
mmcv Documentation

Release 1.1.0

MMCV Contributors

Apr 19, 2023

GET STARTED

1	Get Started	1
2	2: Train with customized datasets	7
3	Backends Support	11
4	Learn about Configs	15
5	Coordinate System	27
6	Customize Data Pipelines	35
7	Dataset Preparation	39
8	Inference	43
9	Model Deployment	47
10	Inference and train with existing models and standard datasets	49
11	Useful Tools	55
12	Visualization	61
13	Datasets	67
14	Supported Tasks	107
15	Customization	115
16	Environment	147
17	Dataset	149
18	Model	151
19	mmdet3d.apis	153
20	mmdet3d.datasets	155
21	mmdet3d.engine	157
22	mmdet3d.evaluation	159

23	mmdet3d.models	161
24	mmdet3d.structures	163
25	mmdet3d.testing	187
26	mmdet3d.visualization	189
27	mmdet3d.utils	191
28	Model Zoo	195
29	Benchmarks	199
30	Changelog of v1.0.x	205
31	Changelog of v1.1	231
32	Compatibility	241
33	FAQ	249
34	English	251
35		253
36	Indices and tables	255
	Python Module Index	257
	Index	259

GET STARTED

1.1 Prerequisites

In this section, we demonstrate how to prepare an environment with PyTorch.

MMDetection3D works on Linux, Windows (experimental support) and macOS. It requires Python 3.7+, CUDA 9.2+, and PyTorch 1.6+.

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
```

1.2 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.

1.2.1 Best Practices

Step 0. Install `MMEEngine`, `MMCV` and `MMDetection` using `MIM`.

```
pip install -U openmim
mim install mmengine
mim install 'mmcv>=2.0.0rc4'
mim install 'mmdet>=3.0.0'
```

Note: In `MMCV-v2.x`, `mmcv-full` is renamed to `mmcv`, if you want to install `mmcv` without CUDA ops, you can use `mim install "mmcv-lite>=2.0.0rc4"` to install the lite version.

Step 1. Install `MMDetection3D`.

Case a: If you develop and run `mmdet3d` directly, install it from source:

```
git clone https://github.com/open-mmlab/mmdetection3d.git -b dev-1.x
# "-b dev-1.x" means checkout to the `dev-1.x` branch.
cd mmdetection3d
pip install -v -e .
# "-v" means verbose, or more output
# "-e" means installing a project in editable mode,
# thus any local modifications made to the code will take effect without reinstallation.
```

Case b: If you use `mmdet3d` as a dependency or third-party package, install it with `MIM`:

```
mim install "mmdet3d>=1.1.0rc0"
```

Note:

1. If you would like to use `opencv-python-headless` instead of `opencv-python`, you can install it before installing `MMCV`.
2. 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 `spconv 2.0`. If the user has installed `spconv 2.0`, the code will use `spconv 2.0` first, which will take up less GPU memory than using the default `mmcv spconv`. Users can use the following commands to install `spconv 2.0`:

```
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` to install it:

```
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"
```

We also support Torchsparse as a sparse convolution backend. If necessary please follow original [installation guide](#) or use pip to install it:

```
sudo apt-get install libsparsehash-dev
pip install --upgrade git+https://github.com/mit-han-lab/torchsparse.git@v1.4.0
```

or omit sudo install by following command:

```
conda install -c bioconda sparsehash
export CPLUS_INCLUDE_PATH=CPLUS_INCLUDE_PATH:${YOUR_CONDA_ENVS_DIR}/include
pip install --upgrade git+https://github.com/mit-han-lab/torchsparse.git@v1.4.0
```

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

1.2.2 Verify the Installation

To verify whether MMDetection3D is installed correctly, we provide some sample codes to run an inference demo.

Step 1. We need to download config and checkpoint files.

```
mim download mmdet3d --config pointpillars_hv_secfn_8xb6-160e_kitti-3d-car --dest .
```

The downloading will take several seconds or more, depending on your network environment. When it is done, you will find two files `pointpillars_hv_secfn_8xb6-160e_kitti-3d-car.py` and `hv_pointpillars_secfn_8xb6-160e_kitti-3d-car_20220331_134606-d42d15ed.pth` in your current folder.

Step 2. Verify the inference demo.

Case a: If you install MMDetection3D from source, just run the following command.

```
python demo/pcd_demo.py demo/data/kitti/000008.bin pointpillars_hv_secfn_8xb6-160e_
kitti-3d-car.py hv_pointpillars_secfn_8xb6-160e_kitti-3d-car_20220331_134606-d42d15ed.
pth --show
```

You will see a visualizer interface with point cloud, where bounding boxes are plotted on cars.

Note:

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 run 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
```

(continues on next page)

(continued from previous page)

```
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')
```

Case b: If you install MMDetection3D with MIM, open your python interpreter and copy&paste the following codes.

```
from mmdet3d.apis import init_model, inference_detector

config_file = 'pointpillars_hv_secfn_8xb6-160e_kitti-3d-car.py'
checkpoint_file = 'hv_pointpillars_secfn_6x8-160e_kitti-3d-car_20220331_134606-d42d15ed.
↳pth'
model = init_model(config_file, checkpoint_file)
inference_detector(model, 'demo/data/kitti/000008.bin')
```

You will see a list of `Det3DDataSample`, and the predictions are in the `pred_instances_3d`, indicating the detected bounding boxes, labels, and scores.

1.2.3 Customize Installation

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.

Install MMEEngine without MIM

To install MMEEngine with pip instead of MIM, please follow [MMEEngine installation guides](#).

For example, you can install MMEEngine by the following command:

```
pip install mmengine
```

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 built for PyTorch 1.12.x and CUDA 11.6:

```
pip install "mmdcv>=2.0.0rc4" -f https://download.openmmlab.com/mmdcv/dist/cu116/torch1.12.
↪0/index.html
```

Install on Google Colab

[Google Colab](#) usually has PyTorch installed, thus we only need to install MMEEngine, MMCV, MMDetection, and MMDetection3D with the following commands.

Step 1. Install MMEEngine, MMCV and MMDetection using MIM.

```
!pip3 install openmim
!mim install mmengine
!mim install "mmdcv>=2.0.0rc4,<2.1.0"
!mim install "mmdet>=3.0.0,<3.1.0"
```

Step 2. Install MMDetection3D from source.

```
!git clone https://github.com/open-mmlab/mmdetection3d.git -b dev-1.x
%cd mmdetection3d
!pip install -e .
```

Step 3. Verification.

```
import mmdet3d
print(mmdet3d.__version__)
# Example output: 1.1.0rc0, or an another version.
```

Note: Within Jupyter, the exclamation mark ! is used to call external executables and %cd is a [magic command](#) to change the current working directory of Python.

Using MMDetection3D with Docker

We provide a [Dockerfile](#) to build an image. Ensure that your `docker` version `>= 19.03`.

```
# build an image with PyTorch 1.9, CUDA 11.1
# If you prefer other versions, just modified the Dockerfile
docker build -t mmdetection3d docker/
```

Run it with:

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

1.2.4 Troubleshooting

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.

1.2.5 Use Multiple Versions of MMDetection3D in Development

Training and testing scripts have already been modified in `PYTHONPATH` in order to make sure the scripts are using their own versions of MMDetection3D.

To install the default version of MMDetection3D in your environment, you can exclude the following code in the related scripts:

```
PYTHONPATH="$(dirname $0)/..":$PYTHONPATH
```

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.

2.1 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 standard 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, you can refer to this to reorganize your dataset into kitti format. About the standard format dataset, and you can refer to [customize_dataset.md](#).

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).

2.1.1 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
```

(continues on next page)

(continued from previous page)

		— ImageSets
		— 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 implement WaymoMetric inherited from KittiMetric for evaluation.

Specifically, we implement a waymo [converter](#) to convert Waymo data into KITTI format and a waymo dataset [class](#) to process it, in addition need to add a waymo [metric](#) to evaluate results. Because we preprocess the raw data and reorganize it like KITTI, the dataset class could be implemented more easily by inheriting from KittiDataset. Regarding the dataset evaluation metric, because Waymo has its own evaluation approach, we need further implement a new Waymo metric; more about the metric could refer to [metric_and_evaluator.md](#). Afterward, users can successfully convert the data format and use WaymoDataset to train and evaluate the model by WaymoMetric.

For more details about the intermediate results of preprocessing of Waymo dataset, please refer to its [waymo_det.md](#).

2.2 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](#).

2.3 Train a new model

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

```
python tools/train.py configs/pointpillars/pointpillars_hv_secfpn_sbn-all_16xb2-2x_
↪waymoD5-3d-3class.py
```

For more detailed usages, please refer to the [Case 1](#).

2.4 Test and inference

To test the trained model, you can simply run

```
python tools/test.py configs/pointpillars/pointpillars_hv_secfpn_sbn-all_16xb2-2x_
↪waymoD5-3d-3class.py work_dirs/pointpillars_hv_secfpn_sbn-all_16xb2-2x_waymoD5-3d-
↪3class/latest.pth
```

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](#).

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.

3.1 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
backend_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,
        backend_args=backend_args),
    # set file client for data base sampler to load db info file
    backend_args=backend_args)

train_pipeline = [
    # set file client for loading training data
    dict(type='LoadPointsFromFile', coord_type='LIDAR', load_dim=4, use_dim=4, backend_
↪args=backend_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, backend_
↪args=backend_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, backend_
↪args=backend_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, backend_args=backend_
↪args)),
        # set file client for loading validation info files (.pkl)
        val=dict(pipeline=test_pipeline, classes=class_names, backend_args=backend_args),
        # set file client for loading testing info files (.pkl)
        test=dict(pipeline=test_pipeline, classes=class_names, backend_args=backend_args))

```

3.2 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
        ...

```

3.3 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.pth'
resume_from = None
workflow = [('train', 1)]

```

3.4 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')

```

3.5 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')
```

3.6 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),
    ])
```

LEARN ABOUT CONFIGS

MMDetection3D and other OpenMMLab repositories use [MMEngine's config system](#). It has a modular and inheritance design, which is convenient to conduct various experiments.

4.1 Config file content

MMDetection3D uses a modular design, all modules with different functions can be configured through the config. Taking PointPillars as an example, we will introduce each field in the config according to different function modules.

4.1.1 Model config

In MMDetection3D's config, we use `model` to setup detection algorithm components. In addition to neural network components such as `voxel_encoder`, `backbone` etc, it also requires `data_preprocessor`, `train_cfg`, and `test_cfg`. `data_preprocessor` is responsible for processing a batch of data output by dataloader. `train_cfg` and `test_cfg` in the model config are training and testing hyperparameters of the components.

```
model = dict(  
    type='VoxelNet',  
    data_preprocessor=dict(  
        type='Det3DDataPreprocessor',  
        voxel=True,  
        voxel_layer=dict(  
            max_num_points=32,  
            point_cloud_range=[0, -39.68, -3, 69.12, 39.68, 1],  
            voxel_size=[0.16, 0.16, 4],  
            max_voxels=(16000, 40000)),  
        voxel_encoder=dict(  
            type='PillarFeatureNet',  
            in_channels=4,  
            feat_channels=[64],  
            with_distance=False,  
            voxel_size=[0.16, 0.16, 4],  
            point_cloud_range=[0, -39.68, -3, 69.12, 39.68, 1]),  
        middle_encoder=dict(  
            type='PointPillarsScatter', in_channels=64, output_shape=[496, 432]),  
        backbone=dict(  
            type='SECOND',  
            in_channels=64,  
            layer_nums=[3, 5, 5],
```

(continues on next page)

(continued from previous page)

```

        layer_strides=[2, 2, 2],
        out_channels=[64, 128, 256]),
    neck=dict(
        type='SECONDFPN',
        in_channels=[64, 128, 256],
        upsample_strides=[1, 2, 4],
        out_channels=[128, 128, 128]),
    bbox_head=dict(
        type='Anchor3DHead',
        num_classes=3,
        in_channels=384,
        feat_channels=384,
        use_direction_classifier=True,
        assign_per_class=True,
        anchor_generator=dict(
            type='AlignedAnchor3DRangeGenerator',
            ranges=[[0, -39.68, -0.6, 69.12, 39.68, -0.6],
                    [0, -39.68, -0.6, 69.12, 39.68, -0.6],
                    [0, -39.68, -1.78, 69.12, 39.68, -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),
        diff_rad_by_sin=True,
        bbox_coder=dict(type='DeltaXYZWLHRBBoxCoder'),
        loss_cls=dict(
            type='mmdet.FocalLoss',
            use_sigmoid=True,
            gamma=2.0,
            alpha=0.25,
            loss_weight=1.0),
        loss_bbox=dict(
            type='mmdet.SmoothL1Loss',
            beta=0.1111111111111111,
            loss_weight=2.0),
        loss_dir=dict(
            type='mmdet.CrossEntropyLoss', use_sigmoid=False,
            loss_weight=0.2)),
    train_cfg=dict(
        assigner=[
            dict(
                type='Max3DIoUAssigner',
                iou_calculator=dict(type='BboxOverlapsNearest3D'),
                pos_iou_thr=0.5,
                neg_iou_thr=0.35,
                min_pos_iou=0.35,
                ignore_iof_thr=-1),
            dict(
                type='Max3DIoUAssigner',
                iou_calculator=dict(type='BboxOverlapsNearest3D'),
                pos_iou_thr=0.5,
                neg_iou_thr=0.35,
                min_pos_iou=0.35,

```

(continues on next page)

(continued from previous page)

```

        ignore_iof_thr=-1),
    dict(
        type='Max3DIoUAssigner',
        iou_calculator=dict(type='BboxOverlapsNearest3D'),
        pos_iou_thr=0.6,
        neg_iou_thr=0.45,
        min_pos_iou=0.45,
        ignore_iof_thr=-1)
    ],
    allowed_border=0,
    pos_weight=-1,
    debug=False),
test_cfg=dict(
    use_rotate_nms=True,
    nms_across_levels=False,
    nms_thr=0.01,
    score_thr=0.1,
    min_bbox_size=0,
    nms_pre=100,
    max_num=50))

```

4.1.2 Dataset and evaluator config

`Dataloaders` are required for the training, validation, and testing of the `runner`. Dataset and data pipeline need to be set to build the dataloader. Due to the complexity of this part, we use intermediate variables to simplify the writing of dataloader configs.

```

dataset_type = 'KittiDataset'
data_root = 'data/kitti/'
class_names = ['Pedestrian', 'Cyclist', 'Car']
point_cloud_range = [0, -39.68, -3, 69.12, 39.68, 1]
input_modality = dict(use_lidar=True, use_camera=False)
metainfo = dict(classes=class_names)

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, Pedestrian=5, Cyclist=5)),
    classes=class_names,
    sample_groups=dict(Car=15, Pedestrian=15, Cyclist=15),
    points_loader=dict(
        type='LoadPointsFromFile', coord_type='LIDAR', load_dim=4, use_dim=4))

train_pipeline = [
    dict(type='LoadPointsFromFile', coord_type='LIDAR', load_dim=4, use_dim=4),
    dict(type='LoadAnnotations3D', with_bbox_3d=True, with_label_3d=True),
    dict(type='ObjectSample', db_sampler=db_sampler, use_ground_plane=True),
    dict(type='RandomFlip3D', flip_ratio_bev_horizontal=0.5),

```

(continues on next page)

(continued from previous page)

```

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='Pack3DDetInputs',
    keys=['points', 'gt_labels_3d', 'gt_bboxes_3d'])
]
test_pipeline = [
    dict(type='LoadPointsFromFile', coord_type='LIDAR', load_dim=4, use_dim=4),
    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='Pack3DDetInputs', keys=['points'])
]
eval_pipeline = [
    dict(type='LoadPointsFromFile', coord_type='LIDAR', load_dim=4, use_dim=4),
    dict(type='Pack3DDetInputs', keys=['points'])
]
train_dataloader = dict(
    batch_size=6,
    num_workers=4,
    persistent_workers=True,
    sampler=dict(type='DefaultSampler', shuffle=True),
    dataset=dict(
        type='RepeatDataset',
        times=2,
        dataset=dict(
            type=dataset_type,
            data_root=data_root,
            ann_file='kitti_infos_train.pkl',
            data_prefix=dict(pts='training/velodyne_reduced'),
            pipeline=train_pipeline,
            modality=input_modality,
            test_mode=False,
            metainfo=metainfo,
            box_type_3d='LiDAR'))))

```

(continues on next page)

(continued from previous page)

```

val_dataloader = dict(
    batch_size=1,
    num_workers=1,
    persistent_workers=True,
    drop_last=False,
    sampler=dict(type='DefaultSampler', shuffle=False),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        data_prefix=dict(pts='training/velodyne_reduced'),
        ann_file='kitti_infos_val.pkl',
        pipeline=test_pipeline,
        modality=input_modality,
        test_mode=True,
        metainfo=metainfo,
        box_type_3d='LiDAR'))
test_dataloader = dict(
    batch_size=1,
    num_workers=1,
    persistent_workers=True,
    drop_last=False,
    sampler=dict(type='DefaultSampler', shuffle=False),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        data_prefix=dict(pts='training/velodyne_reduced'),
        ann_file='kitti_infos_val.pkl',
        pipeline=test_pipeline,
        modality=input_modality,
        test_mode=True,
        metainfo=metainfo,
        box_type_3d='LiDAR'))

```

Evaluators are used to compute the metrics of the trained model on the validation and testing datasets. The config of evaluators consists of one or a list of metric configs:

```

val_evaluator = dict(
    type='KittiMetric',
    ann_file=data_root + 'kitti_infos_val.pkl',
    metric='bbox')
test_evaluator = val_evaluator

```

Since the test dataset has no annotation files, the test_dataloader and test_evaluator config in MMDetection3D are generally equal to the val's. If you want to save the detection results on the test dataset, you can write the config like this:

```

# inference on test dataset and
# format the output results for submission.
test_dataloader = dict(
    batch_size=1,
    num_workers=1,
    persistent_workers=True,

```

(continues on next page)

(continued from previous page)

```

drop_last=False,
sampler=dict(type='DefaultSampler', shuffle=False),
dataset=dict(
    type=dataset_type,
    data_root=data_root,
    data_prefix=dict(pts='testing/velodyne_reduced'),
    ann_file='kitti_infos_test.pkl',
    load_eval_anns=False,
    pipeline=test_pipeline,
    modality=input_modality,
    test_mode=True,
    metainfo=metainfo,
    box_type_3d='LiDAR'))
test_evaluator = dict(
    type='KittiMetric',
    ann_file=data_root + 'kitti_infos_test.pkl',
    metric='bbox',
    format_only=True,
    submission_prefix='results/kitti-3class/kitti_results')

```

4.1.3 Training and testing config

MMEngine's runner uses Loop to control the training, validation, and testing processes. Users can set the maximum training epochs and validation intervals with these fields:

```

train_cfg = dict(
    type='EpochBasedTrainLoop',
    max_epochs=80,
    val_interval=2)
val_cfg = dict(type='ValLoop')
test_cfg = dict(type='TestLoop')

```

4.1.4 Optimization config

optim_wrapper is the field to configure optimization-related settings. The optimizer wrapper not only provides the functions of the optimizer, but also supports functions such as gradient clipping, mixed precision training, etc. Find more in [optimizer wrapper tutorial](#).

```

optim_wrapper = dict( # Optimizer wrapper config
    type='OptimWrapper', # Optimizer wrapper type, switch to AmpOptimWrapper to enable_
    ↪mixed precision training.
    optimizer=dict( # Optimizer config. Support all kinds of optimizers in PyTorch.
    ↪Refer to https://pytorch.org/docs/stable/optim.html#algorithms
        type='AdamW', lr=0.001, betas=(0.95, 0.99), weight_decay=0.01),
    clip_grad=dict(max_norm=35, norm_type=2)) # Gradient clip option. Set None to_
    ↪disable gradient clip. Find usage in https://mengine.readthedocs.io/en/latest/
    ↪tutorials/optim_wrapper.html

```

param_scheduler is a field that configures methods of adjusting optimization hyperparameters such as learning rate and momentum. Users can combine multiple schedulers to create a desired parameter adjustment strategy. Find more in [parameter scheduler tutorial](#) and [parameter scheduler API documents](#).


```

param_scheduler = [
    dict(
        type='CosineAnnealingLR',
        T_max=32,
        eta_min=0.01,
        begin=0,
        end=32,
        by_epoch=True,
        convert_to_iter_based=True),
    dict(
        type='CosineAnnealingLR',
        T_max=48,
        eta_min=1.0000000000000001e-07,
        begin=32,
        end=80,
        by_epoch=True,
        convert_to_iter_based=True),
    dict(
        type='CosineAnnealingMomentum',
        T_max=32,
        eta_min=0.8947368421052632,
        begin=0,
        end=32,
        by_epoch=True,
        convert_to_iter_based=True),
    dict(
        type='CosineAnnealingMomentum',
        T_max=48,
        eta_min=1,
        begin=32,
        end=80,
        by_epoch=True,
        convert_to_iter_based=True),
]

```

4.1.5 Hook config

Users can attach Hooks to training, validation, and testing loops to insert some operations during running. There are two different hook fields, one is `default_hooks` and the other is `custom_hooks`.

`default_hooks` is a dict of hook configs, and they are the hooks must be required at the runtime. They have default priority which should not be modified. If not set, runner will use the default values. To disable a default hook, users can set its config to `None`.

```

default_hooks = dict(
    timer=dict(type='IterTimerHook'),
    logger=dict(type='LoggerHook', interval=50),
    param_scheduler=dict(type='ParamSchedulerHook'),
    checkpoint=dict(type='CheckpointHook', interval=-1),
    sampler_seed=dict(type='DistSamplerSeedHook'),
    visualization=dict(type='Det3DVisualizationHook'))

```

`custom_hooks` is a list of all other hook configs. Users can develop their own hooks and insert them in this field.

```
custom_hooks = []
```

4.1.6 Runtime config

```
default_scope = 'mmdet3d' # The default registry scope to find modules. Refer to https://
↳/mmengine.readthedocs.io/en/latest/advanced_tutorials/registry.html

env_cfg = dict(
    cudnn_benchmark=False, # Whether to enable cudnn benchmark
    mp_cfg=dict( # Multi-processing config
        mp_start_method='fork', # Use fork to start multi-processing threads. 'fork'
↳usually faster than 'spawn' but maybe unsafe. See discussion in https://github.com/
↳pytorch/pytorch/issues/1355
        opencv_num_threads=0), # Disable opencv multi-threads to avoid system being
↳overloaded
    dist_cfg=dict(backend='nccl')) # Distribution configs

vis_backends = [dict(type='LocalVisBackend')] # Visualization backends. Refer to https://
↳/mmengine.readthedocs.io/en/latest/advanced_tutorials/visualization.html
visualizer = dict(
    type='Det3DLocalVisualizer', vis_backends=vis_backends, name='visualizer')

log_processor = dict(
    type='LogProcessor', # Log processor to process runtime logs
    window_size=50, # Smooth interval of log values
    by_epoch=True) # Whether to format logs with epoch type. Should be consistent with
↳the train loop's type.

log_level = 'INFO' # The level of logging.
load_from = None # Load model checkpoint as a pre-trained model from a given path. This
↳will not resume training.
resume = False # Whether to resume from the checkpoint defined in `load_from`. If `load_
↳from` is None, it will resume the latest checkpoint in the `work_dir`.
```

4.2 Config file inheritance

There are 4 basic component types under `configs/_base_`, dataset, model, schedule, default_runtime. Many methods could be easily constructed with one of these models like SECOND, PointPillars, PartA2, VoteNet. The configs that are composed of 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 existing methods. For example, if some modification is made based on PointPillars, users may first inherit the basic PointPillars structure by specifying `_base_ = '../pointpillars/pointpillars_hv_fpn_sbn-all_8xb4-2x_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 [MMEEngine config tutorial](#) for detailed documentation.

