sum'. Defaults to 'mean'.
- **loss_weight** (*float, optional*) – The weight of loss. Defaults to 1.0.

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

Parameters

- **pred** (*torch.Tensor*) – The prediction.
- **target** (*torch.Tensor*) – The learning target of the prediction.
- **sigma** (*torch.Tensor*) – The sigma for uncertainty.
- **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 mmdet3d.models.losses.**UncertainSmoothL1Loss**(*alpha=1.0*, *beta=1.0*, *reduction='mean'*, *loss_weight=1.0*)
Smooth L1 loss with uncertainty.

Please refer to PGD and Multi-Task Learning Using Uncertainty to Weigh Losses for Scene Geometry and Semantics for more details.

Parameters

- **alpha** (*float, optional*) – The coefficient of log(sigma). Defaults to 1.0.
- **beta** (*float, optional*) – The threshold in the piecewise function. Defaults to 1.0.
- **reduction** (*str, optional*) – The method to reduce the loss. Options are 'none', 'mean' and 'sum'. Defaults to 'mean'.
- **loss_weight** (*float, optional*) – The weight of loss. Defaults to 1.0

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

Parameters

- **pred** (*torch.Tensor*) – The prediction.
- **target** (*torch.Tensor*) – The learning target of the prediction.
- **sigma** (*torch.Tensor*) – The sigma for uncertainty.
- **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.

mmdet3d.models.losses.**axis_aligned_iou_loss**(*pred*, *target*)
Calculate the IoU loss (1-IoU) of two sets of axis aligned bounding boxes. Note that predictions and targets are one-to-one corresponded.

Parameters

- **pred** (*torch.Tensor*) – Bbox predictions with shape [..., 6] (x1, y1, z1, x2, y2, z2).
- **target** (*torch.Tensor*) – Bbox targets (gt) with shape [..., 6] (x1, y1, z1, x2, y2, z2).

Returns IoU loss between predictions and targets.

Return type torch.Tensor

mmdet3d.models.losses.**binary_cross_entropy**(*pred*, *label*, *weight=None*, *reduction='mean'*, *avg_factor=None*, *class_weight=None*, *ignore_index=- 100*, *avg_non_ignore=False*)
Calculate the binary CrossEntropy loss.

Parameters

- **pred** (*torch.Tensor*) – The prediction with shape (N, 1) or (N,). When the shape of pred is (N, 1), label will be expanded to one-hot format, and when the shape of pred is (N,), label will not be expanded to one-hot format.
- **label** (*torch.Tensor*) – The learning label of the prediction, with shape (N,).
- **weight** (*torch.Tensor, optional*) – Sample-wise loss weight.
- **reduction** (*str, optional*) – The method used to reduce the loss. Options are "none", "mean" and "sum".
- **avg_factor** (*int, optional*) – Average factor that is used to average the loss. Defaults to None.
- **class_weight** (*list[float], optional*) – The weight for each class.

- **ignore_index** (*int | None*) – The label index to be ignored. If None, it will be set to default value. Default: -100.
- **avg_non_ignore** (*bool*) – The flag decides to whether the loss is only averaged over non-ignored targets. Default: False.

Returns The calculated loss.

Return type torch.Tensor

mmdet3d.models.losses.**chamfer_distance**(*src*, *dst*, *src_weight=1.0*, *dst_weight=1.0*, *criterion_mode='l2'*, *reduction='mean'*)

Calculate Chamfer Distance of two sets.

Parameters

- **src** (*torch.Tensor*) – Source set with shape [B, N, C] to calculate Chamfer Distance.
- **dst** (*torch.Tensor*) – Destination set with shape [B, M, C] to calculate Chamfer Distance.
- **src_weight** (*torch.Tensor or float*) – Weight of source loss.
- **dst_weight** (*torch.Tensor or float*) – Weight of destination loss.
- **criterion_mode** (*str*) – Criterion mode to calculate distance. The valid modes are smooth_l1, l1 or l2.
- **reduction** (*str*) – Method to reduce losses. The valid reduction method are 'none', 'sum' or 'mean'.