By setting the `_base_` field, we can set which files the current configuration file inherits from.

When `_base_` is a string of a file path, it means inheriting the contents from one config file.

```
_base_ = './pointpillars_hv_secfpn_8xb6-160e_kitti-3d-3class.py'
```

When `_base_` is a list of multiple file paths, it means inheriting from multiple files.

```
_base_ = [
    './_base_/models/pointpillars_hv_secfpn_kitti.py',
    './_base_/datasets/kitti-3d-3class.py',
    './_base_/schedules/cyclic-40e.py', './_base_/default_runtime.py'
]
```

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.

4.2.1 Ignore some fields in the base configs

Sometimes, you may set `_delete_=True` to ignore some of the fields in base configs. You may refer to [MMEEngine config tutorial](#) for a simple illustration.

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

```
model = dict(
    type='MVXFasterRCNN',
    data_preprocessor=dict(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/pointpillars_hv_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.

4.2.2 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 a multi-scale strategy to train and test a PointPillars, `train_pipeline/test_pipeline` are intermediate variables we would like to modify.

```
_base_ = './nus-3d.py'
train_pipeline = [
    dict(
        type='LoadPointsFromFile',
        load_dim=5,
        use_dim=5,
        backend_args=backend_args),
    dict(
        type='LoadPointsFromMultiSweeps',
        sweeps_num=10,
        backend_args=backend_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='Pack3DDetInputs',
        keys=['points', 'gt_labels_3d', 'gt_bboxes_3d'])
]
test_pipeline = [
    dict(
        type='LoadPointsFromFile',
        load_dim=5,
        use_dim=5,
        backend_args=backend_args),
    dict(
        type='LoadPointsFromMultiSweeps',
        sweeps_num=10,
        backend_args=backend_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],
                scale_ratio_range=[1., 1.],
```

(continues on next page)

(continued from previous page)

```

        translation_std=[0, 0, 0]),
        dict(type='RandomFlip3D'),
        dict(
            type='PointsRangeFilter', point_cloud_range=point_cloud_range)
    ]),
    dict(type='Pack3DDetInputs', keys=['points'])
]
train_dataloader = dict(dataset=dict(pipeline=train_pipeline))
val_dataloader = dict(dataset=dict(pipeline=test_pipeline))
test_dataloader = dict(dataset=dict(pipeline=test_pipeline))

```

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

4.2.3 Reuse variables in `_base_` file

If the users want to reuse the variables in the base file, they can get a copy of the corresponding variable by using `{{_base_.xxx}}`. E.g:

```

_base_ = './pointpillars_hv_secfpn_8xb6-160e_kitti-3d-3class.py'

a = {{_base_.model}} # variable `a` is equal to the `model` defined in `_base_`

```

4.3 Modify config through script arguments

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

- Update config keys of dict chains

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

- Update keys inside a list of configs

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

- Update values of list/tuple

If the value to be updated is a list or a tuple. For example, the config file normally sets `model.data_preprocessor.mean=[123.675, 116.28, 103.53]`. If you want to change the mean values, you may specify `--cfg-options model.data_preprocessor.mean="[127,127,127]"`. Note that the quotation mark " is necessary to support list/tuple data types, and that **NO** white space is allowed inside the quotation marks in the specified value.

4.4 Config Name Style

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

```
{algorithm name}_{model component names [component1]_[component2]_[...] }_{training_
↪ settings}_{training dataset information}_{testing dataset information}.py
```

The file name is divided to five parts. All parts and components are connected with `_` and words of each part or component should be connected with `-`.

- `{algorithm name}`: The name of the algorithm. It can be a detector name such as `pointpillars`, `fcos3d`, etc.
- `{model component names}`: Names of the components used in the algorithm such as `voxel_encoder`, `backbone`, `neck`, etc. For example, `second_secfpn_head-dcn-circlegenms` means using SECOND's SparseEncoder, SECONDFPN and a detection head with DCN and circle NMS.
- `{training settings}`: Information of training settings such as batch size, augmentations, loss trick, scheduler, and epochs/iterations. For example: `8xb4-tta-cyclic-20e` means using 8-gpus x 4-samples-per-gpu, test time augmentation, cyclic annealing learning rate, and train 20 epochs. Some abbreviations:
 - `{gpu x batch_per_gpu}`: GPUs and samples per GPU. `bN` indicates `N` batch size per GPU. E.g. `4xb4` is the short term of 4-GPUs x 4-samples-per-GPU.
 - `{schedule}`: training schedule, options are `schedule-2x`, `schedule-3x`, `cyclic-20e`, etc. `schedule-2x` and `schedule-3x` mean 24 epochs and 36 epochs respectively. `cyclic-20e` means 20 epochs respectively.
- `{training dataset information}`: Training dataset names like `kitti-3d-3class`, `nus-3d`, `s3dis-seg`, `scannet-seg`, `waymoD5-3d-car`. Here `3d` means dataset used for 3D object detection, and `seg` means dataset used for point cloud segmentation.
- `{testing dataset information}` (optional): Testing dataset name for models trained on one dataset but tested on another. If not mentioned, it means the model was trained and tested on the same dataset type.

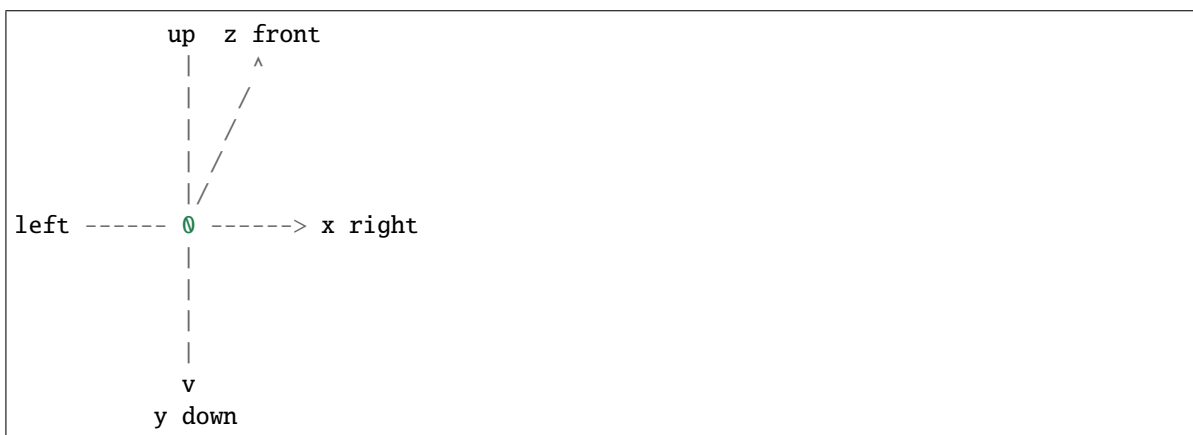
COORDINATE SYSTEM

5.1 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.

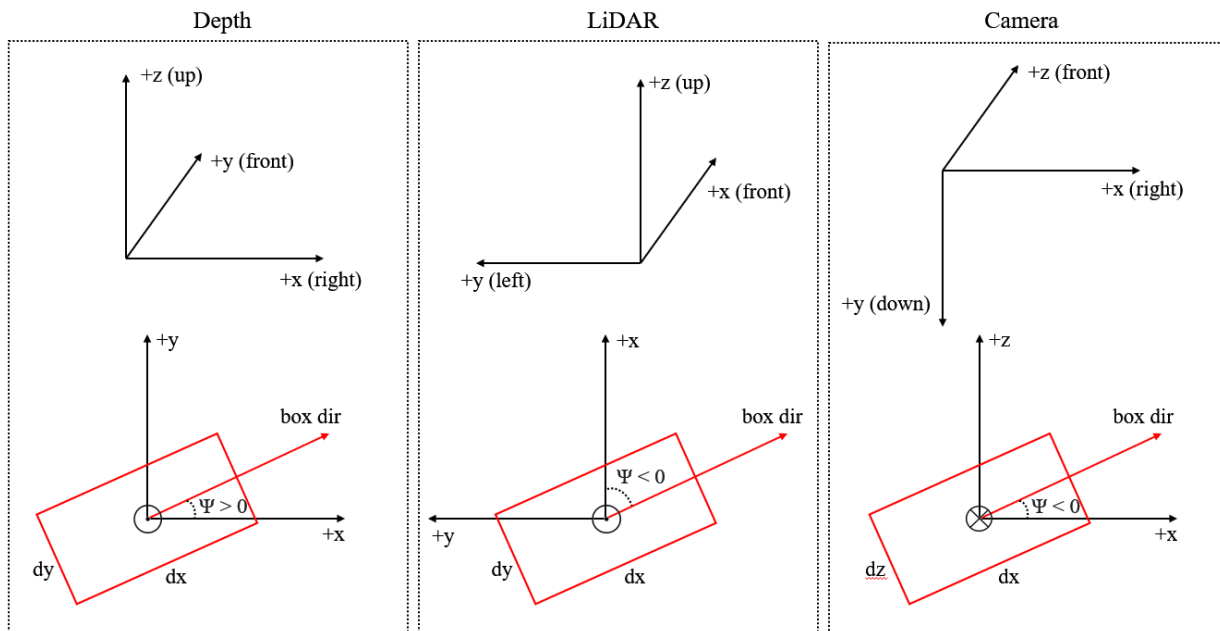


- Depth coordinate system – the coordinate system used by VoteNet, H3DNet, etc., in which the negative direction of the z-axis points to the ground, the positive direction of the x-axis points to the right, and the positive direction of the y-axis points to the front.



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.

5.2 Definition of the yaw angle

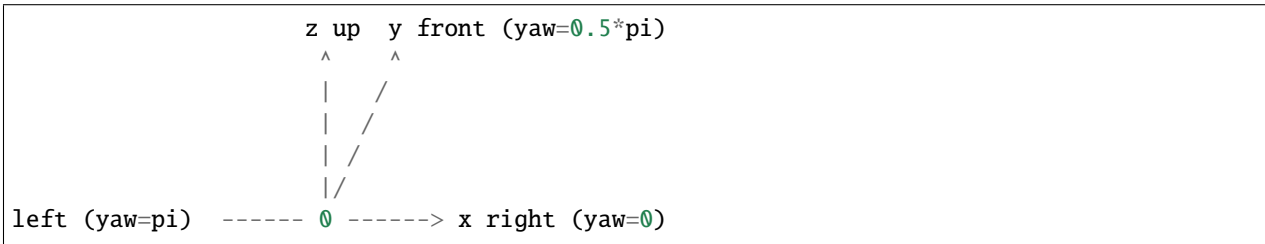
Please refer to [wikipedia](https://en.wikipedia.org/wiki/Yaw) for the standard definition of the yaw angle. In object detection, we choose an axis as the gravity axis, and a reference direction on the plane Π perpendicular to the gravity axis, then the reference direction has a yaw angle of 0, and other 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.

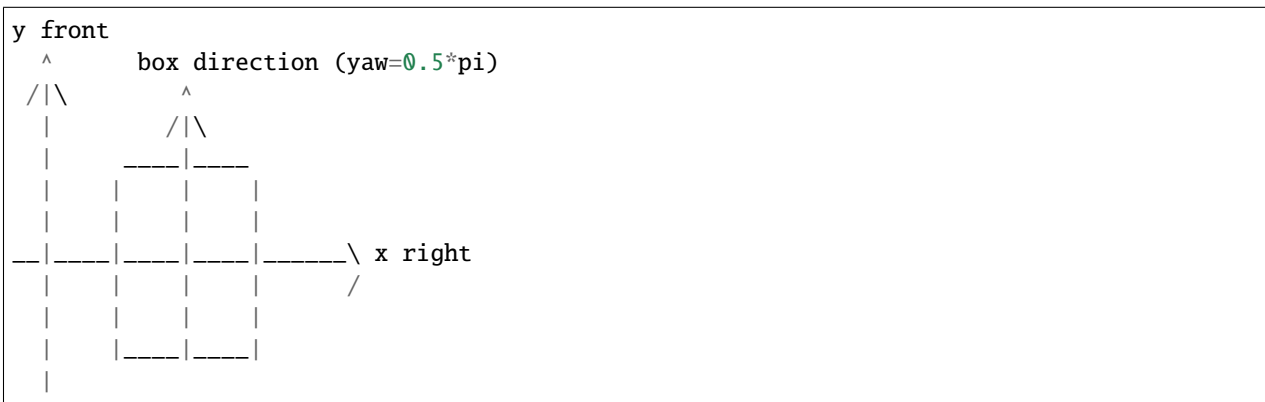
In MMDetection3D, all three coordinate systems are right-handed coordinate systems, which means the ascending direction of the yaw angle is counter-clockwise if viewed from the negative direction of the gravity axis (the axis is

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 $\frac{\pi}{2}$.



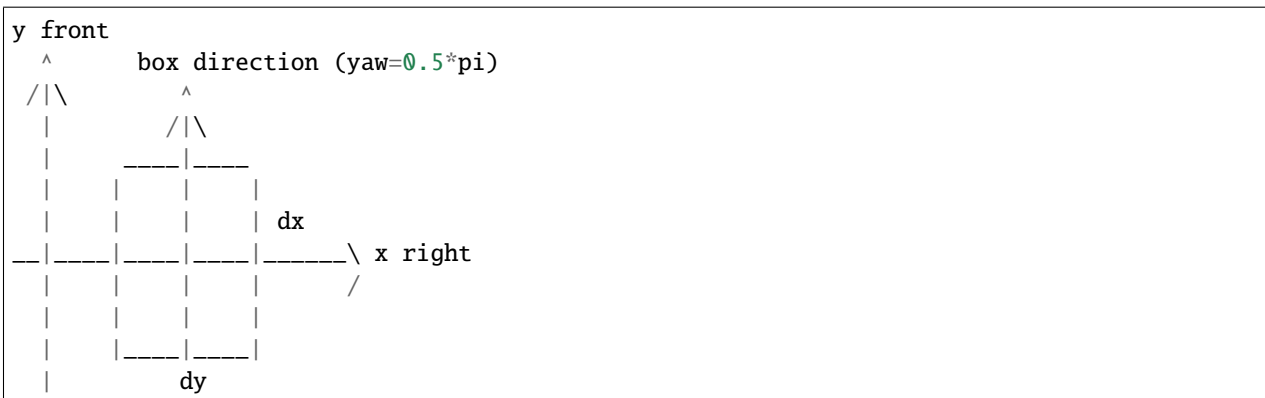
For a box, the value of its yaw angle equals its direction minus a reference direction. In all three coordinate systems in MMDetection3D, the reference direction is always the positive direction of the x-axis, while the direction of a box is defined to be parallel with the x-axis if its yaw angle is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



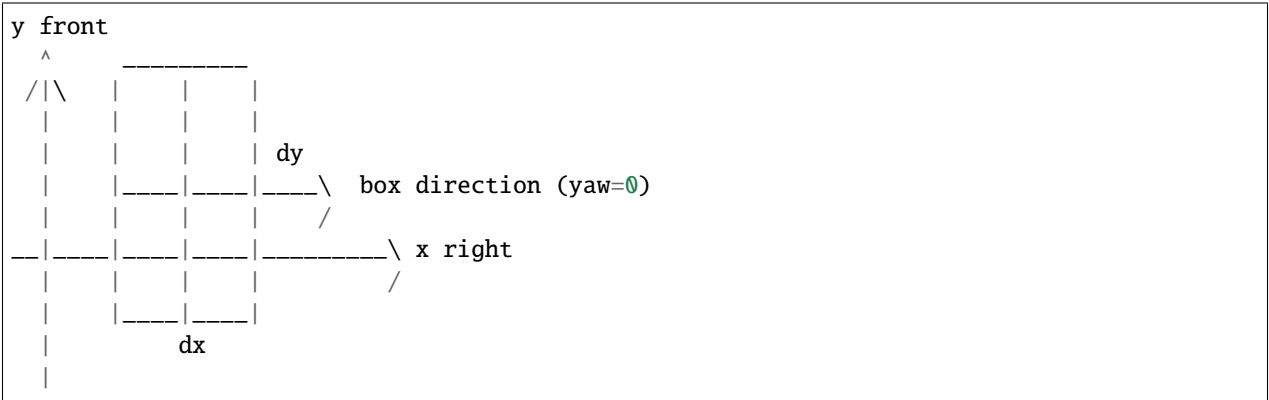
5.3 Definition of the box dimensions

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 .



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

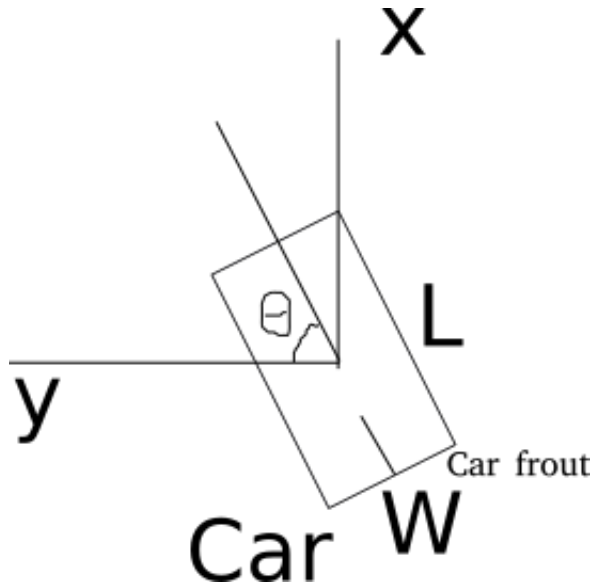


5.4 Relation with raw coordinate systems of supported datasets

5.4.1 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.

5.4.2 Waymo

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

5.4.3 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.

5.4.4 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.

5.4.5 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.

5.4.6 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 [sun-rgb-d_data_utils.py](#).

5.4.7 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.

5.5 Examples

5.5.1 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.

5.5.2 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.

5.5.3 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.

5.6 Common FAQ

5.6.1 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.

5.6.2 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.

5.6.3 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.

5.6.4 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.

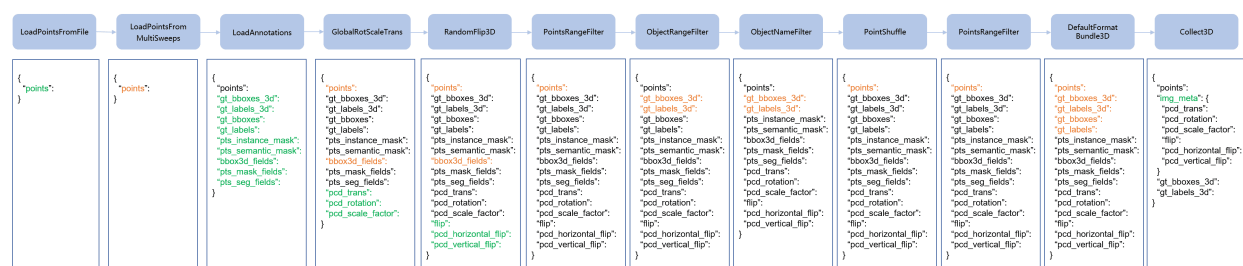
CUSTOMIZE DATA PIPELINES

6.1 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,
        backend_args=backend_args),
    dict(
        type='LoadPointsFromMultiSweeps',
        sweeps_num=10,
        backend_args=backend_args),
    dict(type='LoadAnnotations3D', with_bbox_3d=True, with_label_3d=True),
    dict(
        type='GlobalRotScaleTrans',
        rot_range=[-0.3925, 0.3925],
```

(continues on next page)

(continued from previous page)

```

        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,
        backend_args=backend_args),
    dict(
        type='LoadPointsFromMultiSweeps',
        sweeps_num=10,
        backend_args=backend_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.

6.1.1 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

6.1.2 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

6.1.3 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

6.1.4 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

6.2 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,
        backend_args=backend_args),
    dict(
        type='LoadPointsFromMultiSweeps',
        sweeps_num=10,
        backend_args=backend_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'])
]
```

DATASET PREPARATION

7.1 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

```

7.2 Download and Data Preparation

7.2.1 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 -0 ./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 -0 ./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 -0 ./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 -0 ./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
```

7.2.2 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.
- If you want faster evaluation on Waymo, you can download the preprocessed `metainfo` containing `contextname` and `timestamp` to the directory `data/waymo/waymo_format/`. Then, the dataset config is modified like the following:

```
val_evaluator = dict(
    type='WaymoMetric',
    ann_file='./data/waymo/kitti_format/waymo_infos_val.pkl',
    waymo_bin_file='./data/waymo/waymo_format/gt.bin',
    data_root='./data/waymo/waymo_format',
    backend_args=backend_args,
    convert_kitti_format=True,
    idx2metainfo='data/waymo/waymo_format/idx2metainfo.pkl'
)
```

Now, this trick is only used for LiDAR-based detection methods.

7.2.3 NuScenes

Download nuScenes V1.0 full dataset data [HERE](#). Prepare nusenes data by running:

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

7.2.4 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/dataset_converters/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](#) for more details.

7.2.5 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](#).

7.2.6 Customized Datasets

For using custom datasets, please refer to [Customize Datasets](#).

7.2.7 Update data infos

If you have used v1.0.0rc1-v1.0.0rc4 mmdetection3d to create data infos before, and now you want to use the newest v1.1.0 mmdetection3d, you need to update the data infos file.

```
python tools/dataset_converters/update_infos_to_v2.py --dataset ${DATA_SET} --pkl-path $  
↪${PKL_PATH} --out-dir ${OUT_DIR}
```

- `--dataset` : Name of dataset.
- `--pkl-path` : Specify the data infos pkl file path.
- `--out-dir` : Output direction of the data infos pkl file.

Example:

```
python tools/dataset_converters/update_infos_to_v2.py --dataset kitti --pkl-path ./data/  
↪kitti/kitti_infos_trainval.pkl --out-dir ./data/kitti
```

INFERENCE

8.1 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.

8.2 Testing

8.2.1 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/000008.bin configs/second/second_hv-secfpn_8xb6-
↪80e_kitti-3d-car.py checkpoints/second_hv-secfpn_8xb6-80e_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_
↪8xb16_sunrgbd-3d.py checkpoints/votenet_8xb16_sunrgbd-3d_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/000008.bin demo/data/kitti/000008.png_
→demo/data/kitti/000008.pkl configs/mvxnet/mvx_fpn-dv-second-secfpn_8xb2-80e_kitti-3d-
→3class.py checkpoints/mvx_fpn-dv-second-secfpn_8xb2-80e_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_8xb16_sunrgbd.py checkpoints/imvotenet_stage2_8xb16_sunrgbd_
→20210323_184021-d44dcb66.pth
```

8.2.2 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}] [--cam-type ${CAM_TYPE}] [--score-thr ${SCORE-
→THR}] [--out-dir ${OUT_DIR}] [--show]
```

where the ANNOTATION_FILE should provide the 3D to 2D projection matrix (camera intrinsic matrix), and CAM_TYPE should be specified according to dataset. For example, if you want to inference on the front camera image, the CAM_TYPE should be set as CAM_2 for KITTI, and CAM_FRONT for nuScenes. By specifying CAM_TYPE, you can even infer on any camera images for datasets with multi-view cameras, such as nuScenes and Waymo. SCORE_THR is the 3D bbox threshold while visualization. The visualization results including an image and its predicted 3D bounding boxes projected on the image will be saved in \${OUT_DIR}/IMG_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.pkl configs/fcos3d/fcos3d_r101-caffe-dcn-fpn-head-gn_8xb2-1x_nus-
→mono3d_finetune.py checkpoints/fcos3d_r101-caffe-dcn-fpn-head-gn_8xb2-1x_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](#).