Returns

Source and Destination loss with the corresponding indices.

- **loss_src (torch.Tensor): The min distance** from source to destination.
- **loss_dst (torch.Tensor): The min distance** from destination to source.
- **indices1 (torch.Tensor): Index the min distance point** for each point in source to destination.
- **indices2 (torch.Tensor): Index the min distance point** for each point in destination to source.

Return type tuple

47.8 middle_encoders

class mmdet3d.models.middle_encoders.**PointPillarsScatter**(*in_channels*, *output_shape*)

Point Pillar's Scatter.

Converts learned features from dense tensor to sparse pseudo image.

Parameters

- **in_channels** (*int*) – Channels of input features.
- **output_shape** (*list[int]*) – Required output shape of features.

forward(*voxel_features*, *coors*, *batch_size=None*)

Foraward function to scatter features.

forward_batch(*voxel_features*, *coors*, *batch_size*)
Scatter features of single sample.

Parameters

- **voxel_features** (*torch.Tensor*) – Voxel features in shape (N, C).
- **coors** (*torch.Tensor*) – Coordinates of each voxel in shape (N, 4). The first column indicates the sample ID.
- **batch_size** (*int*) – Number of samples in the current batch.

forward_single(*voxel_features*, *coors*)
Scatter features of single sample.

Parameters

- **voxel_features** (*torch.Tensor*) – Voxel features in shape (N, C).
- **coors** (*torch.Tensor*) – Coordinates of each voxel. The first column indicates the sample ID.

class mmdet3d.models.middle_encoders.**SparseEncoder**(*in_channels*, *sparse_shape*, *order=('conv', 'norm', 'act')*, *norm_cfg={'eps': 0.001, 'momentum': 0.01, 'type': 'BN1d'}*, *base_channels=16*, *output_channels=128*, *encoder_channels=((16), (32, 32, 32), (64, 64, 64), (64, 64, 64))*, *encoder_paddings=((1), (1, 1, 1), (1, 1, 1), ((0, 1, 1), 1, 1))*, *block_type='conv_module'*)
Sparse encoder for SECOND and Part-A2.

Parameters

- **in_channels** (*int*) – The number of input channels.
- **sparse_shape** (*list[int]*) – The sparse shape of input tensor.
- **order** (*list[str], optional*) – Order of conv module. Defaults to (‘conv’, ‘norm’, ‘act’).
- **norm_cfg** (*dict, optional*) – Config of normalization layer. Defaults to dict(type=’BN1d’, eps=1e-3, momentum=0.01).
- **base_channels** (*int, optional*) – Out channels for conv_input layer. Defaults to 16.
- **output_channels** (*int, optional*) – Out channels for conv_out layer. Defaults to 128.
- **encoder_channels** (*tuple[tuple[int]], optional*) – Convolutional channels of each encode block. Defaults to ((16,), (32, 32, 32), (64, 64, 64), (64, 64, 64)).
- **encoder_paddings** (*tuple[tuple[int]], optional*) – Paddings of each encode block. Defaults to ((1,), (1, 1, 1), (1, 1, 1), ((0, 1, 1), 1, 1)).
- **block_type** (*str, optional*) – Type of the block to use. Defaults to ‘conv_module’.

forward(*voxel_features*, *coors*, *batch_size*)
Forward of SparseEncoder.

Parameters

- **voxel_features** (*torch.Tensor*) – Voxel features in shape (N, C).
- **coors** (*torch.Tensor*) – Coordinates in shape (N, 4), the columns in the order of (batch_idx, z_idx, y_idx, x_idx).

- **batch_size** (*int*) – Batch size.

Returns Backbone features.

Return type dict

make_encoder_layers(*make_block*, *norm_cfg*, *in_channels*, *block_type='conv_module'*, *conv_cfg={'type': 'SubMConv3d'}*)

make encoder layers using sparse convs.