8.2.3 3D Segmentation

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

```
python demo/pcd_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_2xb16-cosine-200e_scannet-seg.py checkpoints/pointnet2_ssg_2xb16-cosine-
↪ 200e_scannet-seg_20210514_143644-ee73704a.pth
```


MODEL DEPLOYMENT

MMDet3D 1.1 fully relies on [MMDeploy](#) to deploy models. Please stay tuned and this document will be update soon.

INFERENCE AND TRAIN WITH EXISTING MODELS AND STANDARD DATASETS

10.1 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](#).

10.1.1 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} ${CHECKPOINT_FILE} [--cfg-options test_evaluator.
↪pklfile_prefix=${RESULT_FILE}] [--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} ${CHECKPOINT_FILE} [--cfg-options test_evaluator.
↪pklfile_prefix=${RESULT_FILE}] [--show] [--show-dir ${SHOW_DIR}]

# multi-gpu testing
./tools/dist_test.sh ${CONFIG_FILE} ${CHECKPOINT_FILE} ${GPU_NUM} [--cfg-options test_
↪evaluator.pklfile_prefix=${RESULT_FILE}] [--show] [--show-dir ${SHOW_DIR}]
```

Note:

For now, CPU testing is only supported for SMOKE.

Optional arguments:

- `--show`: If specified, detection results will be plotted in the silent 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.

All evaluation related arguments are set in the `test_evaluator` in corresponding dataset configuration. such as `test_evaluator = dict(type='KittiMetric', ann_file=data_root + 'kitti_infos_val.pkl', pklfile_prefix=None, submission_prefix=None)`

The arguments:

- `type`: The name of the corresponding metric, usually associated with the dataset.
- `ann_file`: The path of annotation file.
- `pklfile_prefix`: An optional argument. The filename of the output results in pickle format. If not specified, the results will not be saved to a file.
- `submission_prefix`: An optional argument. The results will be saved to a file then you can upload it to do the official evaluation.

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_8xb8_scannet-3d.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_8xb8_scannet-3d.py \
    checkpoints/votenet_8x8_scannet-3d-18class_20200620_230238-2cea9c3a.pth \
    --show --show-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_8xb8_scannet-3d.py \
    checkpoints/votenet_8x8_scannet-3d-18class_20200620_230238-2cea9c3a.pth
```

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

```
./tools/slurm_test.sh ${PARTITION} ${JOB_NAME} configs/second/second_hv_secfn_8xb6-
↪80e_kitti-3d-3class.py \
    checkpoints/hv_second_secfn_6x8_80e_kitti-3d-3class_20200620_230238-9208083a.
↪pth
```

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/pointpillars_hv_
↪secfn_sbn-all_8xb4-2x_nus-3d.py \
    checkpoints/hv_pointpillars_fn_sbn-all_4x8_2x_nus-3d_20200620_230405-2fa62f3d.
↪pth \
    --cfg-options 'test_evaluator.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/second_hv_secfpn_8xb6-
↪80e_kitti-3d-3class.py \
    checkpoints/hv_second_secfpn_6x8_80e_kitti-3d-3class_20200620_230238-9208083a.
↪pth \
    --cfg-options 'test_evaluator.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 \
    --cfg-options 'test_evaluator.jsonfile_prefix=results/pp_lyft/results_challenge
↪' \
    'test_evaluator.csv_savepath=results/pp_lyft/results_challenge.csv' \
    'test_evaluator.pklfile_prefix=results/pp_lyft/results_challenge.pkl'
```

Notice: To generate submissions on Lyft, `csv_savepath` must be given in the `--cfg-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 `'lyft_infos_test.pkl'`, which is the official test set of Lyft without annotation. To test on the validation set, please change this to `'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/pointpillars_hv_
↪secfpn_sbn-all_16xb2-2x_waymo-3d-car.py \
    checkpoints/hv_pointpillars_secfpn_sbn-2x16_2x_waymo-3d-car_latest.pth \
    --cfg-options 'test_evaluator.pklfile_prefix=results/waymo-car/kitti_results' \
    'test_evaluator.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/pointpillars_hv_
↪secfpn_sbn-all_16xb2-2x_waymo-3d-car.py \
    checkpoints/hv_pointpillars_secfpn_sbn-2x16_2x_waymo-3d-car_latest.pth \
    --cfg-options 'test_evaluator.pklfile_prefix=results/waymo-car/kitti_results' \
    'test_evaluator.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.

10.2 Train predefined models on standard datasets

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

```
train_cfg = dict(type='EpochBasedTrainLoop', val_interval=1) # 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. '2xb8' 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.

10.2.1 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}`.

10.2.2 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.

10.2.3 Train with multiple GPUs

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

Optional arguments are:

- `--cfg-options 'Key=value'`: Override some settings in the used config.

10.2.4 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 configs/pointpillars/pointpillars_hv_
↪secfpn_8xb6-160e_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.

10.2.5 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 `--cfg-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} --cfg-options 'env_cfg.dist_cfg.port=29500'
CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 ./tools/slurm_train.sh ${PARTITION} ${JOB_NAME} ↪
↪config2.py ${WORK_DIR} --cfg-options 'env_cfg.dist_cfg.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`,

```
env_cfg = dict(
    dist_cfg=dict(backend='nccl', port=29500)
)
```

In `config2.py`,

```
env_cfg = dict(  
    dist_cfg=dict(backend='nccl', port=29501)  
)
```

Then you can launch two jobs with config1.py and config2.py.

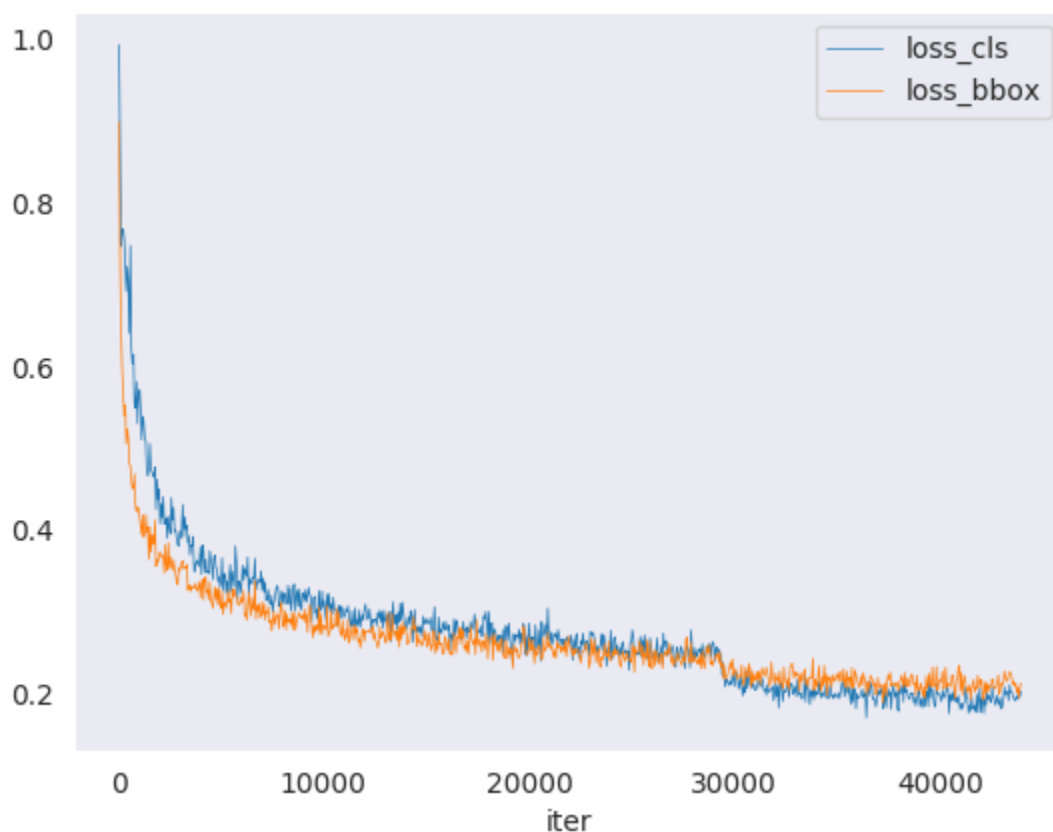
```
CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 ./tools/slurm_train.sh ${PARTITION} ${JOB_NAME}   
↪ config1.py ${WORK_DIR}  
CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 ./tools/slurm_train.sh ${PARTITION} ${JOB_NAME}   
↪ config2.py ${WORK_DIR}
```

USEFUL TOOLS

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

11.1 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
```

11.2 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:

11.2.1 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.

11.2.2 2. Build mmdet3d-serve docker image

```
docker build -t mmdet3d-serve:latest docker/serve/
```

11.2.3 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

11.2.4 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
```

11.3 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 `mmdcv.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.

11.4 Model Conversion

11.4.1 RegNet model to MMDetection

`tools/model_converters/regnet2mmdet.py` convert keys in pyccls pretrained RegNet models to MMDetection style.

```
python tools/model_converters/regnet2mmdet.py ${SRC} ${DST} [-h]
```

11.4.2 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]
```

11.4.3 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`.

11.5 Dataset Conversion

`tools/dataset_converters/` contains tools for converting datasets to other formats. Most of them convert datasets to pickle based info files, like kitti, nusenes 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/dataset_converters/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.

11.6 Miscellaneous

11.6.1 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...]]
```


VISUALIZATION

MMDetection3D provides a `Det3DLocalVisualizer` to visualize and store the state of the model during training and testing, as well as results, with the following features.

1. Support the basic drawing interface for multi-modality data and multi-task.
2. Support multiple backends such as local, TensorBoard, to write training status such as loss, lr, or performance evaluation metrics and to a specified single or multiple backends.
3. Support ground truth visualization on multimodal data, and cross-modal visualization of 3D detection results.

12.1 Basic Drawing Interface

Inherited from `DetLocalVisualizer`, `Det3DLocalVisualizer` provides an interface for drawing common objects on 2D images, such as drawing detection boxes, points, text, lines, circles, polygons, and binary masks. More details about 2D drawing can refer to the [visualization documentation](#) in MMDetection. Here we introduce the 3D drawing interface:

12.1.1 Drawing point cloud on the image

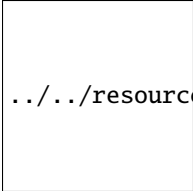
We support drawing point cloud on the image by using `draw_points_on_image`.

```
import mmcv
import numpy as np
from mmengine import load

from mmdet3d.visualization import Det3DLocalVisualizer

info_file = load('demo/data/kitti/000008.pkl')
points = np.fromfile('demo/data/kitti/000008.bin', dtype=np.float32)
points = points.reshape(-1, 4)[:,:3]
lidar2img = np.array(info_file['data_list'][0]['images']['CAM2']['lidar2img'], dtype=np.
    ↪float32)

visualizer = Det3DLocalVisualizer()
img = mmcv.imread('demo/data/kitti/000008.png')
img = mmcv.imconvert(img, 'bgr', 'rgb')
visualizer.set_image(img)
visualizer.draw_points_on_image(points, lidar2img)
visualizer.show()
```



../../resources/points_on_image.png

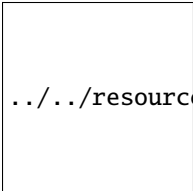
12.1.2 Drawing 3D Boxes on Point Cloud

We support drawing 3D boxes on point cloud by using `draw_bboxes_3d`.

```
import torch

from mmdet3d.visualization import Det3DLocalVisualizer
from mmdet3d.structures import LiDARInstance3DBoxes

points = np.fromfile('tests/data/kitti/training/velodyne/0000000.bin', dtype=np.float32)
points = points.reshape(-1, 4)
visualizer = Det3DLocalVisualizer()
# set point cloud in visualizer
visualizer.set_points(points)
bboxes_3d = LiDARInstance3DBoxes(torch.tensor(
    [[8.7314, -1.8559, -1.5997, 1.2000, 0.4800, 1.8900,
      -1.5808]])),
# Draw 3D bboxes
visualizer.draw_bboxes_3d(bboxes_3d)
visualizer.show()
```



../../resources/pcd.png

12.1.3 Drawing Projected 3D Boxes on Image

We support drawing projected 3D boxes on image by using `draw_proj_bboxes_3d`.

```
import mmcv
import numpy as np
from mmengine import load

from mmdet3d.visualization import Det3DLocalVisualizer
from mmdet3d.structures import CameraInstance3DBoxes

info_file = load('demo/data/kitti/0000008.pkl')
cam2img = np.array(info_file['data_list'][0]['images']['CAM2']['cam2img'], dtype=np.
    ↪float32)
bboxes_3d = []
for instance in info_file['data_list'][0]['instances']:
    bboxes_3d.append(instance['bbox_3d'])
```

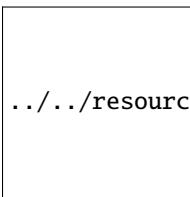
(continues on next page)

(continued from previous page)

```
gt_bboxes_3d = np.array(bboxes_3d, dtype=np.float32)
gt_bboxes_3d = CameraInstance3DBoxes(gt_bboxes_3d)
input_meta = {'cam2img': cam2img}

visualizer = Det3DLocalVisualizer()

img = mmcv.imread('demo/data/kitti/000008.png')
img = mmcv.imconvert(img, 'bgr', 'rgb')
visualizer.set_image(img)
# project 3D bboxes to image
visualizer.draw_proj_bboxes_3d(gt_bboxes_3d, input_meta)
visualizer.show()
```



../../resources/mono3d.png

12.1.4 Drawing BEV Boxes

We support drawing BEV boxes by using `draw_bev_bboxes`.

```
import numpy as np
from mmengine import load

from mmdet3d.visualization import Det3DLocalVisualizer
from mmdet3d.structures import CameraInstance3DBoxes

info_file = load('demo/data/kitti/000008.pkl')
bboxes_3d = []
for instance in info_file['data_list'][0]['instances']:
    bboxes_3d.append(instance['bbox_3d'])
gt_bboxes_3d = np.array(bboxes_3d, dtype=np.float32)
gt_bboxes_3d = CameraInstance3DBoxes(gt_bboxes_3d)

visualizer = Det3DLocalVisualizer()
# set bev image in visualizer
visualizer.set_bev_image()
# draw bev bboxes
visualizer.draw_bev_bboxes(gt_bboxes_3d, edge_colors='orange')
visualizer.show()
```

12.1.5 Drawing 3D Semantic Mask

We support draw segmentation mask via per-point colorization by using `draw_seg_mask`.

```
import torch

from mmdet3d.visualization import Det3DLocalVisualizer

points = np.fromfile('tests/data/s3dis/points/Area_1_office_2.bin', dtype=np.float32)
points = points.reshape(-1, 3)
visualizer = Det3DLocalVisualizer()
mask = np.random.rand(points.shape[0], 3)
points_with_mask = np.concatenate((points, mask), axis=-1)
# Draw 3D points with mask
visualizer.draw_seg_mask(points_with_mask)
visualizer.show()
```

12.2 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 will be saved in `${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. `***_gt.png` and `***_pred.png` in multi-modality detection task and vision-based 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 download the `results.pkl` from the remote server, and visualize the prediction results offline in your local machine.

To visualize the results with `Open3D` backend offline, you can run the following command:

```
python tools/misc/visualize_results.py ${CONFIG_FILE} --result ${RESULTS_PATH} --show-
↳dir ${SHOW_DIR}
```

This allows the inference and results generation to be done in remote server and the users can open them on their host with GUI.

12.3 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}
```

Notice: Once specifying `--output-dir`, the images of views specified by users will be saved when pressing `_ESC_` in open3d window.

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_
↳ lidar_det --aug --output-dir ${OUTPUT_DIR}
```

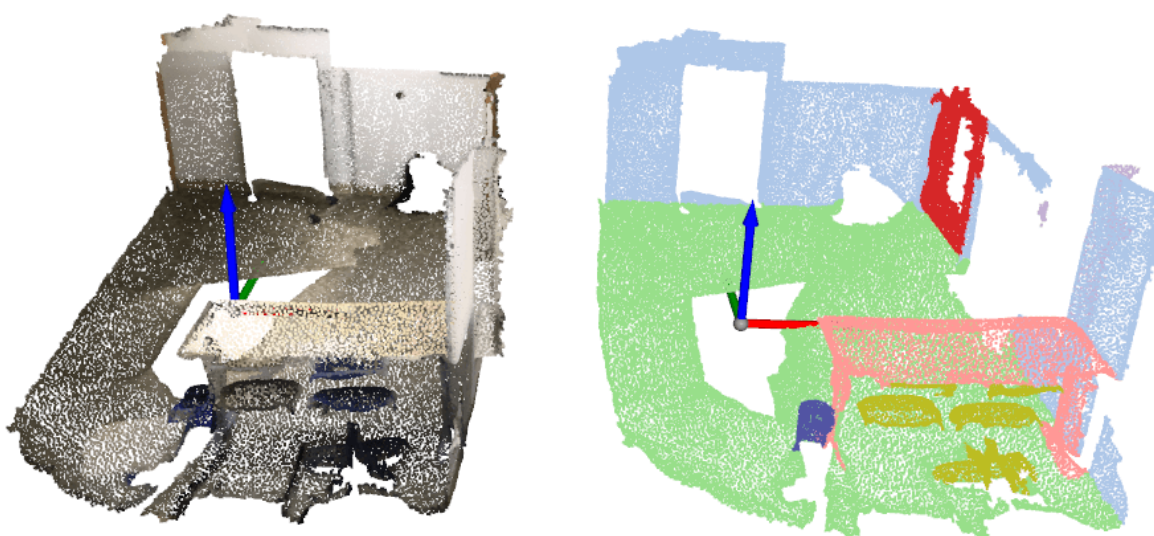
If you also want to show 2D images with 3D bounding boxes projected onto them, you need to find a config that supports multi-modality data loading, and then change the `--task` args to `multi-modality_det`. An example is showed below:

```
python tools/misc/browse_dataset.py configs/mvxnet/dv_mvxfpn_second_secfpn_adamw_2x8_
↳ 80e_kitti-3d-3class.py --task multi-modality_det --output-dir ${OUTPUT_DIR}
```



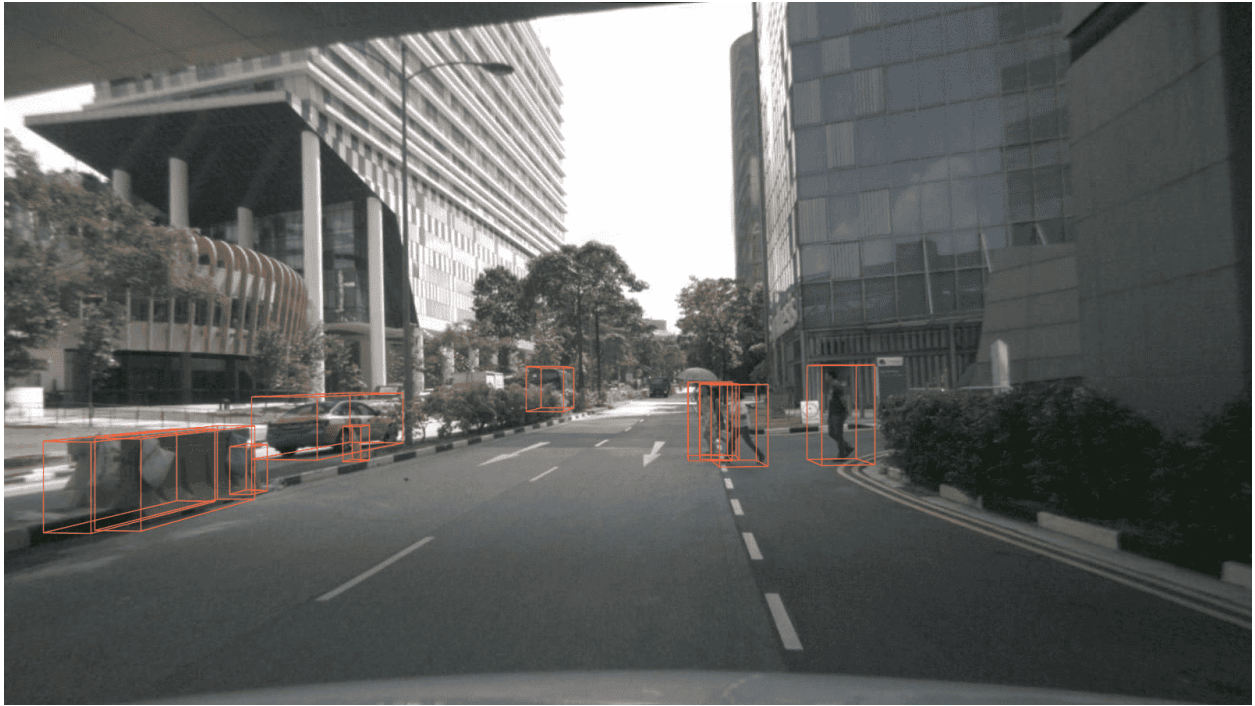
You can simply browse different datasets using different configs, e.g. visualizing the ScanNet dataset in 3D semantic segmentation task:

```
python tools/misc/browse_dataset.py configs/_base_/datasets/scannet_seg-3d-20class.py --
↳ task lidar_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
```



DATASETS

13.1 KITTI Dataset for 3D Object Detection

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

13.1.1 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)
```

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 -0 ./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 -0 ./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 -0 ./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 -0 ./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
│   └── xxxx.bin
├── kitti_infos_train.pkl
├── kitti_infos_val.pkl
└── kitti_dbinfos_train.pkl
```

(continues on next page)

(continued from previous page)

```

├─ kitti_infos_test.pkl
├─ kitti_infos_trainval.pkl

```

- `kitti_gt_database/xxxxx.bin`: point cloud data included in each 3D bounding box of the training dataset.
- `kitti_infos_train.pkl`: training dataset, a dict contains two keys: `metainfo` and `data_list`. `metainfo` contains the basic information for the dataset itself, such as `categories`, `dataset` and `info_version`, while `data_list` is a list of dict, each dict (hereinafter referred to as `info`) contains all the detailed information of single sample as follows:
 - `info['sample_idx']`: The index of this sample in the whole dataset.
 - `info['images']`: Information of images captured by multiple cameras. A dict contains five keys including: `CAM0`, `CAM1`, `CAM2`, `CAM3`, `R0_rect`.
 - * `info['images']['R0_rect']`: Rectifying rotation matrix with shape (4, 4).
 - * `info['images']['CAM2']`: Include some information about the CAM2 camera sensor.
 - `info['images']['CAM2']['img_path']`: The filename of the image.
 - `info['images']['CAM2']['height']`: The height of the image.
 - `info['images']['CAM2']['width']`: The width of the image.
 - `info['images']['CAM2']['cam2img']`: Transformation matrix from camera to image with shape (4, 4).
 - `info['images']['CAM2']['lidar2cam']`: Transformation matrix from lidar to camera with shape (4, 4).
 - `info['images']['CAM2']['lidar2img']`: Transformation matrix from lidar to image with shape (4, 4).
 - `info['lidar_points']`: A dict containing all the information related to the lidar points.
 - * `info['lidar_points']['lidar_path']`: The filename of the lidar point cloud data.
 - * `info['lidar_points']['num_pts_feats']`: The feature dimension of point.
 - * `info['lidar_points']['Tr_velo_to_cam']`: Transformation from Velodyne coordinate to camera coordinate with shape (4, 4).
 - * `info['lidar_points']['Tr_imu_to_velo']`: Transformation from IMU coordinate to Velodyne coordinate with shape (4, 4).
 - `info['instances']`: It is a list of dict. Each dict contains all annotation information of single instance. For the *i*-th instance:
 - * `info['instances'][i]['bbox']`: List of 4 numbers representing the 2D bounding box of the instance, in (x1, y1, x2, y2) order.
 - * `info['instances'][i]['bbox_3d']`: List of 7 numbers representing the 3D bounding box of the instance, in (x, y, z, l, h, w, yaw) order.
 - * `info['instances'][i]['bbox_label']`: An int indicate the 2D label of instance and the -1 indicating ignore.
 - * `info['instances'][i]['bbox_label_3d']`: An int indicate the 3D label of instance and the -1 indicating ignore.
 - * `info['instances'][i]['depth']`: Projected center depth of the 3D bounding box with respect to the image plane.
 - * `info['instances'][i]['num_lidar_pts']`: The number of LiDAR points in the 3D bounding box.

- * info['instances'][i]['center_2d']: Projected 2D center of the 3D bounding box.
- * info['instances'][i]['difficulty']: KITTI difficulty: 'Easy', 'Moderate', 'Hard'.
- * info['instances'][i]['truncated']: Float from 0 (non-truncated) to 1 (truncated), where truncated refers to the object leaving image boundaries.
- * info['instances'][i]['occluded']: Integer (0,1,2,3) indicating occlusion state: 0 = fully visible, 1 = partly occluded, 2 = largely occluded, 3 = unknown.
- * info['instances'][i]['group_ids']: Used for multi-part object.
- info['plane'](optional): Road level information.

Please refer to [kitti_converter.py](#) and [update_infos_to_v2.py](#) for more details.

13.1.2 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  
        use_dim=4),  
    dict(type='LoadAnnotations3D', with_bbox_3d=True, with_label_3d=True),  
    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='Pack3DDetInputs',  
        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.

13.1.3 Evaluation

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

```
bash tools/dist_test.sh configs/pointpillars/pointpillars_hv_secfn_8xb6-160e_kitti-3d-
↪3class.py work_dirs/pointpillars_hv_secfn_8xb6-160e_kitti-3d-3class/latest.pth 8
```

13.1.4 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
```

13.1.5 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:

- First, you need to modify the `test_dataloader` and `test_evaluator` dict in your config file, just like:

```
data_root = 'data/kitti/'
test_dataloader = dict(
    dataset=dict(
        ann_file='kitti_infos_test.pkl',
        load_eval_anns=False,
        data_prefix=dict(pts='testing/velodyne_reduced'))))
test_evaluator = dict(
    ann_file=data_root + 'kitti_infos_test.pkl',
    format_only=True,
    pklfile_prefix='results/kitti-3class/kitti_results',
    submission_prefix='results/kitti-3class/kitti_results')
```

- And then, you can run the test script.

```
./tools/dist_test.sh configs/pointpillars/pointpillars_hv_secfn_8xb6-160e_kitti-3d-
↪3class.py work_dirs/pointpillars_hv_secfn_8xb6-160e_kitti-3d-3class/latest.pth 8
```

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.

13.2 NuScenes Dataset for 3D Object Detection

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

13.2.1 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
│   └── nuscenesc
│       ├── maps
│       ├── samples
│       ├── sweeps
│       ├── v1.0-test
│       └── v1.0-trainval
```

13.2.2 Dataset Preparation

We typically need to organize the useful data information with a .pkl file in a specific style. To prepare these files for nuScenes, run the following command:

```
python tools/create_data.py nuscenesc --root-path ./data/nuscenesc --out-dir ./data/
└─>nuscenesc --extra-tag nuscenesc
```

The folder structure after processing should be as below.

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
│   └── nuscenesc
│       ├── maps
│       ├── samples
│       ├── sweeps
│       ├── v1.0-test
│       ├── v1.0-trainval
│       ├── nuscenesc_database
│       ├── nuscenesc_infos_train.pkl
│       ├── nuscenesc_infos_val.pkl
│       ├── nuscenesc_infos_test.pkl
│       └── nuscenesc_dbinfos_train.pkl
```

- nuscenesc_database/xxxxx.bin: point cloud data included in each 3D bounding box of the training dataset

- `nuscenes_infos_train.pkl`: training dataset, a dict contains two keys: `metainfo` and `data_list`. `metainfo` contains the basic information for the dataset itself, such as `categories`, `dataset` and `info_version`, while `data_list` is a list of dict, each dict (hereinafter referred to as `info`) contains all the detailed information of single sample as follows:
 - `info['sample_idx']`: The index of this sample in the whole dataset.
 - `info['token']`: Sample data token.
 - `info['timestamp']`: Timestamp of the sample data.
 - `info['lidar_points']`: A dict containing all the information related to the lidar points.
 - * `info['lidar_points']['lidar_path']`: The filename of the lidar point cloud data.
 - * `info['lidar_points']['num_pts_feats']`: The feature dimension of point.
 - * `info['lidar_points']['lidar2ego']`: The transformation matrix from this lidar sensor to ego vehicle. (4x4 list)
 - * `info['lidar_points']['ego2global']`: The transformation matrix from the ego vehicle to global coordinates. (4x4 list)
 - `info['lidar_sweeps']`: A list contains sweeps information (The intermediate lidar frames without annotations)
 - * `info['lidar_sweeps'][i]['lidar_points']['data_path']`: The lidar data path of i-th sweep.
 - * `info['lidar_sweeps'][i]['lidar_points']['lidar2ego']`: The transformation matrix from this lidar sensor to ego vehicle. (4x4 list)
 - * `info['lidar_sweeps'][i]['lidar_points']['ego2global']`: The transformation matrix from the ego vehicle to global coordinates. (4x4 list)
 - * `info['lidar_sweeps'][i]['lidar2sensor']`: The transformation matrix from the main lidar sensor to the current sensor (for collecting the sweep data). (4x4 list)
 - * `info['lidar_sweeps'][i]['timestamp']`: Timestamp of the sweep data.
 - * `info['lidar_sweeps'][i]['sample_data_token']`: The sweep sample data token.
 - `info['images']`: A dict contains six keys corresponding to each camera: `'CAM_FRONT'`, `'CAM_FRONT_RIGHT'`, `'CAM_FRONT_LEFT'`, `'CAM_BACK'`, `'CAM_BACK_LEFT'`, `'CAM_BACK_RIGHT'`. Each dict contains all data information related to corresponding camera.
 - * `info['images']['CAM_XXX']['img_path']`: The filename of the image.
 - * `info['images']['CAM_XXX']['cam2img']`: The transformation matrix recording the intrinsic parameters when projecting 3D points to each image plane. (3x3 list)
 - * `info['images']['CAM_XXX']['sample_data_token']`: Sample data token of image.
 - * `info['images']['CAM_XXX']['timestamp']`: Timestamp of the image.
 - * `info['images']['CAM_XXX']['cam2ego']`: The transformation matrix from this camera sensor to ego vehicle. (4x4 list)
 - * `info['images']['CAM_XXX']['lidar2cam']`: The transformation matrix from lidar sensor to this camera. (4x4 list)
 - `info['instances']`: It is a list of dict. Each dict contains all annotation information of single instance. For the i-th instance:
 - * `info['instances'][i]['bbox_3d']`: List of 7 numbers representing the 3D bounding box of the instance, in (x, y, z, l, w, h, yaw) order.

- * info['instances'][i]['bbox_label_3d']: A int indicate the label of instance and the -1 indicate ignore.
- * info['instances'][i]['velocity']: Velocities of 3D bounding boxes (no vertical measurements due to inaccuracy), a list has shape (2,).
- * info['instances'][i]['num_lidar_pts']: Number of lidar points included in each 3D bounding box.
- * info['instances'][i]['num_radar_pts']: Number of radar points included in each 3D bounding box.
- * info['instances'][i]['bbox_3d_isvalid']: 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.
- info['cam_instances']: It is a dict containing keys 'CAM_FRONT', 'CAM_FRONT_RIGHT', 'CAM_FRONT_LEFT', 'CAM_BACK', 'CAM_BACK_LEFT', 'CAM_BACK_RIGHT'. For vision-based 3D object detection task, we split 3D annotations of the whole scenes according to the camera they belong to. For the i-th instance:
 - * info['cam_instances']['CAM_XXX'][i]['bbox_label']: Label of instance.
 - * info['cam_instances']['CAM_XXX'][i]['bbox_label_3d']: Label of instance.
 - * info['cam_instances']['CAM_XXX'][i]['bbox']: 2D bounding box annotation (exterior rectangle of the projected 3D box), a list arrange as [x1, y1, x2, y2].
 - * info['cam_instances']['CAM_XXX'][i]['center_2d']: Projected center location on the image, a list has shape (2,).
 - * info['cam_instances']['CAM_XXX'][i]['depth']: The depth of projected center.
 - * info['cam_instances']['CAM_XXX'][i]['velocity']: Velocities of 3D bounding boxes (no vertical measurements due to inaccuracy), a list has shape (2,).
 - * info['cam_instances']['CAM_XXX'][i]['attr_label']: The attr label of instance. We maintain a default attribute collection and mapping for attribute classification.
 - * info['cam_instances']['CAM_XXX'][i]['bbox_3d']: List of 7 numbers representing the 3D bounding box of the instance, in (x, y, z, l, h, w, yaw) order.

Note:

1. The differences between `bbox_3d` in `instances` and that in `cam_instances`. Both `bbox_3d` have been converted to MMDet3D coordinate system, but `bboxes_3d` in `instances` is in LiDAR coordinate format and `bboxes_3d` in `cam_instances` is in Camera coordinate format. Mind the difference between them in 3D Box representation ('l, w, h' and 'l, h, w').
2. Here we only explain the data recorded in the training info files. The same applies to validation and testing set (the .pkl file of test set does not contains `instances` and `cam_instances`).

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

13.2.3 Training pipeline

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',
```

(continues on next page)

(continued from previous page)

```

        load_dim=5,
        use_dim=5),
    dict(
        type='LoadPointsFromMultiSweeps',
        sweeps_num=10),
    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='Pack3DDetInputs',
        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.

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='mmdet.Resize', scale=(1600, 900), keep_ratio=True),
    dict(type='RandomFlip3D', flip_ratio_bev_horizontal=0.5),
    dict(
        type='Pack3DDetInputs',
        keys=[
            'img', 'gt_bboxes', 'gt_bboxes_labels', 'attr_labels', 'gt_bboxes_3d',
            'gt_labels_3d', 'centers_2d', '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.

13.2.4 Evaluation

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

```
bash ./tools/dist_test.sh configs/pointpillars/pointpillars_hv_fpn_sbn-all_8xb4-2x_nus-
↪3d.py checkpoints/hv_pointpillars_fpn_sbn-all_4x8_2x_nus-3d_20200620_230405-2fa62f3d.
↪pth 8
```

13.2.5 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
```


13.2.6 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.

You should modify the `jsonfile_prefix` in the `test_evaluator` of corresponding configuration. For example, adding `test_evaluator = dict(type='NuScenesMetric', jsonfile_prefix='work_dirs/pp-nus/results_eval.json')` or using `--cfg-options "test_evaluator.jsonfile_prefix=work_dirs/pp-nus/results_eval.json"` after the test command.

```
./tools/dist_test.sh configs/pointpillars/pointpillars_hv_fpn_sbn-all_8xb4-2x_nus-3d.py \
  --work_dirs/pp-nus/latest.pth 8 --cfg-options 'test_evaluator.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.

13.2.7 Notes

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.

13.3 Lyft Dataset for 3D Object Detection

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

13.3.1 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
```

(continues on next page)

(continued from previous page)

```

— 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.

13.3.2 Dataset Preparation

The way to organize Lyft dataset is similar to nuScenes. We also generate the `.pkl` 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/dataset_converters/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)

```

(continues on next page)

(continued from previous page)

```

|— 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.pkl`: training dataset, a dict contains two keys: `metainfo` and `data_list`. `metainfo` contains the basic information for the dataset itself, such as `categories`, `dataset` and `info_version`, while `data_list` is a list of dict, each dict (hereinafter referred to as `info`) contains all the detailed information of single sample as follows:
 - `info['sample_idx']`: The index of this sample in the whole dataset.
 - `info['token']`: Sample data token.
 - `info['timestamp']`: Timestamp of the sample data.
 - `info['lidar_points']`: A dict containing all the information related to the lidar points.
 - * `info['lidar_points']['lidar_path']`: The filename of the lidar point cloud data.
 - * `info['lidar_points']['num_pts_feats']`: The feature dimension of point.
 - * `info['lidar_points']['lidar2ego']`: The transformation matrix from this lidar sensor to ego vehicle. (4x4 list)
 - * `info['lidar_points']['ego2global']`: The transformation matrix from the ego vehicle to global coordinates. (4x4 list)
 - `info['lidar_sweeps']`: A list contains sweeps information (The intermediate lidar frames without annotations).
 - * `info['lidar_sweeps'][i]['lidar_points']['data_path']`: The lidar data path of i-th sweep.
 - * `info['lidar_sweeps'][i]['lidar_points']['lidar2ego']`: The transformation matrix from this lidar sensor to ego vehicle in i-th sweep timestamp
 - * `info['lidar_sweeps'][i]['lidar_points']['ego2global']`: The transformation matrix from the ego vehicle in i-th sweep timestamp to global coordinates. (4x4 list)
 - * `info['lidar_sweeps'][i]['lidar2sensor']`: The transformation matrix from the keyframe lidar to the i-th frame lidar. (4x4 list)
 - * `info['lidar_sweeps'][i]['timestamp']`: Timestamp of the sweep data.
 - * `info['lidar_sweeps'][i]['sample_data_token']`: The sweep sample data token.
 - `info['images']`: A dict contains six keys corresponding to each camera: `'CAM_FRONT'`, `'CAM_FRONT_RIGHT'`, `'CAM_FRONT_LEFT'`, `'CAM_BACK'`, `'CAM_BACK_LEFT'`, `'CAM_BACK_RIGHT'`. Each dict contains all data information related to corresponding camera.
 - * `info['images']['CAM_XXX']['img_path']`: The filename of the image.

- * info['images'][CAM_XXX][cam2img]: The transformation matrix recording the intrinsic parameters when projecting 3D points to each image plane. (3x3 list)
- * info['images'][CAM_XXX][sample_data_token]: Sample data token of image.
- * info['images'][CAM_XXX][timestamp]: Timestamp of the image.
- * info['images'][CAM_XXX][cam2ego]: The transformation matrix from this camera sensor to ego vehicle. (4x4 list)
- * info['images'][CAM_XXX][lidar2cam]: The transformation matrix from lidar sensor to this camera. (4x4 list)
- info['instances']: It is a list of dict. Each dict contains all annotation information of single instance. For the i-th instance:
 - * info['instances'][i]['bbox_3d']: List of 7 numbers representing the 3D bounding box in lidar coordinate system of the instance, in (x, y, z, l, w, h, yaw) order.
 - * info['instances'][i]['bbox_label_3d']: A int starting from 0 indicates the label of instance, while the -1 indicates ignore class.
 - * info['instances'][i]['bbox_3d_isvalid']: 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.

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:
 - Without info['instances'][i]['velocity']: There is no velocity measurement on Lyft.
 - Without info['instances'][i]['num_lidar_pts'] and info['instances'][i]['num_radar_pts']

Here we only explain the data recorded in the training info files. The same applies to the validation set and test set (without instances).

Please refer to [lyft_converter.py](#) for more details about the structure of lyft_infos_xxx.pkl.

13.3.3 Training pipeline

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),
    dict(
        type='LoadPointsFromMultiSweeps',
        sweeps_num=10),
    dict(type='LoadAnnotations3D', with_bbox_3d=True, with_label_3d=True),
    dict(
        type='GlobalRotScaleTrans',
```

(continues on next page)

(continued from previous page)

```

        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='PointShuffle'),
    dict(
        type='Pack3DDetInputs',
        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.

13.3.4 Evaluation

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

```

bash ./tools/dist_test.sh configs/pointpillars/pointpillars_hv_fpn_sbn-all_8xb2-2x_lyft-
→ 3d.py checkpoints/hv_pointpillars_fpn_sbn-all_2x8_2x_lyft-3d_20210517_202818-fc6904c3.
→ pth 8

```

13.3.5 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          |
+-----+-----+

```

13.3.6 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/pointpillars_hv_fpn_sbn-all_8xb2-2x_lyft-3d.py_
↪work_dirs/pp-lyft/latest.pth 8 --cfg-options test_evaluator.jsonfile_prefix=work_dirs/
↪pp-lyft/results_challenge test_evaluator.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.

13.4 Waymo Dataset

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

13.4.1 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

```

(continues on next page)

(continued from previous page)

				— 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.

13.4.2 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.

13.4.3 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
# git clone https://github.com/Abyssaledge/waymo-open-dataset-master waymo-od # if you
↳ want to use faster multi-thread version.
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
↳ {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/evaluation/functional/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/dist_test.sh configs/pointpillars/pointpillars_hv_secfn_sbn-all_16xb2-2x_waymo-
↪3d-car.py checkpoints/hv_pointpillars_secfn_sbn-2x16_2x_waymo-3d-car_latest.pth
```

`pklfile_prefix` should be set in `test_evaluator` of configuration if the bin file is needed to be generated, so you can add `--cfg-options "test_evaluator.pklfile_prefix=xxxx"` in the end of command if you want do it.

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.

13.4.4 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.

`submission_prefix` should be set in `test_evaluator` of configuration before you run the test command if you want to generate the bin files and make a submission to the leaderboard..

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/evaluation/
↪functional/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/evaluation/functional/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.

13.5 SUN RGB-D for 3D Object Detection

13.5.1 Dataset preparation

For the overall process, please refer to the [README](#) page for SUN RGB-D.

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
```

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)),:,:) = [];
```

(continues on next page)

(continued from previous page)

```

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
│   ├── 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.

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 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/xxxxxx.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['lidar_points']**: A dict containing all information related to the the lidar points.
 - * **info['lidar_points']['num_pts_feats']**: The feature dimension of point.
 - * **info['lidar_points']['lidar_path']**: The filename of the lidar point cloud data.
 - **info['images']**: A dict containing all information relate to the image data.
 - * **info['images']['CAM0']['img_path']**: The filename of the image.
 - * **info['images']['CAM0']['depth2img']**: Transformation matrix from depth to image with shape (4, 4).
 - * **info['images']['CAM0']['height']**: The height of image.
 - * **info['images']['CAM0']['width']**: The width of image.

- info['instances']: A list of dict contains all the annotations of this frame. Each dict corresponds to annotations of single instance. For the i-th instance:
 - * info['instances'][i]['bbox_3d']: List of 7 numbers representing the 3D bounding box in depth coordinate system.
 - * info['instances'][i]['bbox']: List of 4 numbers representing the 2D bounding box of the instance, in (x1, y1, x2, y2) order.
 - * info['instances'][i]['bbox_label_3d']: An int indicates the 3D label of instance and the -1 indicates ignore class.
 - * info['instances'][i]['bbox_label']: An int indicates the 2D label of instance and the -1 indicates ignore class.
- sunrgbd_infos_val.pkl: The val data infos, which shares the same format as sunrgbd_infos_train.pkl.

13.5.2 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='Pack3DDetInputs',
        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,
        load_dim=6,
        use_dim=[0, 1, 2]),
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations3D'),
    dict(type='LoadAnnotations', with_bbox=True),
    dict(type='Resize', scale=(1333, 600), keep_ratio=True),
    dict(type='RandomFlip', flip_ratio=0.0),
    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='Pack3DDetInputs',
        keys=['points', 'gt_bboxes_3d', 'gt_labels_3d', 'img', 'gt_bboxes', 'gt_bboxes_
↪ labels'])
]

```

Data augmentation for images:

- **Resize**: resize the input image, `keep_ratio=True` means the ratio of the image is kept unchanged.
- **RandomFlip**: randomly flip the input image.

The image augmentation functions are implemented in `MMDetection`.

13.5.3 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` for more details.

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.

13.6 ScanNet for 3D Object Detection

13.6.1 Dataset preparation

For the overall process, please refer to the [README](#) page for ScanNet.

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,
```

(continues on next page)

(continued from previous page)

```

    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 = {}
        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():

```

(continues on next page)

(continued from previous page)

```

    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 .npz files.

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.

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
├── meta_data
├── batch_load_scannet_data.py
├── load_scannet_data.py
├── scannet_utils.py
├── README.md
├── scans
├── scans_test
├── scannet_instance_data
└── points

```

(continues on next page)

(continued from previous page)

```

|   |— 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['lidar_points']`: A dict containing all information related to the lidar points.
 - * `info['lidar_points']['lidar_path']`: The filename of the lidar point cloud data.
 - * `info['lidar_points']['num_pts_feats']`: The feature dimension of point.
 - * `info['lidar_points']['axis_align_matrix']`: The transformation matrix to align the axis.
 - `info['pts_semantic_mask_path']`: The filename of the semantic mask annotation.
 - `info['pts_instance_mask_path']`: The filename of the instance mask annotation.
 - `info['instances']`: A list of dict contains all annotations, each dict contains all annotation information of single instance. For the *i*-th instance:
 - * `info['instances'][i]['bbox_3d']`: List of 6 numbers representing the axis-aligned 3D bounding box of the instance in depth coordinate system, in (x, y, z, l, w, h) order.
 - * `info['instances'][i]['bbox_label_3d']`: The label of each 3d bounding boxes.
- `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.

13.6.2 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'),
    dict(type='PointSample', num_points=40000),
    dict(
        type='RandomFlip3D',
        sync_2d=False,
        flip_ratio_bev_horizontal=0.5,
        flip_ratio_bev_vertical=0.5),
    dict(
        type='GlobalRotScaleTrans',
        rot_range=[-0.087266, 0.087266],
        scale_ratio_range=[1.0, 1.0],
        shift_height=True),
    dict(
        type='Pack3DDetInputs',
        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).

13.6.3 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` for more details.

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.

13.7 ScanNet for 3D Semantic Segmentation

13.7.1 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.

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.

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
│   ├── xxxx.bin
├── instance_mask
│   ├── xxxx.bin
├── semantic_mask
│   ├── xxxx.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.