Parameters

- **make_block** (*method*) – A bounded function to build blocks.
- **norm_cfg** (*dict[str]*) – Config of normalization layer.
- **in_channels** (*int*) – The number of encoder input channels.
- **block_type** (*str, optional*) – Type of the block to use. Defaults to 'conv_module'.
- **conv_cfg** (*dict, optional*) – Config of conv layer. Defaults to dict(type='SubMConv3d').

Returns The number of encoder output channels.

Return type int

class mmdet3d.models.middle_encoders.**SparseEncoderSASSD**(*in_channels*, *sparse_shape*, *order=('conv', 'norm', 'act')*, *norm_cfg={'eps': 0.001, 'momentum': 0.01, 'type': 'BN1d'}*, *base_channels=16*, *output_channels=128*, *encoder_channels=((16), (32, 32, 32), (64, 64, 64), (64, 64, 64))*, *encoder_paddings=((1), (1, 1, 1), (1, 1, 1), ((0, 1, 1), 1, 1))*, *block_type='conv_module'*)

Sparse encoder for SASSD

Parameters

- **in_channels** (*int*) – The number of input channels.
- **sparse_shape** (*list[int]*) – The sparse shape of input tensor.
- **order** (*list[str], optional*) – Order of conv module. Defaults to ('conv', 'norm', 'act').
- **norm_cfg** (*dict, optional*) – Config of normalization layer. Defaults to dict(type='BN1d', eps=1e-3, momentum=0.01).
- **base_channels** (*int, optional*) – Out channels for conv_input layer. Defaults to 16.
- **output_channels** (*int, optional*) – Out channels for conv_out layer. Defaults to 128.
- **encoder_channels** (*tuple[tuple[int]], optional*) – Convolutional channels of each encode block. Defaults to ((16,), (32, 32, 32), (64, 64, 64), (64, 64, 64)).
- **encoder_paddings** (*tuple[tuple[int]], optional*) – Paddings of each encode block. Defaults to ((1,), (1, 1, 1), (1, 1, 1), ((0, 1, 1), 1, 1)).
- **block_type** (*str, optional*) – Type of the block to use. Defaults to 'conv_module'.

aux_loss(*points*, *point_cls*, *point_reg*, *gt_bboxes*)
Calculate auxiliary loss.

Parameters

- **points** (*torch.Tensor*) – Mean feature value of the points.
- **point_cls** (*torch.Tensor*) – Classificaion result of the points.
- **point_reg** (*torch.Tensor*) – Regression offsets of the points.
- **gt_bboxes** (list[BaseInstance3DBoxes]) – Ground truth boxes for each sample.

Returns Backbone features.

Return type dict

calculate_pts_offsets(*points*, *boxes*)
Find all boxes in which each point is, as well as the offsets from the box centers.

Parameters

- **points** (*torch.Tensor*) – [M, 3], [x, y, z] in LiDAR/DEPTH coordinate
- **boxes** (*torch.Tensor*) – [T, 7], num_valid_boxes <= T, [x, y, z, x_size, y_size, z_size, rz], (x, y, z) is the bottom center.

Returns

Point indices of boxes with the shape of (T, M). Default background = 0. And offsets from the box centers of points, if it belows to the box, with the shape of (M, 3). Default background = 0.

Return type tuple[torch.Tensor]

forward(*voxel_features*, *coors*, *batch_size*, *test_mode=False*)
Forward of SparseEncoder.

Parameters

- **voxel_features** (*torch.Tensor*) – Voxel features in shape (N, C).
- **coors** (*torch.Tensor*) – Coordinates in shape (N, 4), the columns in the order of (batch_idx, z_idx, y_idx, x_idx).
- **batch_size** (*int*) – Batch size.
- **test_mode** (*bool, optional*) – Whether in test mode. Defaults to False.

Returns

Backbone features. tuple[torch.Tensor]: Mean feature value of the points,

Classificaion result of the points, Regression offsets of the points.

Return type dict

get_auxiliary_targets(*nxyz*, *gt_boxes3d*, *enlarge=1.0*)
Get auxiliary target.