13.7.2 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,
        with_label_3d=False,
```

(continues on next page)

(continued from previous page)

```

        with_mask_3d=False,
        with_seg_3d=True),
    dict(
        type='PointSegClassMapping'),
    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='Pack3DDetInputs', 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.

13.7.3 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](#).

13.7.4 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](#).

13.8 S3DIS for 3D Semantic Segmentation

13.8.1 Dataset preparation

For the overall process, please refer to the [README](#) page for S3DIS.

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.int64))
    np.save(f'{out_filename}_ins_label.npy', data_label[:, 7].astype(np.int64))

```

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.

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_idxes.npy: Re-sampling index for each scene. Different rooms will be sampled multiple times according to their number of points to balance training data.

13.8.2 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'),
    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],
        clip_range=[-0.05, 0.05]),
    dict(type='RandomDropPointsColor', drop_ratio=0.2),
```

(continues on next page)

(continued from previous page)

```
dict(type='Pack3DDetInputs', 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`.

13.8.3 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
train_dataloader = dict(
    batch_size=8,
    num_workers=4,
    persistent_workers=True,
    sampler=dict(type='DefaultSampler', shuffle=True),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        ann_files=[f's3dis_infos_Area_{i}.pkl' for i in train_area],
        meta_info=meta_info,
        data_prefix=data_prefix,
        pipeline=train_pipeline,
        modality=input_modality,
        ignore_index=len(class_names),
        scene_idxs=[
            f'seg_info/Area_{i}_resampled_scene_idxs.npy' for i in train_area
        ],
        test_mode=False))
test_dataloader = dict(
    batch_size=1,
```

(continues on next page)

(continued from previous page)

```
num_workers=1,
persistent_workers=True,
drop_last=False,
sampler=dict(type='DefaultSampler', shuffle=False),
dataset=dict(
    type=dataset_type,
    data_root=data_root,
    ann_files=f's3dis_infos_Area_{test_area}.pkl',
    metainfo=metainfo,
    data_prefix=data_prefix,
    pipeline=test_pipeline,
    modality=input_modality,
    ignore_index=len(class_names),
    scene_idxs=f'seg_info/Area_{test_area}_resampled_scene_idxs.npy',
    test_mode=True))
val_dataloader = test_dataloader
```

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.

SUPPORTED TASKS

14.1 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.

14.1.1 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` 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  
    │   │   └── velodyne_reduced  
    │   └── training  
    │       ├── calib  
    │       ├── image_2  
    │       ├── label_2  
    │       ├── velodyne  
    │       └── velodyne_reduced
```

(continues on next page)

(continued from previous page)

```

— kitti_gt_database
— kitti_infos_train.pkl
— kitti_infos_trainval.pkl
— kitti_infos_val.pkl
— kitti_infos_test.pkl
— kitti_dbinfos_train.pkl

```

14.1.2 Training

Then let us train a model with provided configs for PointPillars. 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/pointpillars/pointpillars_hv_secfpn_8xb6-160e_kitti-3d-
↪3class.py 8
```

Note that 8xb6 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](#). We have supported `--auto-scale-lr` to enable automatically scaling LR.

14.1.3 Quantitative Evaluation

During training, the model checkpoints will be evaluated regularly according to the setting of `train_cfg = dict(val_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/pointpillars_hv_secfpn_8xb6-160e_kitti-3d-
↪3class.py work_dirs/pointpillars/latest.pth 8
```


14.1.4 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 specify the `submission_prefix` for corresponding evaluator, e.g., add `test_evaluator = dict(type='KittiMetric', ann_file=data_root + 'kitti_infos_test.pkl', format_only=True, pklfile_prefix='results/kitti-3class/kitti_results', submission_prefix='results/kitti-3class/kitti_results')` in the configuration then you can get the results file. Please guarantee the `data_prefix` and `ann_file` in [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.

14.1.5 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 `--show` 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](#).

14.2 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.

14.2.1 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
```

(continues on next page)

(continued from previous page)

```

├── v1.0-test
├── v1.0-trainval
├── nusenes_database
├── nusenes_infos_train.pkl
├── nusenes_infos_trainval.pkl
├── nusenes_infos_val.pkl
├── nusenes_infos_test.pkl
└── nusenes_dbinfos_train.pkl

```

14.2.2 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-dcn_fpn_head-gn_8xb2-1x_nus-
↪mono3d.py 8
```

Note that 8xb2 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, you should add `--auto-scale-lr` to enable automatically scaling learning rate. A basic rule can be referred to [here](#).

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

```
./tools/dist_train.sh configs/fcos3d/fcos3d_r101-caffe-dcn_fpn_head-gn_8xb2-1x_nus-
↪mono3d_finetune.py 8
```

After training a baseline model with the previous script, please remember to modify the path [here](#) correspondingly.

14.2.3 Quantitative Evaluation

During training, the model checkpoints will be evaluated regularly according to the setting of `train_cfg = dict(val_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

```

(continues on next page)

(continued from previous page)

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		

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-dcn_fpn_head-gn_8xb2-1x_nus-mono3d.
↪py work_dirs/fcos3d/latest.pth 8
```

14.2.4 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 specify the `jsonfile_prefix` for corresponding evaluator, e.g., add `test_evaluator = dict(type='NuscenesMetric', jsonfile_prefix=work_dirs/fcos3d/test_submission)` in the configuration then you can get the results file.

Please guarantee the `data_prefix` and `ann_file` in [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.

14.2.5 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.

14.3 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.

14.3.1 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
│       ├── scannet_infos_train.pkl
│       ├── scannet_infos_val.pkl
│       └── scannet_infos_test.pkl
```

14.3.2 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_2xb16-cosine-200e_scannet-seg.py 2
```

Note that 2xb16 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](#).

14.3.3 Quantitative Evaluation

During training, the model checkpoints will be evaluated regularly according to the setting of `train_cfg = dict(val_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 | _
↪showercurtain | 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 8
```

14.3.4 Testing and Making a Submission

If you would like to only conduct inference or test the model performance on the online benchmark, you should change `ann_file='scannet_infos_val.pkl'` to `ann_file='scannet_infos_test.pkl'` in the ScanNet dataset's `config`. Remember to specify the `submission_prefix` in the `test_evaluator`, e.g., adding `test_evaluator = dict(type='SegMetric', submission_prefix=work_dirs/pointnet2_ssg/test_submission)` or just add `--cfg-options test_evaluator.submission_prefix=work_dirs/pointnet2_ssg/test_submission` in the end of command. After generating the results, you can basically compress the folder and upload to the [ScanNet evaluation server](#).

14.3.5 Qualitative Validation

MMDetection3D also provides versatile tools for visualization such that you can use `tools/misc/visualize_results.py` with results pkl file for offline visualization of add `--show` in the end of test command to do the online 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](#).

CUSTOMIZATION

15.1 Customize Datasets

In this note, you will know how to train and test predefined models with customized datasets.

The basic steps are as below:

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

15.1.1 Data Preparation

The ideal situation is that we can reorganize the customized raw data and convert the annotation format into KITTI style. However, considering some calibration files and 3D annotations in KITTI format are difficult to obtain for customized datasets, we introduce the basic data format in the doc.

Basic Data Format

Point cloud Format

Currently, we only support .bin format point cloud for training and inference. Before training on your own datasets, you need to convert your point cloud files with other formats to .bin files. The common point cloud data formats include .pcd and .las, we list some open-source tools for reference.

1. Convert .pcd to .bin: <https://github.com/DanielPollithy/pypcd>

- You can install pypcd with the following command:

```
pip install git+https://github.com/DanielPollithy/pypcd.git
```

- You can use the following script to read the .pcd file and convert it to .bin format for saving:

```
import numpy as np
from pypcd import pypcd

pcd_data = pypcd.PointCloud.from_path('point_cloud_data.pcd')
points = np.zeros([pcd_data.width, 4], dtype=np.float32)
points[:, 0] = pcd_data.pc_data['x'].copy()
points[:, 1] = pcd_data.pc_data['y'].copy()
```

(continues on next page)

(continued from previous page)

```
points[:, 2] = pcd_data.pc_data['z'].copy()
points[:, 3] = pcd_data.pc_data['intensity'].copy().astype(np.float32)
with open('point_cloud_data.bin', 'wb') as f:
    f.write(points.tobytes())
```

2. Convert .las to .bin: The common conversion path is .las -> .pcd -> .bin, and the conversion path .las -> .pcd can be achieved through [this tool](#).

Label Format

The most basic information: 3D bounding box and category label of each scene need to be contained in the .txt annotation file. Each line represents a 3D box in a certain scene as follow:

```
# format: [x, y, z, dx, dy, dz, yaw, category_name]
1.23 1.42 0.23 3.96 1.65 1.55 1.56 Car
3.51 2.15 0.42 1.05 0.87 1.86 1.23 Pedestrian
...
```

Note: Currently we only support KITTI Metric evaluation for customized datasets evaluation.

The 3D Box should be stored in unified 3D coordinates.

Calibration Format

For the point cloud data collected by each LiDAR, they are usually fused and converted to a certain LiDAR coordinate. So typically the calibration information file should contain the intrinsic matrix of each camera and the transformation extrinsic matrix from the LiDAR to each camera in .txt calibration file, while P_x represents the intrinsic matrix of camera_x and lidar2camx represents the transformation extrinsic matrix from the lidar to camera_x.

```
P0
P1
P2
P3
P4
...
lidar2cam0
lidar2cam1
lidar2cam2
lidar2cam3
lidar2cam4
...
```


Raw Data Structure

LiDAR-Based 3D Detection

The raw data for LiDAR-based 3D object detection are typically organized as follows, where `ImageSets` contains split files indicating which files belong to training/validation set, `points` includes point cloud data which are supposed to be stored in `.bin` format and `labels` includes label files for 3D detection.

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
│   ├── custom
│   │   ├── ImageSets
│   │   │   ├── train.txt
│   │   │   └── val.txt
│   │   ├── points
│   │   │   ├── 000000.bin
│   │   │   ├── 000001.bin
│   │   │   └── ...
│   │   └── labels
│   │       ├── 000000.txt
│   │       ├── 000001.txt
│   │       └── ...
```

Vision-Based 3D Detection

The raw data for vision-based 3D object detection are typically organized as follows, where `ImageSets` contains split files indicating which files belong to training/validation set, `images` contains the images from different cameras, for example, images from `camera_x` need to be placed in `images/images_x`, `calibs` contains calibration information files which store the camera intrinsic matrix of each camera, and `labels` includes label files for 3D detection.

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
│   ├── custom
│   │   ├── ImageSets
│   │   │   ├── train.txt
│   │   │   └── val.txt
│   │   ├── calibs
│   │   │   ├── 000000.txt
│   │   │   ├── 000001.txt
│   │   │   └── ...
│   │   ├── images
│   │   │   ├── images_0
│   │   │   │   ├── 000000.png
│   │   │   │   ├── 000001.png
│   │   │   │   └── ...
│   │   │   └── images_1
```

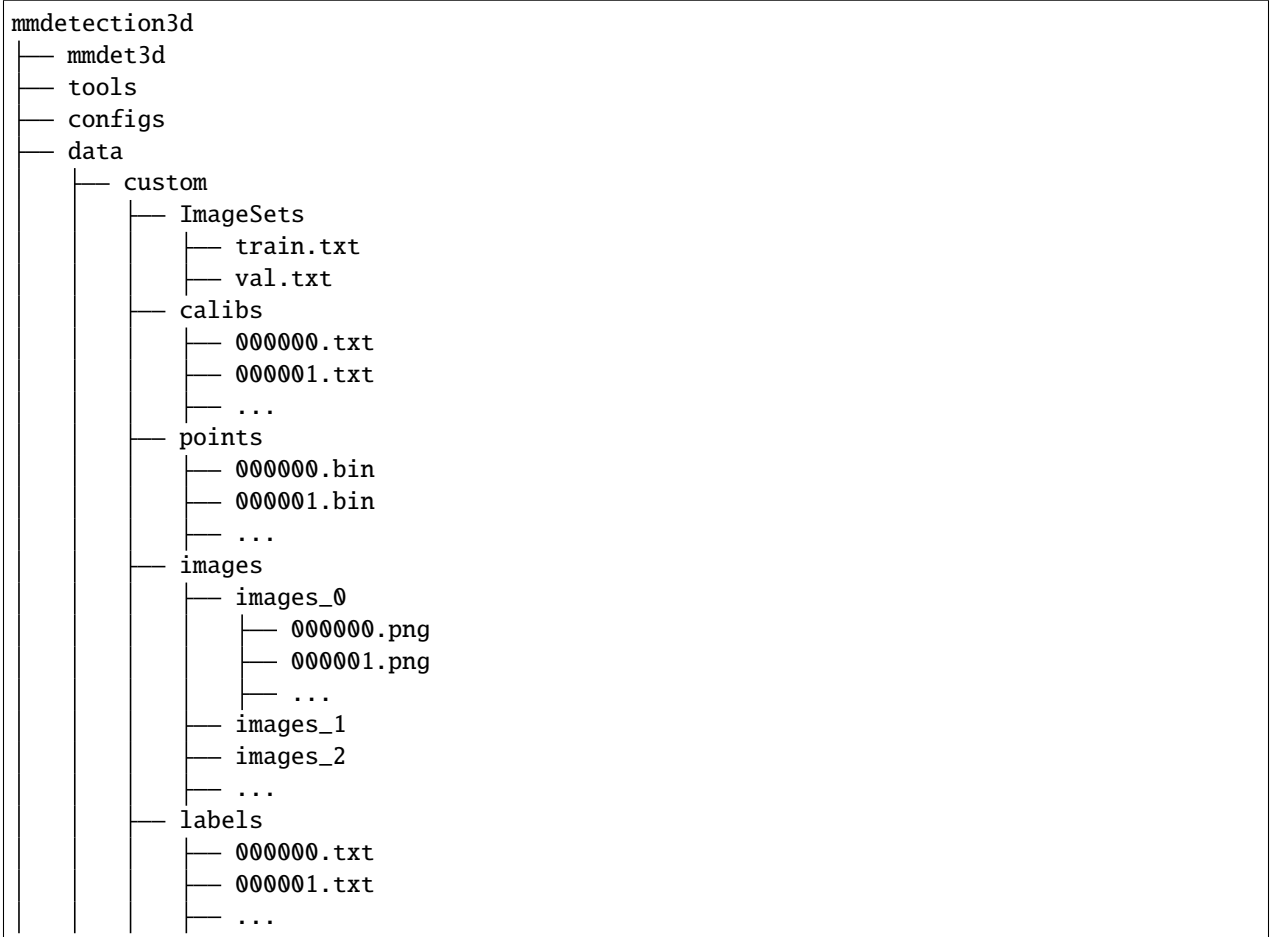
(continues on next page)

(continued from previous page)



Multi-Modality 3D Detection

The raw data for multi-modality 3D object detection are typically organized as follows. Different from vision-based 3D object detection, calibration information files in `calibs` store the camera intrinsic matrix of each camera and extrinsic matrix.



LiDAR-Based 3D Semantic Segmentation

The raw data for LiDAR-based 3D semantic segmentation are typically organized as follows, where `ImageSets` contains split files indicating which files belong to training/validation set, `points` includes point cloud data, and `semantic_mask` includes point-level label.

```
mmdetection3d
├── mmdet3d
├── tools
├── configs
├── data
│   ├── custom
│   │   ├── ImageSets
│   │   │   ├── train.txt
│   │   │   └── val.txt
│   │   ├── points
│   │   │   ├── 000000.bin
│   │   │   ├── 000001.bin
│   │   │   └── ...
│   │   └── semantic_mask
│   │       ├── 000000.bin
│   │       ├── 000001.bin
│   │       └── ...
```

Data Converter

Once you prepared the raw data following our instruction, you can directly use the following command to generate training/validation information files.

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

15.1.2 An example of customized dataset

Once we finish data preparation, we can create a new dataset in `mmdet3d/datasets/my_dataset.py` to load the data.

```
import mmengine

from mmdet3d.registry import DATASETS
from .det3d_dataset import Det3DDataset

@DATASETS.register_module()
class MyDataset(Det3DDataset):

    # replace with all the classes in customized pkl info file
    METAINFO = {
        'classes': ('Pedestrian', 'Cyclist', 'Car')
    }
```

(continues on next page)

(continued from previous page)

```

def parse_ann_info(self, info):
    """Process the `instances` in data info to `ann_info`.

    Args:
        info (dict): Data information of single data sample.

    Returns:
        dict: Annotation information consists of the following keys:

        - gt_bboxes_3d (:obj:`LiDARInstance3DBBoxes`):
          3D ground truth bboxes.
        - gt_labels_3d (np.ndarray): Labels of ground truths.
    """
    ann_info = super().parse_ann_info(info)
    if ann_info is None:
        ann_info = dict()
        # empty instance
        ann_info['gt_bboxes_3d'] = np.zeros((0, 7), dtype=np.float32)
        ann_info['gt_labels_3d'] = np.zeros(0, dtype=np.int64)

    # filter the gt classes not used in training
    ann_info = self._remove_dontcare(ann_info)
    gt_bboxes_3d = LiDARInstance3DBBoxes(ann_info['gt_bboxes_3d'])
    ann_info['gt_bboxes_3d'] = gt_bboxes_3d
    return ann_info

```

After the data pre-processing, there are two steps for users to train the customized new dataset:

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

Here we take training PointPillars on customized dataset as an example:

Prepare a config

Here we demonstrate a config sample for pure point cloud training.

Prepare dataset config

In configs/_base_/datasets/custom.py:

```

# dataset settings
dataset_type = 'MyDataset'
data_root = 'data/custom/'
class_names = ['Pedestrian', 'Cyclist', 'Car'] # replace with your dataset class
point_cloud_range = [0, -40, -3, 70.4, 40, 1] # adjust according to your dataset
input_modality = dict(use_lidar=True, use_camera=False)
metainfo = dict(classes=class_names)

train_pipeline = [
    dict(

```

(continues on next page)

(continued from previous page)

```

        type='LoadPointsFromFile',
        coord_type='LIDAR',
        load_dim=4, # replace with your point cloud data dimension
        use_dim=4), # replace with the actual dimension used in training and inference
    dict(
        type='LoadAnnotations3D',
        with_bbox_3d=True,
        with_label_3d=True),
    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='Pack3DDetInputs',
        keys=['points', 'gt_bboxes_3d', 'gt_labels_3d'])
]
test_pipeline = [
    dict(
        type='LoadPointsFromFile',
        coord_type='LIDAR',
        load_dim=4, # replace with your point cloud data dimension
        use_dim=4),
    dict(type='Pack3DDetInputs', keys=['points'])
]
# construct a pipeline for data and gt loading in show function
eval_pipeline = [
    dict(type='LoadPointsFromFile', coord_type='LIDAR', load_dim=4, use_dim=4),
    dict(type='Pack3DDetInputs', keys=['points']),
]
train_dataloader = dict(
    batch_size=6,
    num_workers=4,
    persistent_workers=True,
    sampler=dict(type='DefaultSampler', shuffle=True),
    dataset=dict(
        type='RepeatDataset',
        times=2,
        dataset=dict(
            type=dataset_type,
            data_root=data_root,
            ann_file='custom_infos_train.pkl', # specify your training pkl info
            data_prefix=dict(pts='points'),

```

(continues on next page)

(continued from previous page)

```

        pipeline=train_pipeline,
        modality=input_modality,
        test_mode=False,
        metainfo=metainfo,
        box_type_3d='LiDAR'))
val_dataloader = dict(
    batch_size=1,
    num_workers=1,
    persistent_workers=True,
    drop_last=False,
    sampler=dict(type='DefaultSampler', shuffle=False),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        data_prefix=dict(pts='points'),
        ann_file='custom_infos_val.pkl', # specify your validation pkl info
        pipeline=test_pipeline,
        modality=input_modality,
        test_mode=True,
        metainfo=metainfo,
        box_type_3d='LiDAR'))
val_evaluator = dict(
    type='KittiMetric',
    ann_file=data_root + 'custom_infos_val.pkl', # specify your validation pkl info
    metric='bbox')

```

Prepare model config

For voxel-based detectors such as SECOND, PointPillars and CenterPoint, the point cloud range and voxel size should be adjusted according to your dataset. 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 $[(1-(-3))/0.1+1, (40-(-40))/0.05, (70.4-0)/0.05]=[41, 1600, 1408]$. When changing `point_cloud_range`, remember to change the shape of intermediate feature map in `middle_encoder` according to the `voxel_size`.

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 objects in the entire training dataset as `anchor_size` to obtain the best results.

In `configs/_base_/models/pointpillars_hv_secfpn_custom.py`:

```

voxel_size = [0.16, 0.16, 4] # adjust according to your dataset
point_cloud_range = [0, -39.68, -3, 69.12, 39.68, 1] # adjust according to your dataset
model = dict(
    type='VoxelNet',
    data_preprocessor=dict(
        type='Det3DDataPreprocessor',
        voxel=True,

```

(continues on next page)

(continued from previous page)

```

voxel_layer=dict(
    max_num_points=32,
    point_cloud_range=point_cloud_range,
    voxel_size=voxel_size,
    max_voxels=(16000, 40000))),
voxel_encoder=dict(
    type='PillarFeatureNet',
    in_channels=4,
    feat_channels=[64],
    with_distance=False,
    voxel_size=voxel_size,
    point_cloud_range=point_cloud_range),
# the `output_shape` should be adjusted according to `point_cloud_range`
# and `voxel_size`
middle_encoder=dict(
    type='PointPillarsScatter', in_channels=64, output_shape=[496, 432]),
backbone=dict(
    type='SECOND',
    in_channels=64,
    layer_nums=[3, 5, 5],
    layer_strides=[2, 2, 2],
    out_channels=[64, 128, 256]),
neck=dict(
    type='SECONDFPN',
    in_channels=[64, 128, 256],
    upsample_strides=[1, 2, 4],
    out_channels=[128, 128, 128]),
bbox_head=dict(
    type='Anchor3DHead',
    num_classes=3,
    in_channels=384,
    feat_channels=384,
    use_direction_classifier=True,
    assign_per_class=True,
    # adjust the `ranges` and `sizes` according to your dataset
    anchor_generator=dict(
        type='AlignedAnchor3DRangeGenerator',
        ranges=[
            [0, -39.68, -0.6, 69.12, 39.68, -0.6],
            [0, -39.68, -0.6, 69.12, 39.68, -0.6],
            [0, -39.68, -1.78, 69.12, 39.68, -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),
    diff_rad_by_sin=True,
    bbox_coder=dict(type='DeltaXYZWLHRBBoxCoder'),
    loss_cls=dict(
        type='mmdet.FocalLoss',
        use_sigmoid=True,
        gamma=2.0,
        alpha=0.25,

```

(continues on next page)

(continued from previous page)

```

        loss_weight=1.0),
    loss_bbox=dict(
        type='mmdet.SmoothL1Loss', beta=1.0 / 9.0, loss_weight=2.0),
    loss_dir=dict(
        type='mmdet.CrossEntropyLoss', use_sigmoid=False,
        loss_weight=0.2)),
# model training and testing settings
train_cfg=dict(
    assigner=[
        dict( # for Pedestrian
            type='Max3DIoUAssigner',
            iou_calculator=dict(type='BboxOverlapsNearest3D'),
            pos_iou_thr=0.5,
            neg_iou_thr=0.35,
            min_pos_iou=0.35,
            ignore_iof_thr=-1),
        dict( # for Cyclist
            type='Max3DIoUAssigner',
            iou_calculator=dict(type='BboxOverlapsNearest3D'),
            pos_iou_thr=0.5,
            neg_iou_thr=0.35,
            min_pos_iou=0.35,
            ignore_iof_thr=-1),
        dict( # for Car
            type='Max3DIoUAssigner',
            iou_calculator=dict(type='BboxOverlapsNearest3D'),
            pos_iou_thr=0.6,
            neg_iou_thr=0.45,
            min_pos_iou=0.45,
            ignore_iof_thr=-1),
    ],
    allowed_border=0,
    pos_weight=-1,
    debug=False),
test_cfg=dict(
    use_rotate_nms=True,
    nms_across_levels=False,
    nms_thr=0.01,
    score_thr=0.1,
    min_bbox_size=0,
    nms_pre=100,
    max_num=50))

```


Prepare overall config

We combine all the configs above in `configs/pointpillars/pointpillars_hv_secfpn_8xb6_custom.py`:

```
_base_ = [
    '../_base_/models/pointpillars_hv_secfpn_custom.py',
    '../_base_/datasets/custom.py',
    '../_base_/schedules/cyclic-40e.py', '../_base_/default_runtime.py'
]
```

Visualize your dataset (optional)

To validate whether your prepared data and config are correct, it's highly recommended to use `tools/misc/browse_dataset.py` script to visualize your dataset and annotations before training and validation. Please refer to [visualization doc](#) for more details.