Parameters

- **nxyz** (*torch.Tensor*) – Mean features of the points.
- **gt_boxes3d** (*torch.Tensor*) – Coordinates in shape (N, 4), the columns in the order of (batch_idx, z_idx, y_idx, x_idx).
- **enlarge** (*int, optional*) – Enlaged scale. Defaults to 1.0.

Returns

Label of the points and center offsets of the points.

Return type tuple[torch.Tensor]

make_auxiliary_points(*source_tensor*, *target*, *offset=(0.0, - 40.0, - 3.0)*, *voxel_size=(0.05, 0.05, 0.1)*)
Make auxiliary points for loss computation.

Parameters

- **source_tensor** (*torch.Tensor*) – (M, C) features to be propigated.
- **target** (*torch.Tensor*) – (N, 4) bxyz positions of the target features.
- **offset** (*tuple[float], optional*) – Voxelization offset. Defaults to (0., -40., -3.)
- **voxel_size** (*tuple[float], optional*) – Voxelization size. Defaults to (.05, .05, .1)

Returns (N, C) tensor of the features of the target features.

Return type torch.Tensor

class mmdet3d.models.middle_encoders.**SparseUNet**(*in_channels*, *sparse_shape*, *order=('conv', 'norm', 'act')*, *norm_cfg={'eps': 0.001, 'momentum': 0.01, 'type': 'BN1d'}*, *base_channels=16*, *output_channels=128*, *encoder_channels=((16), (32, 32, 32), (64, 64, 64), (64, 64, 64))*, *encoder_paddings=((1), (1, 1, 1), (1, 1, 1), ((0, 1, 1), 1, 1))*, *decoder_channels=((64, 64, 64), (64, 64, 32), (32, 32, 16), (16, 16, 16))*, *decoder_paddings=((1, 0), (1, 0), (0, 0), (0, 1))*, *init_cfg=None*)

SparseUNet for PartA^2.

See the paper for more details.

Parameters

- **in_channels** (*int*) – The number of input channels.
- **sparse_shape** (*list[int]*) – The sparse shape of input tensor.
- **norm_cfg** (*dict*) – Config of normalization layer.
- **base_channels** (*int*) – Out channels for conv_input layer.
- **output_channels** (*int*) – Out channels for conv_out layer.
- **encoder_channels** (*tuple[tuple[int]]*) – Convolutional channels of each encode block.
- **encoder_paddings** (*tuple[tuple[int]]*) – Paddings of each encode block.
- **decoder_channels** (*tuple[tuple[int]]*) – Convolutional channels of each decode block.
- **decoder_paddings** (*tuple[tuple[int]]*) – Paddings of each decode block.

decoder_layer_forward(*x_lateral*, *x_bottom*, *lateral_layer*, *merge_layer*, *upsample_layer*)
Forward of upsample and residual block.

Parameters

- **x_lateral** (SparseConvTensor) – Lateral tensor.
- **x_bottom** (SparseConvTensor) – Feature from bottom layer.

- **lateral_layer** (*SparseBasicBlock*) – Convolution for lateral tensor.
- **merge_layer** (*SparseSequential*) – Convolution for merging features.
- **upsample_layer** (*SparseSequential*) – Convolution for upsampling.

Returns Upsampled feature.

Return type SparseConvTensor

forward(*voxel_features*, *coors*, *batch_size*)
Forward of SparseUNet.

Parameters

- **voxel_features** (*torch.float32*) – Voxel features in shape [N, C].
- **coors** (*torch.int32*) – Coordinates in shape [N, 4], the columns in the order of (batch_idx, z_idx, y_idx, x_idx).
- **batch_size** (*int*) – Batch size.

Returns Backbone features.

Return type dict[str, torch.Tensor]

make_decoder_layers(*make_block*, *norm_cfg*, *in_channels*)
make decoder layers using sparse convs.

Parameters

- **make_block** (*method*) – A bounded function to build blocks.
- **norm_cfg** (*dict[str]*) – Config of normalization layer.
- **in_channels** (*int*) – The number of encoder input channels.