15.1.3 Evaluation

Once the data and config have been prepared, you can directly run the training/testing script following our doc.

Note: We only provide an implementation for KITTI style evaluation for the customized dataset. It should be included in the dataset config:

```
val_evaluator = dict(
    type='KittiMetric',
    ann_file=data_root + 'custom_infos_val.pkl', # specify your validation pkl info
    metric='bbox')
```

15.2 Customize Models

We basically categorize model components into 6 types:

- encoder: Including 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`.

15.2.1 Develop new components

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 HV-SECOND)

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

```
import torch.nn as nn

from mmdet3d.registry import MODELS

@MODELS.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.voxel_encoder'],
    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=yyy),
    ...
)
```

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`.

```
from mmengine.model import BaseModule

from mmdet3d.registry import MODELS

@MODELS.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=yyy),
    ...
)
```

Add a new neck

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

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

```
from mmengine.model import BaseModule

from mmdet3d.registry import MODELS

@MODELS.register_module()
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 to 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]),
    ...
)
```

Add a new head

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 two functions of the new module as the following. Sometimes other related functions like `loss` and `get_targets` are also required.

```
from mmengine.model import BaseModule

from mmdet3d.registry import MODELS

@MODELS.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,
                 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),
```

(continues on next page)

(continued from previous page)

```

        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):
        pass

```

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 mmdet.models.roi_heads import BaseRoIHead

from mmdet3d.registry import MODELS, TASK_UTILS


class Base3DRoIHead(BaseRoIHead):
    """Base class for 3d RoIHeads."""

    def __init__(self,
                 bbox_head=None,
                 bbox_roi_extractor=None,
                 mask_head=None,
                 mask_roi_extractor=None,
                 train_cfg=None,
                 test_cfg=None,
                 init_cfg=None):
        super(Base3DRoIHead, self).__init__(
            bbox_head=bbox_head,
            bbox_roi_extractor=bbox_roi_extractor,
            mask_head=mask_head,
            mask_roi_extractor=mask_roi_extractor,
            train_cfg=train_cfg,
            test_cfg=test_cfg,
            init_cfg=init_cfg)

    def init_bbox_head(self, bbox_roi_extractor: dict,
                      bbox_head: dict) -> None:
        """Initialize box head and box roi extractor.

        Args:
            bbox_roi_extractor (dict or ConfigDict): Config of box
                roi extractor.
            bbox_head (dict or ConfigDict): Config of box in box head.
        """
        self.bbox_roi_extractor = MODELS.build(bbox_roi_extractor)
        self.bbox_head = MODELS.build(bbox_head)

    def init_assigner_sampler(self):

```

(continues on next page)

(continued from previous page)

```

"""Initialize assigner and sampler."""
self.bbox_assigner = None
self.bbox_sampler = None
if self.train_cfg:
    if isinstance(self.train_cfg.assigner, dict):
        self.bbox_assigner = TASK_UTILS.build(self.train_cfg.assigner)
    elif isinstance(self.train_cfg.assigner, list):
        self.bbox_assigner = [
            TASK_UTILS.build(res) for res in self.train_cfg.assigner
        ]
    self.bbox_sampler = TASK_UTILS.build(self.train_cfg.sampler)

def init_mask_head(self):
    """Initialize mask head, skip since `PartAggregationROIHead` does not
    have one."""
    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 typing import Dict, List, Tuple

from mmdet.models.task_modules import AssignResult, SamplingResult
from mmengine import ConfigDict
from torch import Tensor
from torch.nn import functional as F

from mmdet3d.registry import MODELS
from mmdet3d.structures import bbox3d2roi
from mmdet3d.utils import InstanceList
from ...structures.det3d_data_sample import SampleList
from .base_3droi_head import Base3DRoIHead

@MODELS.register_module()
class PartAggregationROIHead(Base3DRoIHead):
    """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.
        bbox_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: dict,
                 num_classes: int = 3,

```

(continues on next page)

(continued from previous page)

```

        seg_roi_extractor: dict = None,
        bbox_head: dict = None,
        bbox_roi_extractor: dict = None,
        train_cfg: dict = None,
        test_cfg: dict = None,
        init_cfg: dict = None) -> None:
    super(PartAggregationROIHead, self).__init__(
        bbox_head=bbox_head,
        bbox_roi_extractor=bbox_roi_extractor,
        train_cfg=train_cfg,
        test_cfg=test_cfg,
        init_cfg=init_cfg)
    self.num_classes = num_classes
    assert semantic_head is not None
    self.init_seg_head(seg_roi_extractor, semantic_head)

    def init_seg_head(self, seg_roi_extractor: dict,
                      semantic_head: dict) -> None:
        """Initialize semantic head and seg roi extractor.

        Args:
            seg_roi_extractor (dict): Config of seg
                roi extractor.
            semantic_head (dict): Config of semantic head.
        """
        self.semantic_head = MODELS.build(semantic_head)
        self.seg_roi_extractor = MODELS.build(seg_roi_extractor)

    @property
    def with_semantic(self):
        """bool: whether the head has semantic branch"""
        return hasattr(self,
                        'semantic_head') and self.semantic_head is not None

    def predict(self,
                feats_dict: Dict,
                rpn_results_list: InstanceList,
                batch_data_samples: SampleList,
                rescale: bool = False,
                **kwargs) -> InstanceList:
        """Perform forward propagation of the roi head and predict detection
        results on the features of the upstream network.

        Args:
            feats_dict (dict): Contains features from the first stage.
            rpn_results_list (List[:obj:`InstanceData`]): Detection results
                of rpn head.
            batch_data_samples (List[:obj:`Det3DDataSample`]): The Data
                samples. It usually includes information such as
                `gt_instance_3d`, `gt_panoptic_seg_3d` and `gt_sem_seg_3d`.
            rescale (bool): If True, return boxes in original image space.
                Defaults to False.

```

(continues on next page)

(continued from previous page)

```

Returns:
    list[:obj:`InstanceData`]: Detection results of each sample
    after the post process.
    Each item usually contains following keys.

    - scores_3d (Tensor): Classification scores, has a shape
      (num_instances, )
    - labels_3d (Tensor): Labels of bboxes, has a shape
      (num_instances, ).
    - bboxes_3d (BaseInstance3DBBoxes): Prediction of bboxes,
      contains a tensor with shape (num_instances, C), where
      C >= 7.
    """
    assert self.with_bbox, 'Bbox head must be implemented in PartA2.'
    assert self.with_semantic, 'Semantic head must be implemented' \
        ' in PartA2.'

    batch_input metas = [
        data_samples.metainfo for data_samples in batch_data_samples
    ]
    voxels_dict = feats_dict.pop('voxels_dict')
    # TODO: Split predict semantic and bbox
    results_list = self.predict_bbox(feats_dict, voxels_dict,
                                     batch_input metas, rpn_results_list,
                                     self.test_cfg)

    return results_list

def predict_bbox(self, feats_dict: Dict, voxel_dict: Dict,
                 batch_input metas: List[dict],
                 rpn_results_list: InstanceList,
                 test_cfg: ConfigDict) -> InstanceList:
    """Perform forward propagation of the bbox head and predict detection
    results on the features of the upstream network.

    Args:
        feats_dict (dict): Contains features from the first stage.
        voxel_dict (dict): Contains information of voxels.
        batch_input metas (list[dict], Optional): Batch image meta info.
            Defaults to None.
        rpn_results_list (List[:obj:`InstanceData`]): Detection results
            of rpn head.
        test_cfg (Config): Test config.

    Returns:
        list[:obj:`InstanceData`]: Detection results of each sample
        after the post process.
        Each item usually contains following keys.

        - scores_3d (Tensor): Classification scores, has a shape
          (num_instances, )
        - labels_3d (Tensor): Labels of bboxes, has a shape

```

(continues on next page)

(continued from previous page)

```

        (num_instances, ).
    - bboxes_3d (BaseInstance3DBBoxes): Prediction of bboxes,
      contains a tensor with shape (num_instances, C), where
      C >= 7.
    """
    ...

def loss(self, feats_dict: Dict, rpn_results_list: InstanceList,
        batch_data_samples: SampleList, **kwargs) -> dict:
    """Perform forward propagation and loss calculation of the detection
    roi on the features of the upstream network.

    Args:
        feats_dict (dict): Contains features from the first stage.
        rpn_results_list (List[:obj:`InstanceData`]): Detection results
            of rpn head.
        batch_data_samples (List[:obj:`Det3DDataSample`]): The Data
            samples. It usually includes information such as
            `gt_instance_3d`, `gt_panoptic_seg_3d` and `gt_sem_seg_3d`.

    Returns:
        dict[str, Tensor]: A dictionary of loss components
    """
    assert len(rpn_results_list) == len(batch_data_samples)
    losses = dict()
    batch_gt_instances_3d = []
    batch_gt_instances_ignore = []
    voxels_dict = feats_dict.pop('voxels_dict')
    for data_sample in batch_data_samples:
        batch_gt_instances_3d.append(data_sample.gt_instances_3d)
        if 'ignored_instances' in data_sample:
            batch_gt_instances_ignore.append(data_sample.ignored_instances)
        else:
            batch_gt_instances_ignore.append(None)
    if self.with_semantic:
        semantic_results = self._semantic_forward_train(
            feats_dict, voxels_dict, batch_gt_instances_3d)
        losses.update(semantic_results.pop('loss_semantic'))

    sample_results = self._assign_and_sample(rpn_results_list,
                                             batch_gt_instances_3d)
    if self.with_bbox:
        feats_dict.update(semantic_results)
        bbox_results = self._bbox_forward_train(feats_dict, voxels_dict,
                                                sample_results)
        losses.update(bbox_results['loss_bbox'])

    return losses

```

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/roi_heads/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'],
    allow_failed_imports=False)
```

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='mmdet.FocalLoss',
                use_sigmoid=True,
                reduction='sum',
                gamma=2.0,
                alpha=0.25,
                loss_weight=1.0),
            loss_part=dict(
                type='mmdet.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,
                mode='max')),
        bbox_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],
```

(continues on next page)

(continued from previous page)

```

merge_conv_channels=[128, 128],
down_conv_channels=[128, 256],
bbox_coder=dict(type='DeltaXYZWLRBBoxCoder'),
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='mmdet.SmoothL1Loss',
    beta=1.0 / 9.0,
    reduction='sum',
    loss_weight=1.0),
loss_cls=dict(
    type='mmdet.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.

Add a 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 to implement it in `mmdet3d/models/losses/my_loss.py`. The decorator `weighted_loss` enables the loss to be weighted for each element.

```

import torch
import torch.nn as nn
from mmdet.models.losses.utils import weighted_loss

from mmdet3d.registry import MODELS

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

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

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

```

(continues on next page)

(continued from previous page)

```

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'],
    allow_failed_imports=False)

```

to the config file and achieve the same goal.

To use it, users should 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)
```

15.3 Customize Runtime Settings

15.3.1 Customize optimization settings

Optimization related configuration is now all managed by `optim_wrapper` which usually has three fields: `optimizer`, `paramwise_cfg`, `clip_grad`. Please refer to [OptimWrapper](#) for more details. See the example below, where `AdamW` is used as an optimizer, the learning rate of the backbone is reduced by a factor of 10, and gradient clipping is added.

```

optim_wrapper = dict(
    type='OptimWrapper',
    # optimizer
    optimizer=dict(
        type='AdamW',
        lr=0.0001,
        weight_decay=0.05,
        eps=1e-8,
        betas=(0.9, 0.999)),

```

(continues on next page)

(continued from previous page)

```
# Parameter-level learning rate and weight decay settings
paramwise_cfg=dict(
    custom_keys={
        'backbone': dict(lr_mult=0.1, decay_mult=1.0),
    },
    norm_decay_mult=0.0),

# gradient clipping
clip_grad=dict(max_norm=0.01, norm_type=2))
```

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 in `optim_wrapper` 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:

```
optim_wrapper = dict(
    type='OptimWrapper',
    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 `optimizer`. The users can directly set arguments following the [API doc](#) of PyTorch.

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/engine/optimizers`, and then implement the new optimizer in a file, e.g., in `mmdet3d/engine/optimizers/my_optimizer.py`:

```
from torch.optim import Optimizer

from mmdet3d.registry import OPTIMIZERS

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

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

2. Add the optimizer to registry

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

- Modify `mmdet3d/engine/optimizers/__init__.py` to import it.

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

```
from .my_optimizer import MyOptimizer
```

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

```
custom_imports = dict(imports=['mmdet3d.engine.optimizers.my_optimizer'], allow_
↪ failed_imports=False)
```

The module `mmdet3d.engine.optimizers.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.engine.optimizers.my_optimizer.MyOptimizer` **cannot** be imported directly.

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

3. Specify the optimizer in the config file

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

```
optim_wrapper = dict(
    type='OptimWrapper',
    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:

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

Customize optimizer wrapper constructor

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

```
from mmengine.optim import DefaultOptimWrapperConstructor

from mmdet3d.registry import OPTIM_WRAPPER_CONSTRUCTORS
from .my_optimizer import MyOptimizer

@OPTIM_WRAPPER_CONSTRUCTORS.register_module()
class MyOptimizerWrapperConstructor(DefaultOptimWrapperConstructor):
```

(continues on next page)

(continued from previous page)

```

def __init__(self,
              optim_wrapper_cfg: dict,
              paramwise_cfg: Optional[dict] = None):
    pass

def __call__(self, model: nn.Module) -> OptimWrapper:

    return optim_wrapper

```

The default optimizer wrapper constructor is implemented [here](#), which could also serve as a template for the new optimizer wrapper constructor.

Additional settings

Tricks not implemented by the optimizer should be implemented through optimizer wrapper 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:

```

optim_wrapper = dict(
    _delete_=True, clip_grad=dict(max_norm=35, norm_type=2))

```

If your config inherits the base config which already sets the `optim_wrapper`, you might need `_delete_=True` to override the unnecessary settings. See the [config documentation](#) for more details.

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

```

param_scheduler = [
    # learning rate scheduler
    # During the first 8 epochs, learning rate increases from 0 to lr * 10
    # during the next 12 epochs, learning rate decreases from lr * 10 to lr * 1e-4
    dict(
        type='CosineAnnealingLR',
        T_max=8,
        eta_min=lr * 10,
        begin=0,
        end=8,
        by_epoch=True,
        convert_to_iter_based=True),
    dict(
        type='CosineAnnealingLR',
        T_max=12,
        eta_min=lr * 1e-4,
        begin=8,
        end=20,
        by_epoch=True,

```

(continues on next page)

(continued from previous page)

```

        convert_to_iter_based=True),
    # momentum scheduler
    # During the first 8 epochs, momentum increases from 0 to 0.85 / 0.95
    # during the next 12 epochs, momentum increases from 0.85 / 0.95 to 1
    dict(
        type='CosineAnnealingMomentum',
        T_max=8,
        eta_min=0.85 / 0.95,
        begin=0,
        end=8,
        by_epoch=True,
        convert_to_iter_based=True),
    dict(
        type='CosineAnnealingMomentum',
        T_max=12,
        eta_min=1,
        begin=8,
        end=20,
        by_epoch=True,
        convert_to_iter_based=True)
]

```

15.3.2 Customize training schedules

By default we use step learning rate with 1x schedule, this calls `MultiStepLR` in MMEEngine. We support many other learning rate schedule [here](#), such as `CosineAnnealingLR` and `PolyLR` schedules. Here are some examples:

- Poly schedule:

```

param_scheduler = [
    dict(
        type='PolyLR',
        power=0.9,
        eta_min=1e-4,
        begin=0,
        end=8,
        by_epoch=True)]

```

- CosineAnnealing schedule:

```

param_scheduler = [
    dict(
        type='CosineAnnealingLR',
        T_max=8,
        eta_min=lr * 1e-5,
        begin=0,
        end=8,
        by_epoch=True)]

```

15.3.3 Customize train loop

By default, EpochBasedTrainLoop is used in train_cfg and validation is done after every train epoch, as follows:

```
train_cfg = dict(type='EpochBasedTrainLoop', max_epochs=12, val_begin=1, val_interval=1)
```

Actually, both IterBasedTrainLoop and EpochBasedTrainLoop support dynamic interval, see the following example:

```
# Before 365001th iteration, we do evaluation every 5000 iterations.
# After 365000th iteration, we do evaluation every 368750 iterations,
# which means that we do evaluation at the end of training.

interval = 5000
max_iters = 368750
dynamic_intervals = [(max_iters // interval * interval + 1, max_iters)]
train_cfg = dict(
    type='IterBasedTrainLoop',
    max_iters=max_iters,
    val_interval=interval,
    dynamic_intervals=dynamic_intervals)
```

15.3.4 Customize hooks

Customize self-implemented hooks

1. Implement a new hook

MMEngine provides many useful [hooks](#), but there are some occasions when the users might need to implement a new hook. MMDetection3D supports customized hooks in training based on MMEngine after v1.1.0rc0. Thus the users could implement a hook directly in mmdet3d or their mmdet3d-based codebases and use the hook by only modifying the config in training. Here we give an example of creating a new hook in mmdet3d and using it in training.

```
from mmengine.hooks import Hook

from mmdet3d.registry import HOOKS

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

    def __init__(self, a, b):

    def before_run(self, runner) -> None:

    def after_run(self, runner) -> None:

    def before_train(self, runner) -> None:

    def after_train(self, runner) -> None:

    def before_train_epoch(self, runner) -> None:
```

(continues on next page)

(continued from previous page)

```

def after_train_epoch(self, runner) -> None:

def before_train_iter(self,
                        runner,
                        batch_idx: int,
                        data_batch: DATA_BATCH = None) -> None:

def after_train_iter(self,
                     runner,
                     batch_idx: int,
                     data_batch: DATA_BATCH = None,
                     outputs: Optional[dict] = None) -> None:

```

Depending on the functionality of the hook, users need to specify what the hook will do at each stage of the training in `before_run`, `after_run`, `before_train`, `after_train`, `before_train_epoch`, `after_train_epoch`, `before_train_iter`, and `after_train_iter`. There are more points where hooks can be inserted, refer to [base hook class](#) for more details.

2. Register the new hook

Then we need to make `MyHook` imported. Assuming the file is in `mmdet3d/engine/hooks/my_hook.py`, there are two ways to do that:

- Modify `mmdet3d/engine/hooks/__init__.py` to import it.

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

```
from .my_hook import MyHook
```

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

```
custom_imports = dict(imports=['mmdet3d.engine.hooks.my_hook'], allow_failed_
↳ imports=False)
```

3. Modify the config

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

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

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

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

Use hooks implemented in MMDetection3D

If the hook is already implemented in MMDetection3D, you can directly modify the config to use the hook as below.

Example: DisableObjectSampleHook

We implement a customized hook named `DisableObjectSampleHook` to disable `ObjectSample` augmentation during training after specified epoch.

We can set it in the config file if needed:

```
custom_hooks = [dict(type='DisableObjectSampleHook', disable_after_epoch=15)]
```

Modify default runtime hooks

There are some common hooks that are registered through `default_hooks`, they are

- `IterTimerHook`: A hook that logs 'data_time' for loading data and 'time' for a model training step.
- `LoggerHook`: A hook that collects logs from different components of `Runner` and writes them to terminal, json file, tensorboard and wandb etc.
- `ParamSchedulerHook`: A hook that updates some hyper-parameters in optimizer, e.g., learning rate and momentum.
- `CheckpointHook`: A hook that saves checkpoints periodically.
- `DistSamplerSeedHook`: A hook that sets the seed for sampler and batch_sampler.
- `Det3DVisualizationHook`: A hook used to visualize validation and testing process prediction results.

`IterTimerHook`, `ParamSchedulerHook` and `DistSamplerSeedHook` are simple and no need to be modified usually, so here we reveal what we can do with `LoggerHook`, `CheckpointHook` and `Det3DVisualizationHook`.

CheckpointHook

Except saving checkpoints periodically, `CheckpointHook` provides other options such as `max_keep_ckpts`, `save_optimizer` and etc. The users could set `max_keep_ckpts` to only save small number of checkpoints or decide whether to store state dict of optimizer by `save_optimizer`. More details of the arguments are [here](#).

```
default_hooks = dict(  
    checkpoint=dict(  
        type='CheckpointHook',  
        interval=1,  
        max_keep_ckpts=3,  
        save_optimizer=True))
```

LoggerHook

The LoggerHook enables setting intervals. Detailed instructions can be found in the [docstring](#).

```
default_hooks = dict(logger=dict(type='LoggerHook', interval=50))
```

Det3DVisualizationHook

Det3DVisualizationHook use DetLocalVisualizer to visualize prediction results, and Det3DLocalVisualizer current supports different backends, e.g., TensorboardVisBackend and WandbVisBackend (see [docstring](#) for more details). The users could add multi backends to do visualization as follows.

```
default_hooks = dict(
    visualization=dict(type='Det3DVisualizationHook', draw=True))

vis_backends = [dict(type='LocalVisBackend'),
                dict(type='TensorboardVisBackend')]

visualizer = dict(
    type='Det3DLocalVisualizer', vis_backends=vis_backends, name='visualizer')
```

Along with the release of OpenMMLab 2.0, MMDetection3D (namely MMDet3D) 1.1 made many significant changes, resulting in less redundant, more efficient code and a more consistent overall design. These changes break backward compatibility. Therefore, we prepared this migration guide to make the transition as smooth as possible so that all users can enjoy the productivity benefits of the new MMDet3D and the entire OpenMMLab 2.0 ecosystem.

ENVIRONMENT

MMDet3D 1.1 depends on the new foundational library [MMEngine](#) for training deep learning models, and therefore has an entirely different dependency chain compared with MMDet3D 1.0. Even if you have a well-rounded MMDet3D 1.0 / 0.x environment before, you still need to create a new Python environment for MMDet3D 1.1. We provide a detailed [installation guide](#) for reference.