Returns The number of encoder output channels.

Return type int

make_encoder_layers(*make_block*, *norm_cfg*, *in_channels*)
make encoder layers using sparse convs.

Parameters

- **make_block** (*method*) – A bounded function to build blocks.
- **norm_cfg** (*dict[str]*) – Config of normalization layer.
- **in_channels** (*int*) – The number of encoder input channels.

Returns The number of encoder output channels.

Return type int

static reduce_channel(*x*, *out_channels*)
reduce channel for element-wise addition.

Parameters

- **x** (SparseConvTensor) – Sparse tensor, x.features are in shape (N, C1).
- **out_channels** (*int*) – The number of channel after reduction.

Returns Channel reduced feature.

Return type SparseConvTensor

47.9 model_utils

class mmdet3d.models.model_utils.**EdgeFusionModule**(*out_channels*, *feat_channels*, *kernel_size=3*, *act_cfg={'type': 'ReLU'}*, *norm_cfg={'type': 'BN1d'}*)

Edge Fusion Module for feature map.

Parameters

- **out_channels** (*int*) – The number of output channels.
- **feat_channels** (*int*) – The number of channels in feature map during edge feature fusion.
- **kernel_size** (*int, optional*) – Kernel size of convolution. Default: 3.
- **act_cfg** (*dict, optional*) – Config of activation. Default: dict(type='ReLU').
- **norm_cfg** (*dict, optional*) – Config of normalization. Default: dict(type='BN1d')).

forward(*features*, *fused_features*, *edge_indices*, *edge_lens*, *output_h*, *output_w*)

Forward pass.

Parameters

- **features** (*torch.Tensor*) – Different representative features for fusion.
- **fused_features** (*torch.Tensor*) – Different representative features to be fused.
- **edge_indices** (*torch.Tensor*) – Batch image edge indices.
- **edge_lens** (*list[int]*) – List of edge length of each image.
- **output_h** (*int*) – Height of output feature map.
- **output_w** (*int*) – Width of output feature map.

Returns Fused feature maps.

Return type torch.Tensor

class mmdet3d.models.model_utils.**GroupFree3DMHA**(*embed_dims*, *num_heads*, *attn_drop=0.0*, *proj_drop=0.0*, *dropout_layer={'drop_prob': 0.0, 'type': 'DropOut'}*, *init_cfg=None*, *batch_first=False*, ***kwargs*)

A warpper for torch.nn.MultiheadAttention for GroupFree3D.

This module implements MultiheadAttention with identity connection, and positional encoding used in DETR is also passed as input.

Parameters

- **embed_dims** (*int*) – The embedding dimension.
- **num_heads** (*int*) – Parallel attention heads. Same as *nn.MultiheadAttention*.
- **attn_drop** (*float, optional*) – A Dropout layer on attn_output_weights. Defaults to 0.0.
- **proj_drop** (*float, optional*) – A Dropout layer. Defaults to 0.0.
- **(obj** (*init_cfg*) – *ConfigDict*, optional): The dropout_layer used when adding the shortcut.
- **(obj** – *mmcv.ConfigDict*, optional): The Config for initialization. Default: None.

- **batch_first** (*bool, optional*) – Key, Query and Value are shape of (batch, n, embed_dim) or (n, batch, embed_dim). Defaults to False.

forward(*query*, *key*, *value*, *identity*, *query_pos=None*, *key_pos=None*, *attn_mask=None*, *key_padding_mask=None*, ***kwargs*)

Forward function for *GroupFree3DMHA*.