The configuration files in our new version have a lot of modifications because of the differences between MMCV 1.x and MMEngine. The guides for migration from MMCV to MMEngine can be seen [here](#).

We have renamed the names of the remote branches in MMDet3D 1.1 (renaming 1.1 to main, master to 1.0, and dev to dev-1.0). If your local branches in the git system are not aligned with branches of the remote repo, you can use the following commands to resolve it:

```
git fetch origin
git checkout main
git branch main_backup # backup your main branch
git reset --hard origin/main
```


DATASET

You should update the annotation files generated in the 1.0 version since some key words and structures of annotation in MMDet3D 1.1 have changed. Taking KITTI as an example, the update script is as follows:

```
python tools/dataset_converters/update_infos_to_v2.py
    --dataset kitti
    --pkl-path ./data/kitti/kitti_infos_train.pkl
    --out-dir ./kitti_v2/
```

If your annotation files are generated in the 0.x version, you should first update them to 1.0 version using this script. Alternatively, you can re-generate annotation files from scratch using this script.

MODEL

MMDet3D 1.1 supports loading weights trained on the old version (1.0 version). For models that are important or frequently used, we have thoroughly verified their precisions in the 1.1 version. Especially for some models that may experience potential performance drop or training bugs in the old version, such as [centerpoint](#), we have checked them and ensured the right precision in the new version. If you encounter any problem, please feel free to raise an [issue](#). Additionally, we have added some of the latest SOTA methods in our package and projects, making MMDet3D 1.1 a highly recommended choice for implementing your project.

MMDET3D.APIS

MMDET3D.DATASETS

20.1 datasets

20.2 transforms

MMDET3D.ENGINE

21.1 hooks

MMDET3D.EVALUATION

22.1 functional

22.2 metrics

CHAPTER
TWENTYTHREE

MMDet3D.MODELS

23.1 backbones

23.2 data_preprocessors

23.3 decode_heads

23.4 dense_heads

23.5 detectors

23.6 layers

23.7 losses

23.8 middle_encoders

23.9 necks

23.10 roi_heads

23.11 segmentors

23.12 task_modules

23.13 test_time_augs

23.14 utils

23.15 voxel_encoders

MMDet3D.Structures

24.1 structures

24.2 bbox_3d

class `mmdet3d.structures.bbox_3d.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** (*BaseInstance3DBBoxes*) – 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 mth point is in the tth box, then $A[m, t] == 1$, otherwise $A[m, t] == 0$.

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*, **args*, ***kwargs*)

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.structures.bbox_3d.Box3DMode(value)`

Enum of different ways to represent a box.

Coordinates in LiDAR:



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:



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:



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, correct_yaw=False*)

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.
- **correct_yaw (bool) –** If the yaw is rotated by *rt_mat*.

Returns

(tuple | list | np.ndarray | torch.Tensor | [BaseInstance3DBoxes](#)): The converted box of the same type.

class mmdet3d.structures.bbox_3d.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:



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, correct_yaw=False)

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.
- **correct_yaw** (*bool*) – Whether to convert the yaw angle to the target coordinate. Defaults to False.

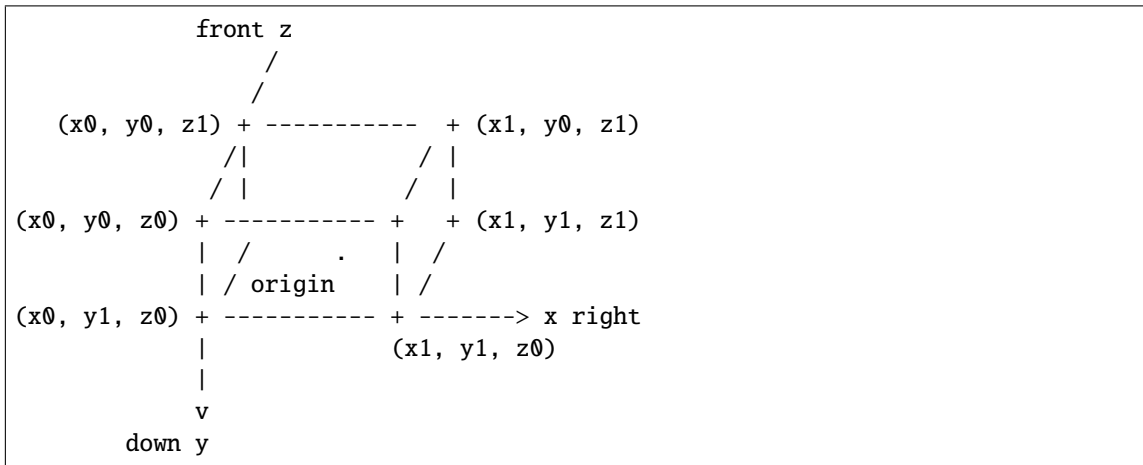
Returns The converted box of the same type in the dst mode.

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 `mmcv3d.structures.bbox_3d.Coord3DMode`(*value*)

Enum of different ways to represent a box and point cloud.

Coordinates in LiDAR:



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:



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:



(continues on next page)

(continued from previous page)

```
| /
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.structures.bbox_3d.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.

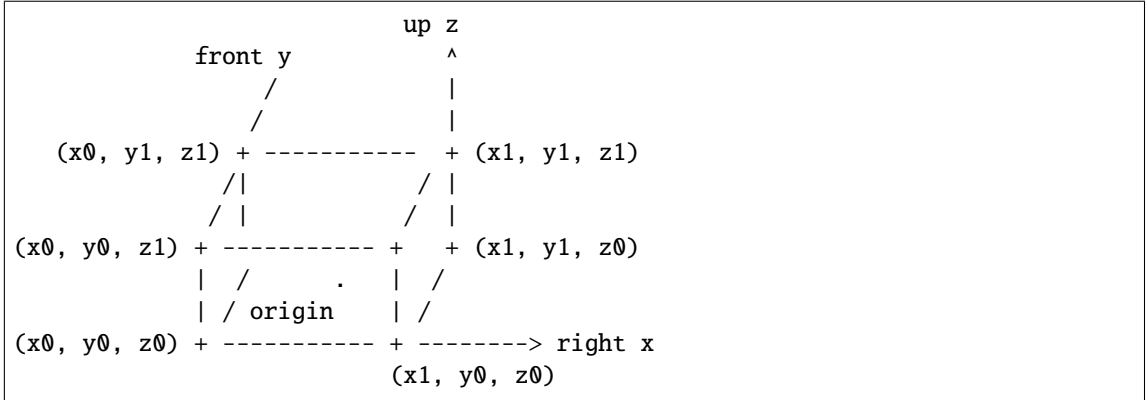
Returns The converted box of the same type in the dst mode.

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

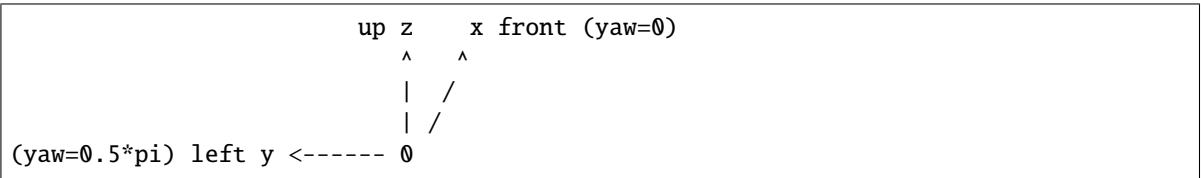
When points is None, the function returns *None*, otherwise it returns the rotated points and the rotation matrix *rot_mat_T*.

Return type tuple or *None*

class `mmdet3d.structures.bbox_3d.LiDARInstance3DBoxes`(*tensor*, *box_dim=7*, *with_yaw=True*, *origin=(0.5, 0.5, 0)*)

3D boxes of instances in LIDAR coordinates.

Coordinates in LiDAR:



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*, *correct_yaw=False*)

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.
- **correct_yaw** (*bool*) – If convert the yaw angle to the target coordinate. Defaults to *False*.

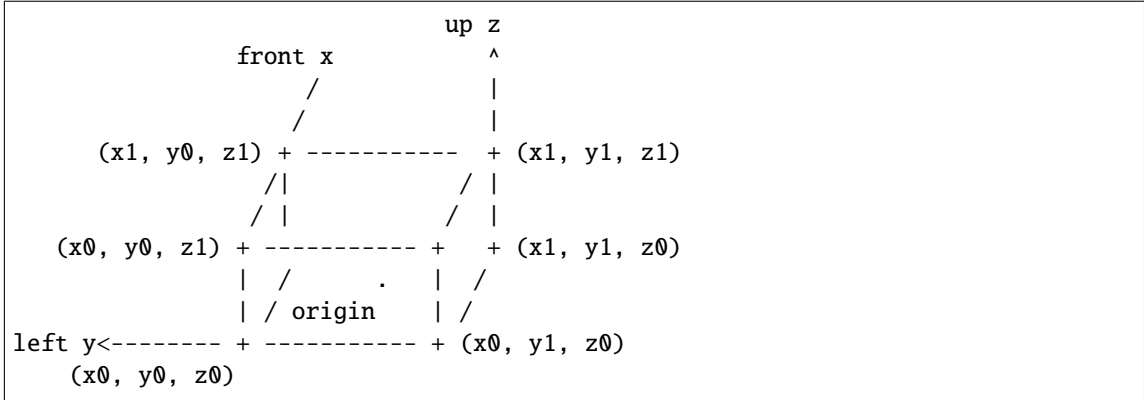
Returns The converted box of the same type in the dst mode.

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

`mmdet3d.structures.bbox_3d.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.structures.bbox_3d.get_proj_mat_by_coord_type(img_meta, coord_type)`

Obtain image features using points.

Parameters

- **img_meta** (*dict*) – Meta info.
- **coord_type** (*str*) – ‘DEPTH’ or ‘CAMERA’ or ‘LIDAR’. Can be case-insensitive.

Returns transformation matrix.

Return type torch.Tensor

`mmdet3d.structures.bbox_3d.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.structures.bbox_3d.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 $\text{np.pi} / 2$ (roll)
2. change orientation from local yaw to global yaw
3. convert yaw by $(\text{np.pi} / 2 - \text{yaw})$

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.structures.bbox_3d.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.structures.bbox_3d.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.structures.bbox_3d.rotation_3d_in_axis(points, angles, axis=0, return_mat=False, clockwise=False)`

Rotate points by angles according to axis.

Parameters

- **points** (*np.ndarray* | *torch.Tensor* | *list* | *tuple*) – Points of shape (N, M, 3).
- **angles** (*np.ndarray* | *torch.Tensor* | *list* | *tuple* | *float*) – Vector of angles in shape (N,)
- **axis** (*int*, *optional*) – The axis to be rotated. Defaults to 0.
- **return_mat** – Whether or not return the rotation matrix (transposed). Defaults to False.
- **clockwise** – Whether the rotation is clockwise. Defaults to False.

Raises **ValueError** – when the axis is not in range [0, 1, 2], it will raise value error.

Returns Rotated points in shape (N, M, 3).

Return type (*torch.Tensor* | *np.ndarray*)

`mmdet3d.structures.bbox_3d.xywhr2xyxyr(bboxes_xywhr)`

Convert a rotated boxes in XYWHR format to XYXYR format.

Parameters **bboxes_xywhr** (*torch.Tensor* | *np.ndarray*) – Rotated boxes in XYWHR format.

Returns Converted boxes in XYXYR format.

Return type (*torch.Tensor* | *np.ndarray*)

24.3 ops

24.4 points

class `mmdet3d.structures.points.BasePoints`(*tensor*, *points_dim*=3, *attribute_dims*=None)

Base class for Points.

Parameters

- **tensor** (*torch.Tensor* / *np.ndarray* / *list*) – a $N \times \text{points_dim}$ matrix.
- **points_dim** (*int*, *optional*) – Number of the dimension of a point. Each row is (x, y, z). Defaults to 3.
- **attribute_dims** (*dict*, *optional*) – Dictionary to indicate the meaning of extra dimension. Defaults to None.

tensor

Float matrix of $N \times \text{points_dim}$.

Type `torch.Tensor`

points_dim

Integer indicating the dimension of a point. Each row is (x, y, z, ...).

Type `int`

attribute_dims

Dictionary to indicate the meaning of extra dimension. Defaults to None.

Type `bool`

rotation_axis

Default rotation axis for points rotation.

Type `int`

property bev

BEV of the points in shape (N, 2).

Type `torch.Tensor`

classmethod `cat`(*points_list*)

Concatenate a list of Points into a single Points.

Parameters **points_list** (list[[BasePoints](#)]) – List of points.

Returns The concatenated Points.

Return type [BasePoints](#)

clone()

Clone the Points.

Returns

Box object with the same properties as self.

Return type [BasePoints](#)

property color

`torch.Tensor`: A vector with color of each point in shape (N, 3), or None.

abstract convert_to(*dst*, *rt_mat=None*)

Convert self to *dst* mode.

Parameters

- **dst** (*CoordMode*) – 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 *BasePoints*

property coord

Coordinates of each point in shape (N, 3).

Type *torch.Tensor*

property device

The device of the points are on.

Type *str*

abstract flip(*bev_direction='horizontal'*)

Flip the points along given BEV direction.

Parameters **bev_direction** (*str*) – Flip direction (horizontal or vertical).

property height

torch.Tensor: A vector with height of each point in shape (N, 1), or None.

in_range_3d(*point_range*)

Check whether the points are in the given range.

Parameters **point_range** (*list | torch.Tensor*) – The range of point (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 point is inside the reference range.

Return type *torch.Tensor*

in_range_bev(*point_range*)

Check whether the points are in the given range.

Parameters **point_range** (*list | torch.Tensor*) – The range of point in order of (x_min, y_min, x_max, y_max).

Returns

Indicating whether each point is inside the reference range.

Return type *torch.Tensor*

new_point(*data*)

Create a new point object with data.

The new point 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 point object with data, the object's other properties are similar to self.

Return type *BasePoints***rotate**(*rotation*, *axis=None*)

Rotate points with the given rotation matrix or angle.

Parameters

- **rotation** (*float* | *np.ndarray* | *torch.Tensor*) – Rotation matrix or angle.
- **axis** (*int*, *optional*) – Axis to rotate at. Defaults to None.

scale(*scale_factor*)

Scale the points with horizontal and vertical scaling factors.

Parameters **scale_factors** (*float*) – Scale factors to scale the points.**property shape**

Shape of points.

Type *torch.Shape***shuffle**()

Shuffle the points.

Returns The shuffled index.**Return type** *torch.Tensor***to**(*device*)

Convert current points 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 *BasePoints***translate**(*trans_vector*)

Translate points with the given translation vector.

Parameters **trans_vector** (*np.ndarray*, *torch.Tensor*) – Translation vector of size 3 or nx3.**class** `mmcv3d.structures.points.CameraPoints`(*tensor*, *points_dim=3*, *attribute_dims=None*)

Points of instances in CAM coordinates.

Parameters

- **tensor** (*torch.Tensor* | *np.ndarray* | *list*) – a N x points_dim matrix.
- **points_dim** (*int*, *optional*) – Number of the dimension of a point. Each row is (x, y, z). Defaults to 3.

- **attribute_dims** (*dict*, *optional*) – Dictionary to indicate the meaning of extra dimension. Defaults to None.

tensor

Float matrix of N x points_dim.

Type torch.Tensor

points_dim

Integer indicating the dimension of a point. Each row is (x, y, z, ...).

Type int

attribute_dims

Dictionary to indicate the meaning of extra dimension. Defaults to None.

Type bool

rotation_axis

Default rotation axis for points rotation.

Type int

property bev

BEV of the points in shape (N, 2).

Type torch.Tensor

convert_to(*dst*, *rt_mat=None*)

Convert self to dst mode.

Parameters

- **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 in the *dst* mode.

Return type *BasePoints*

flip(*bev_direction='horizontal'*)

Flip the points along given BEV direction.

Parameters **bev_direction** (*str*) – Flip direction (horizontal or vertical).

class `mmcv3d.structures.points.DepthPoints`(*tensor*, *points_dim=3*, *attribute_dims=None*)

Points of instances in DEPTH coordinates.

Parameters

- **tensor** (*torch.Tensor* | *np.ndarray* | *list*) – a N x points_dim matrix.
- **points_dim** (*int*, *optional*) – Number of the dimension of a point. Each row is (x, y, z). Defaults to 3.
- **attribute_dims** (*dict*, *optional*) – Dictionary to indicate the meaning of extra dimension. Defaults to None.

tensor

Float matrix of N x points_dim.

Type torch.Tensor

points_dim

Integer indicating the dimension of a point. Each row is (x, y, z, ...).

Type int

attribute_dims

Dictionary to indicate the meaning of extra dimension. Defaults to None.

Type bool

rotation_axis

Default rotation axis for points rotation.

Type int

convert_to(*dst*, *rt_mat=None*)

Convert self to *dst* mode.

Parameters

- **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 in the *dst* mode.

Return type *BasePoints*

flip(*bev_direction='horizontal'*)

Flip the points along given BEV direction.

Parameters **bev_direction** (*str*) – Flip direction (horizontal or vertical).

class mmdet3d.structures.points.LiDARPoints(*tensor*, *points_dim=3*, *attribute_dims=None*)

Points of instances in LIDAR coordinates.

Parameters

- **tensor** (*torch.Tensor* / *np.ndarray* / *list*) – a N x points_dim matrix.
- **points_dim** (*int*, *optional*) – Number of the dimension of a point. Each row is (x, y, z). Defaults to 3.
- **attribute_dims** (*dict*, *optional*) – Dictionary to indicate the meaning of extra dimension. Defaults to None.

tensor

Float matrix of N x points_dim.

Type torch.Tensor

points_dim

Integer indicating the dimension of a point. Each row is (x, y, z, ...).

Type int

attribute_dims

Dictionary to indicate the meaning of extra dimension. Defaults to None.

Type bool

rotation_axis

Default rotation axis for points rotation.

Type `int`

convert_to(*dst*, *rt_mat=None*)

Convert self to *dst* mode.

Parameters

- **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 in the *dst* mode.

Return type `BasePoints`

flip(*bev_direction='horizontal'*)

Flip the points along given BEV direction.

Parameters **bev_direction** (`str`) – Flip direction (horizontal or vertical).

MMDET3D.TESTING

MMDET3D.VISUALIZATION

MMDet3D.UTILS

class `mmdet3d.utils.ArrayConverter`(*template_array: Optional[Union[tuple, list, int, float, numpy.ndarray, torch.Tensor]] = None*)

Utility class for data-type agnostic processing.

Parameters (**tuple** | **list** | **int** | **float** | **np.ndarray** | **torch.Tensor**, optional): *template_array* – Defaults to None.

convert(*input_array: Union[tuple, list, int, float, numpy.ndarray, torch.Tensor]*, *target_type: Optional[type] = None*, *target_array: Optional[Union[numpy.ndarray, torch.Tensor]] = None*) → Union[numpy.ndarray, torch.Tensor]

Convert input array to target data type.

Parameters (**tuple** | **list** | **int** | **float** | **np.ndarray** | **torch.Tensor**): *input_array* – Input array.

:param target_type (**np.ndarray** or **torch.Tensor**: optional): Type to which input array is converted. Defaults to None.

:param [optional]: Type to which input array is converted.] Defaults to None.

Parameters **target_array** (**np.ndarray** | **torch.Tensor**, optional) – Template array to which input array is converted. Defaults to None.

Raises

- **ValueError** – If input is list or tuple and cannot be converted to a NumPy array, a ValueError is raised.
- **TypeError** – If input type does not belong to the above range, or the contents of a list or tuple do not share the same data type, a TypeError is raised.

Returns The converted array.

Return type np.ndarray or torch.Tensor

recover(*input_array: Union[numpy.ndarray, torch.Tensor]*) → Union[numpy.ndarray, torch.Tensor]
Recover input type to original array type.

Parameters **input_array** (**np.ndarray** | **torch.Tensor**) – Input array.

Returns Converted array.

Return type np.ndarray or torch.Tensor

set_template(*array: Union[tuple, list, int, float, numpy.ndarray, torch.Tensor]*) → None
Set template array.

Parameters `array` (`tuple` | `list` | `int` | `float` | `np.ndarray` | `torch.Tensor`) – Template array.

Raises

- **ValueError** – If input is list or tuple and cannot be converted to a NumPy array, a ValueError is raised.
- **TypeError** – If input type does not belong to the above range, or the contents of a list or tuple do not share the same data type, a TypeError is raised.

`mmdet3d.utils.array_converter`(`to_torch`: `bool = True`, `apply_to`: `Tuple[str, ...] = ()`, `template_arg_name`: `Optional[str] = None`, `recover`: `bool = True`) → Callable

Wrapper function for data-type agnostic processing.

First converts input arrays to PyTorch tensors or NumPy ndarrays for middle calculation, then convert output to original data-type if `recover=True`.

Parameters

- **to_torch** (`bool`) – Whether convert to PyTorch tensors for middle calculation. Defaults to True.
- **apply_to** (`Tuple[str, ...]`) – The arguments to which we apply data-type conversion. Defaults to an empty tuple.
- **template_arg_name** (`str`, `optional`) – Argument serving as the template (return arrays should have the same dtype and device as the template). Defaults to None. If None, we will use the first argument in `apply_to` as the template argument.
- **recover** (`bool`) – Whether or not recover the wrapped function outputs to the `template_arg_name` type. Defaults to True.

Raises

- **ValueError** – When `template_arg_name` is not among all args, or when `apply_to` contains an arg which is not among all args, a ValueError will be raised. When the template argument or an argument to convert is a list or tuple, and cannot be converted to a NumPy array, a ValueError will be raised.
- **TypeError** – When the type of the template argument or an argument to convert does not belong to the above range, or the contents of such an list-or-tuple-type argument do not share the same data type, a TypeError is raised.

Returns wrapped function.

Return type (function)

Example

```
>>> import torch
>>> import numpy as np
>>>
>>> # Use torch addition for a + b,
>>> # and convert return values to the type of a
>>> @array_converter(apply_to=('a', 'b'))
>>> def simple_add(a, b):
>>>     return a + b
>>>
>>> a = np.array([1.1])
```

(continues on next page)

(continued from previous page)