**kwargs allow passing a more general data flow when combining with other operations in *transformerlayer*.

Parameters

- **query** (*Tensor*) – The input query with shape [num_queries, bs, embed_dims]. Same in *nn.MultiheadAttention.forward*.
- **key** (*Tensor*) – The key tensor with shape [num_keys, bs, embed_dims]. Same in *nn.MultiheadAttention.forward*. If None, the `query` will be used.
- **value** (*Tensor*) – The value tensor with same shape as *key*. Same in *nn.MultiheadAttention.forward*. If None, the *key* will be used.
- **identity** (*Tensor*) – This tensor, with the same shape as x, will be used for the identity link. If None, *x* will be used.
- **query_pos** (*Tensor, optional*) – The positional encoding for query, with the same shape as *x*. Defaults to None. If not None, it will be added to *x* before forward function.
- **key_pos** (*Tensor, optional*) – The positional encoding for *key*, with the same shape as *key*. Defaults to None. If not None, it will be added to *key* before forward function. If None, and *query_pos* has the same shape as *key*, then *query_pos* will be used for *key_pos*. Defaults to None.
- **attn_mask** (*Tensor, optional*) – ByteTensor mask with shape [num_queries, num_keys]. Same in *nn.MultiheadAttention.forward*. Defaults to None.
- **key_padding_mask** (*Tensor, optional*) – ByteTensor with shape [bs, num_keys]. Same in *nn.MultiheadAttention.forward*. Defaults to None.

Returns forwarded results with shape [num_queries, bs, embed_dims].

Return type Tensor

class mmdet3d.models.model_utils.**VoteModule**(*in_channels*, *vote_per_seed=1*, *gt_per_seed=3*, *num_points=- 1*, *conv_channels=(16, 16)*, *conv_cfg={'type': 'Conv1d'}*, *norm_cfg={'type': 'BN1d'}*, *act_cfg={'type': 'ReLU'}*, *norm_feats=True*, *with_res_feat=True*, *vote_xyz_range=None*, *vote_loss=None*)

Vote module.

Generate votes from seed point features.

Parameters

- **in_channels** (*int*) – Number of channels of seed point features.
- **vote_per_seed** (*int, optional*) – Number of votes generated from each seed point. Default: 1.
- **gt_per_seed** (*int, optional*) – Number of ground truth votes generated from each seed point. Default: 3.
- **num_points** (*int, optional*) – Number of points to be used for voting. Default: 1.

- **conv_channels** (*tuple[int], optional*) – Out channels of vote generating convolution. Default: (16, 16).
- **conv_cfg** (*dict, optional*) – Config of convolution. Default: dict(type='Conv1d').
- **norm_cfg** (*dict, optional*) – Config of normalization. Default: dict(type='BN1d').
- **norm_feats** (*bool, optional*) – Whether to normalize features. Default: True.
- **with_res_feat** (*bool, optional*) – Whether to predict residual features. Default: True.
- **vote_xyz_range** (*list[float], optional*) – The range of points translation. Default: None.
- **vote_loss** (*dict, optional*) – Config of vote loss. Default: None.

forward(*seed_points*, *seed_feats*)
forward.

Parameters

- **seed_points** (*torch.Tensor*) – Coordinate of the seed points in shape (B, N, 3).
- **seed_feats** (*torch.Tensor*) – Features of the seed points in shape (B, C, N).

Returns

- **vote_points: Voted xyz based on the seed points** with shape (B, M, 3), M=num_seed*vote_per_seed.
- **vote_features: Voted features based on the seed points with** shape (B, C, M) where M=num_seed*vote_per_seed, C=vote_feature_dim.

Return type tuple[torch.Tensor]

get_loss(*seed_points*, *vote_points*, *seed_indices*, *vote_targets_mask*, *vote_targets*)
Calculate loss of voting module.

Parameters

- **seed_points** (*torch.Tensor*) – Coordinate of the seed points.
- **vote_points** (*torch.Tensor*) – Coordinate of the vote points.
- **seed_indices** (*torch.Tensor*) – Indices of seed points in raw points.
- **vote_targets_mask** (*torch.Tensor*) – Mask of valid vote targets.
- **vote_targets** (*torch.Tensor*) – Targets of votes.

Returns Weighted vote loss.

Return type torch.Tensor

CHAPTER

FORTYEIGHT

ENGLISH

CHAPTER

FORTYNINE

CHAPTER

FIFTY

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

m

INDEX

A

B

C

D

E

F

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y