```
>>> b = np.array([2.2])
>>> simple_add(a, b)
>>>
>>> # Use numpy addition for a + b,
>>> # and convert return values to the type of b
>>> @array_converter(to_torch=False, apply_to=('a', 'b'),
>>>                  template_arg_name='b')
>>> def simple_add(a, b):
>>>     return a + b
>>>
>>> simple_add()
>>>
>>> # Use torch funcs for floor(a) if flag=True else ceil(a),
>>> # and return the torch tensor
>>> @array_converter(apply_to=('a',), recover=False)
>>> def floor_or_ceil(a, flag=True):
>>>     return torch.floor(a) if flag else torch.ceil(a)
>>>
>>> floor_or_ceil(a, flag=False)
```

`mmdet3d.utils.collect_env()`

Collect the information of the running environments.

`mmdet3d.utils.compat_cfg(cfg)`

This function would modify some filed to keep the compatibility of config.

For example, it will move some args which will be deprecated to the correct fields.

`mmdet3d.utils.register_all_modules(init_default_scope: bool = True) → None`

Register all modules in mmdet3d into the registries.

Parameters `init_default_scope` (*bool*) – Whether initialize the mmdet default scope. When `init_default_scope=True`, the global default scope will be set to `mmdet3d`, and all registries will build modules from mmdet3d’s registry node. To understand more about the registry, please refer to <https://github.com/open-mmlab/mengine/blob/main/docs/en/tutorials/registry.md> Defaults to True.

`mmdet3d.utils.setup_multi_processes(cfg)`

Setup multi-processing environment variables.

28.1 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.

28.2 Baselines

28.2.1 SECOND

Please refer to [SECOND](#) for details. We provide SECOND baselines on KITTI and Waymo datasets.

28.2.2 PointPillars

Please refer to [PointPillars](#) for details. We provide pointpillars baselines on KITTI, nuScenes, Lyft, and Waymo datasets.

28.2.3 Part-A2

Please refer to [Part-A2](#) for details.

28.2.4 VoteNet

Please refer to [VoteNet](#) for details. We provide VoteNet baselines on ScanNet and SUNRGBD datasets.

28.2.5 Dynamic Voxelization

Please refer to [Dynamic Voxelization](#) for details.

28.2.6 MVXNet

Please refer to [MVXNet](#) for details.

28.2.7 RegNetX

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

28.2.8 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.

28.2.9 H3DNet

Please refer to [H3DNet](#) for details.

28.2.10 3DSSD

Please refer to [3DSSD](#) for details.

28.2.11 CenterPoint

Please refer to [CenterPoint](#) for details.

28.2.12 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.

28.2.13 ImVoteNet

Please refer to [ImVoteNet](#) for details. We provide ImVoteNet baselines on SUNRGBD dataset.

28.2.14 FCOS3D

Please refer to [FCOS3D](#) for details. We provide FCOS3D baselines on the nuScenes dataset.

28.2.15 PointNet++

Please refer to [PointNet++](#) for details. We provide PointNet++ baselines on ScanNet and S3DIS datasets.

28.2.16 Group-Free-3D

Please refer to [Group-Free-3D](#) for details. We provide Group-Free-3D baselines on ScanNet dataset.

28.2.17 ImVoxelNet

Please refer to [ImVoxelNet](#) for details. We provide ImVoxelNet baselines on KITTI dataset.

28.2.18 PAConv

Please refer to [PAConv](#) for details. We provide PAConv baselines on S3DIS dataset.

28.2.19 DGCNN

Please refer to [DGCNN](#) for details. We provide DGCNN baselines on S3DIS dataset.

28.2.20 SMOKE

Please refer to [SMOKE](#) for details. We provide SMOKE baselines on KITTI dataset.

28.2.21 PGD

Please refer to [PGD](#) for details. We provide PGD baselines on KITTI and nuScenes dataset.

28.2.22 PointRCNN

Please refer to [PointRCNN](#) for details. We provide PointRCNN baselines on KITTI dataset.

28.2.23 MonoFlex

Please refer to [MonoFlex](#) for details. We provide MonoFlex baselines on KITTI dataset.

28.2.24 SA-SSD

Please refer to [SA-SSD](#) for details. We provide SA-SSD baselines on the KITTI dataset.

28.2.25 FCAF3D

Please refer to [FCAF3D](#) for details. We provide FCAF3D baselines on the ScanNet, S3DIS, and SUN RGB-D datasets.

28.2.26 PV-RCNN

Please refer to [PV-RCNN](#) for details. We provide PV-RCNN baselines on the KITTI dataset.

28.2.27 Mixed Precision (FP16) Training

Please refer to [Mixed Precision \(FP16\) Training on PointPillars](#) for details.

BENCHMARKS

Here we benchmark the training and testing speed of models in MMDetection3D, with some other open source 3D detection codebases.

29.1 Settings

- **Hardware:** 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.

29.2 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 x.

29.3 Details of Comparison

29.3.1 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):

```

29.3.2 VoteNet

- **MMDetection3D**: With release v0.1.0, run

```
./tools/dist_train.sh configs/votenet/votenet_8xb16_sunrgbd-3d.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
@@ -10,6 +10,7 @@ import os
```

(continues on next page)

(continued from previous page)

```

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)

```

29.3.3 PointPillars-car

- **MMDetection3D**: With release v0.1.0, run

```

./tools/dist_train.sh configs/benchmark/hv_pointpillars_secfn_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_spmiddlefhd_rpn2_mghead_syncbn.py --work_dir=
↪$NUSC_CBGS_WORK_DIR
# python -m torch.distributed.launch --nproc_per_node=8 ./tools/train.py examples/
↪second/configs/ lyft_all_vfev3_spmiddlefhd_rpn2_mghead_syncbn.py --work_
↪dir=$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

```

29.3.4 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
```

29.3.5 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
```

29.3.6 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
```


CHANGELOG OF V1.0.X

30.1 v1.0.0 (6/4/2023)

30.1.1 Improvements

- Add BN in FPN to avoid loss Nan in MVXNet (#2282)
- Update s3dis_data_utils.py (#2232)

30.1.2 Bug Fixes

- Fix precision error when using mixed precision on CenterPoint (#2341)
- Replace `np.transpose` with `torch.permute` to speed up (@2273)
- Update links of SECOND checkpoints (#2185)

30.1.3 Contributors

A total of 7 developers contributed to this release. @JingweiZhang12, @ZCMax, @Xiangxu-0103, @vansinhu, @cs1488, @sunjiahao1999, @Ginray

30.2 v1.0.0rc7 (7/1/2023)

30.2.1 Improvements

- Support training and testing on MLU (#2167)

30.2.2 Contributors

A total of 1 developers contributed to this release. @mengpenghui

30.3 v1.0.0rc6 (2/12/2022)

30.3.1 New Features

- Add Projects/ folder and the first example project (#2082)

30.3.2 Improvements

- Update Waymo converter to save storage space (#1759)
- Update model link and performance of CenterPoint (#1916)

30.3.3 Bug Fixes

- Fix GPU memory occupancy problem in PointRCNN (#1928)
- Fix sampling bug in IoUNegPiecewiseSampler (#2018)

30.3.4 Contributors

A total of 6 developers contributed to this release.

@oyel, @zzj403, @VVssssk, @Tai-Wang, @tpoisonooo, @JingweiZhang12, @ZCMax

30.4 v1.0.0rc5 (11/10/2022)

30.4.1 New Features

- Support ImVoxelNet on SUN RGB-D (#1738)

30.4.2 Improvements

- Fix the cross-codebase reference problem in metafile README (#1644)
- Update the Chinese documentation about getting started (#1715)
- Fix docs link and add docs link checker (#1811)

30.4.3 Bug Fixes

- Fix a visualization bug that is potentially triggered by empty prediction labels (#1725)
- Fix point cloud segmentation visualization bug due to wrong parameter passing (#1858)
- Fix Nan loss bug during PointRCNN training (#1874)

30.4.4 Contributors

A total of 9 developers contributed to this release.

@ZwwWayne, @Tai-Wang, @filaPro, @VVsssssk, @ZCMax, @Xiangxu-0103, @holtvogt, @tpoisonooo, @lianqing01

30.5 v1.0.0rc4 (8/8/2022)

30.5.1 Highlights

- Support [FCAF3D](#)

30.5.2 New Features

- Support [FCAF3D](#) (#1547)
- Add the transformation to support multi-camera 3D object detection (#1580)
- Support lift-splat-shoot view transformer (#1598)

30.5.3 Improvements

- Remove the limitation of the maximum number of points during SUN RGB-D preprocessing (#1555)
- Support circle CI (#1647)
- Add mim to extras_require in setup.py (#1560, #1574)
- Update dockerfile package version (#1697)

30.5.4 Bug Fixes

- Flip yaw angle for DepthInstance3DBBoxes.overlaps (#1548, #1556)
- Fix DGCNN configs (#1587)
- Fix bbox head not registered bug (#1625)
- Fix missing objects in S3DIS preprocessing (#1665)
- Fix spconv2.0 model loading bug (#1699)

30.5.5 Contributors

A total of 9 developers contributed to this release.

@Tai-Wang, @ZwwWayne, @filaPro, @lianqing11, @ZCMax, @HuangJunJie2017, @Xiangxu-0103, @ChonghaoSima, @VVsssssk

30.6 v1.0.0rc3 (8/6/2022)

30.6.1 Highlights

- Support [SA-SSD](#)

30.6.2 New Features

- Support [SA-SSD](#) (#1337)

30.6.3 Improvements

- Add Chinese documentation for vision-only 3D detection (#1438)
- Update CenterPoint pretrained models that are compatible with refactored coordinate systems (#1450)
- Configure myst-parser to parse anchor tag in the documentation (#1488)
- Replace markdownlint with mdformat for avoiding installing ruby (#1489)
- Add missing `gt_names` when getting annotation info in Custom3DDataset (#1519)
- Support S3DIS full ceph training (#1542)
- Rewrite the installation and FAQ documentation (#1545)

30.6.4 Bug Fixes

- Fix the incorrect registry name when building RoI extractors (#1460)
- Fix the potential problems caused by the registry scope update when composing pipelines (#1466) and using CocoDataset (#1536)
- Fix the missing selection with `order` in the `box3d_nms` introduced by #1403 (#1479)
- Update the `PointPillars config` to make it consistent with the log (#1486)
- Fix heading anchor in documentation (#1490)
- Fix the compatibility of mmcv in the dockerfile (#1508)
- Make `overwrite_sconv` packaged when building whl (#1516)
- Fix the requirement of mmcv and mmdet (#1537)
- Update configs of PartA2 and support its compatibility with sconv 2.0 (#1538)

30.6.5 Contributors

A total of 13 developers contributed to this release.

@Xiangxu-0103, @ZCMax, @jshilong, @filaPro, @atinfinit, @Tai-Wang, @wenbo-yu, @yi-chen-isuzu, @ZwwWayne, @wchen61, @VVsssssk, @AlexPasqua, @lianqing11

30.7 v1.0.0rc2 (1/5/2022)

30.7.1 Highlights

- Support spconv 2.0
- Support MinkowskiEngine with MinkResNet
- Support training models on custom datasets with only point clouds
- Update Registry to distinguish the scope of built functions
- Replace mmcv.iou3d with a set of bird-eye-view (BEV) operators to unify the operations of rotated boxes

30.7.2 New Features

- Add loader arguments in the configuration files (#1388)
- Support [spconv 2.0](#) when the package is installed. Users can still use spconv 1.x in MMCV with CUDA 9.0 (only cost more memory) without losing the compatibility of model weights between two versions (#1421)
- Support MinkowskiEngine with MinkResNet (#1422)

30.7.3 Improvements

- Add the documentation for model deployment (#1373, #1436)
- Add Chinese documentation of
 - Speed benchmark (#1379)
 - LiDAR-based 3D detection (#1368)
 - LiDAR 3D segmentation (#1420)
 - Coordinate system refactoring (#1384)
- Support training models on custom datasets with only point clouds (#1393)
- Replace mmcv.iou3d with a set of bird-eye-view (BEV) operators to unify the operations of rotated boxes (#1403, #1418)
- Update Registry to distinguish the scope of building functions (#1412, #1443)
- Replace recommonmark with myst_parser for documentation rendering (#1414)

30.7.4 Bug Fixes

- Fix the show pipeline in the `browse_dataset.py` (#1376)
- Fix missing `init` files after coordinate system refactoring (#1383)
- Fix the incorrect yaw in the visualization caused by coordinate system refactoring (#1407)
- Fix `NaiveSyncBatchNorm1d` and `NaiveSyncBatchNorm2d` to support non-distributed cases and more general inputs (#1435)

30.7.5 Contributors

A total of 11 developers contributed to this release.

@ZCMax, @ZwwWayne, @Tai-Wang, @VVssssk, @HanaRo, @JoeyforJoy, @ansonlcy, @filaPro, @jshilong, @Xiangxu-0103, @deleomike

30.8 v1.0.0rc1 (1/4/2022)

30.8.1 Compatibility

- We migrate all the `mmdet3d` ops to `mmcv` and do not need to compile them when installing `mmdet3d`.
- To fix the imprecise timestamp and optimize its saving method, we reformat the point cloud data during Waymo data conversion. The data conversion time is also optimized significantly by supporting parallel processing. Please re-generate KITTI format Waymo data if necessary. See more details in the [compatibility documentation](#).
- We update some of the model checkpoints after the refactor of coordinate systems. Please stay tuned for the release of the remaining model checkpoints.

30.8.2 Highlights

- Migrate all the `mmdet3d` ops to `mmcv`
- Support parallel waymo data converter
- Add ScanNet instance segmentation dataset with metrics
- Better compatibility for windows with CI support, op migration and bug fixes
- Support loading annotations from Ceph

30.8.3 New Features

- Add ScanNet instance segmentation dataset with metrics (#1230)
- Support different random seeds for different ranks (#1321)
- Support loading annotations from Ceph (#1325)
- Support resuming from the latest checkpoint automatically (#1329)
- Add windows CI (#1345)

30.8.4 Improvements

- Update the table format and OpenMMLab project orders in [README.md](#) (#1272, #1283)
- Migrate all the mmdet3d ops to mmcv (#1240, #1286, #1290, #1333)
- Add `with_plane` flag in the KITTI data conversion (#1278)
- Update instructions and links in the documentation (#1300, 1309, #1319)
- Support parallel Waymo dataset converter and ground truth database generator (#1327)
- Add quick installation commands to [getting_started.md](#) (#1366)

30.8.5 Bug Fixes

- Update nuimages configs to use new nms config style (#1258)
- Fix the usage of `np.long` for windows compatibility (#1270)
- Fix the incorrect indexing in `BasePoints` (#1274)
- Fix the incorrect indexing in the `pillar_scatter.forward_single` (#1280)
- Fix unit tests that use GPUs (#1301)
- Fix incorrect feature dimensions in `DynamicPillarFeatureNet` caused by previous upgrading of `PillarFeatureNet` (#1302)
- Remove the `CameraPoints` constraint in `PointSample` (#1314)
- Fix imprecise timestamps saving of Waymo dataset (#1327)

30.8.6 Contributors

A total of 9 developers contributed to this release.

@ZCMax, @ZwwWayne, @wHao-Wu, @Tai-Wang, @wangruohui, @zjwzcx, @Xiangxu-0103, @EdAyers, @hongye-dev, @zhanggefan

30.9 v1.0.0rc0 (18/2/2022)

30.9.1 Compatibility

- We refactor our three coordinate systems to make their rotation directions and origins more consistent, and further remove unnecessary hacks in different datasets and models. Therefore, please re-generate data infos or convert the old version to the new one with our provided scripts. We will also provide updated checkpoints in the next version. Please refer to the [compatibility documentation](#) for more details.
- Unify the camera keys for consistent transformation between coordinate systems on different datasets. The modification changes the key names to `lidar2img`, `depth2img`, `cam2img`, etc., for easier understanding. Customized codes using legacy keys may be influenced.
- The next release will begin to move files of CUDA ops to `MMCV`. It will influence the way to import related functions. We will not break the compatibility but will raise a warning first and please prepare to migrate it.

30.9.2 Highlights

- Support new monocular 3D detectors: [PGD](#), [SMOKE](#), [MonoFlex](#)
- Support a new LiDAR-based detector: [PointRCNN](#)
- Support a new backbone: [DGCNN](#)
- Support 3D object detection on the S3DIS dataset
- Support compilation on Windows
- Full benchmark for PConv on S3DIS
- Further enhancement for documentation, especially on the Chinese documentation

30.9.3 New Features

- Support 3D object detection on the S3DIS dataset (#835)
- Support PointRCNN (#842, #843, #856, #974, #1022, #1109, #1125)
- Support DGCNN (#896)
- Support PGD (#938, #940, #948, #950, #964, #1014, #1065, #1070, #1157)
- Support SMOKE (#939, #955, #959, #975, #988, #999, #1029)
- Support MonoFlex (#1026, #1044, #1114, #1115, #1183)
- Support CPU Training (#1196)

30.9.4 Improvements

- Support point sampling based on distance metric (#667, #840)
- Refactor coordinate systems (#677, #774, #803, #899, #906, #912, #968, #1001)
- Unify camera keys in PointFusion and transformations between different systems (#791, #805)
- Refine documentation (#792, #827, #829, #836, #849, #854, #859, #1111, #1113, #1116, #1121, #1132, #1135, #1185, #1193, #1226)
- Add a script to support benchmark regression (#808)
- Benchmark PConvCUDA on S3DIS (#847)
- Support to download pdf and epub documentation (#850)
- Change the `repeat` setting in Group-Free-3D configs to reduce training epochs (#855)
- Support KITTI AP40 evaluation metric (#927)
- Add the `mmdet3d2torchserve` tool for SECOND (#977)
- Add code-spell pre-commit hook and fix typos (#995)
- Support the latest numba version (#1043)
- Set a default seed to use when the random seed is not specified (#1072)
- Distribute mix-precision models to each algorithm folder (#1074)
- Add abstract and a representative figure for each algorithm (#1086)
- Upgrade pre-commit hook (#1088, #1217)

- Support augmented data and ground truth visualization (#1092)
- Add local yaw property for CameraInstance3DBoxes (#1130)
- Lock the required numba version to 0.53.0 (#1159)
- Support the usage of plane information for KITTI dataset (#1162)
- Deprecate the support for “python setup.py test” (#1164)
- Reduce the number of multi-process threads to accelerate training (#1168)
- Support 3D flip augmentation for semantic segmentation (#1181)
- Update README format for each model (#1195)

30.9.5 Bug Fixes

- Fix compiling errors on Windows (#766)
- Fix the deprecated nms setting in the ImVoteNet config (#828)
- Use the latest `wrap_fp16_model` import from mmcv (#861)
- Remove 2D annotations generation on Lyft (#867)
- Update index files for the Chinese documentation to be consistent with the English version (#873)
- Fix the nested list transpose in the CenterPoint head (#879)
- Fix deprecated pretrained model loading for RegNet (#889)
- Fix the incorrect dimension indices of rotations and testing config in the CenterPoint test time augmentation (#892)
- Fix and improve visualization tools (#956, #1066, #1073)
- Fix PointPillars FLOPs calculation error (#1075)
- Fix missing dimension information in the SUN RGB-D data generation (#1120)
- Fix incorrect anchor range settings in the PointPillars `config` for KITTI (#1163)
- Fix incorrect model information in the RegNet metafile (#1184)
- Fix bugs in non-distributed multi-gpu training and testing (#1197)
- Fix a potential assertion error when generating corners from an empty box (#1212)
- Upgrade bazel version according to the requirement of Waymo Devkit (#1223)

30.9.6 Contributors

A total of 12 developers contributed to this release.

@THU17cyz, @wHao-Wu, @wangruohui, @Wuziyi616, @filaPro, @ZwwWayne, @Tai-Wang, @DCNSW, @xieenze, @robin-karlsson0, @ZCMax, @Otteri

30.10 v0.18.1 (1/2/2022)

30.10.1 Improvements

- Support Flip3D augmentation in semantic segmentation task (#1182)
- Update regnet metafile (#1184)
- Add point cloud annotation tools introduction in FAQ (#1185)
- Add missing explanations of `cam_intrinsic` in the nuScenes dataset doc (#1193)

30.10.2 Bug Fixes

- Deprecate the support for “python setup.py test” (#1164)
- Fix the rotation matrix while rotation axis=0 (#1182)
- Fix the bug in non-distributed multi-gpu training/testing (#1197)
- Fix a potential bug when generating corners for empty bounding boxes (#1212)

30.10.3 Contributors

A total of 4 developers contributed to this release.

@ZwwWayne, @ZCMax, @Tai-Wang, @wHao-Wu

30.11 v0.18.0 (1/1/2022)

30.11.1 Highlights

- Update the required minimum version of mmdet and mmseg

30.11.2 Improvements

- Use the official markdownlint hook and add codespell hook for pre-committing (#1088)
- Improve CI operation (#1095, #1102, #1103)
- Use shared menu content from OpenMMLab’s theme and remove duplicated contents from config (#1111)
- Refactor the structure of documentation (#1113, #1121)
- Update the required minimum version of mmdet and mmseg (#1147)

30.11.3 Bug Fixes

- Fix symlink failure on Windows (#1096)
- Fix the upper bound of mmcv version in the mminstall requirements (#1104)
- Fix API documentation compilation and mmcv build errors (#1116)
- Fix figure links and pdf documentation compilation (#1132, #1135)

30.11.4 Contributors

A total of 4 developers contributed to this release.

@ZwwWayne, @ZCMax, @Tai-Wang, @wHao-Wu

30.12 v0.17.3 (1/12/2021)

30.12.1 Improvements

- Change the default show value to `False` in `show_result` function to avoid unnecessary errors (#1034)
- Improve the visualization of detection results with colored points in `single_gpu_test` (#1050)
- Clean unnecessary `custom_imports` in `entrypoints` (#1068)

30.12.2 Bug Fixes

- Update mmcv version in the Dockerfile (#1036)
- Fix the memory-leak problem when loading checkpoints in `init_model` (#1045)
- Fix incorrect velocity indexing when formatting boxes on nuScenes (#1049)
- Explicitly set cuda device ID in `init_model` to avoid memory allocation on unexpected devices (#1056)
- Fix PointPillars FLOPs calculation error (#1076)

30.12.3 Contributors

A total of 5 developers contributed to this release.

@wHao-Wu, @Tai-Wang, @ZCMax, @MilkClouds, @aldakata

30.13 v0.17.2 (1/11/2021)

30.13.1 Improvements

- Update Group-Free-3D and FCOS3D bibtex (#985)
- Update the solutions for incompatibility of pycocotools in the FAQ (#993)
- Add Chinese documentation for the KITTI (#1003) and Lyft (#1010) dataset tutorial

- Add the H3DNet checkpoint converter for incompatible keys (#1007)

30.13.2 Bug Fixes

- Update mmdetection and mmsegmentation version in the Dockerfile (#992)
- Fix links in the Chinese documentation (#1015)

30.13.3 Contributors

A total of 4 developers contributed to this release.

@Tai-Wang, @wHao-Wu, @ZwwWayne, @ZCMax

30.14 v0.17.1 (1/10/2021)

30.14.1 Highlights

- Support a faster but non-deterministic version of hard voxelization
- Completion of dataset tutorials and the Chinese documentation
- Improved the aesthetics of the documentation format

30.14.2 Improvements

- Add Chinese documentation for training on customized datasets and designing customized models (#729, #820)
- Support a faster but non-deterministic version of hard voxelization (#904)
- Update paper titles and code details for metafiles (#917)
- Add a tutorial for KITTI dataset (#953)
- Use Pytorch sphinx theme to improve the format of documentation (#958)
- Use the docker to accelerate CI (#971)

30.14.3 Bug Fixes

- Fix the sphinx version used in the documentation (#902)
- Fix a dynamic scatter bug that discards the first voxel by mistake when all input points are valid (#915)
- Fix the inconsistent variable names used in the `unit test` for voxel generator (#919)
- Upgrade to use `build_prior_generator` to replace the legacy `build_anchor_generator` (#941)
- Fix a minor bug caused by a too small difference set in the FreeAnchor Head (#944)

30.14.4 Contributors

A total of 8 developers contributed to this release.

@DCNSW, @zhanggefan, @mickeyouyou, @ZCMax, @wHao-Wu, @tojimahammatov, @xiliu8006, @Tai-Wang

30.15 v0.17.0 (1/9/2021)

30.15.1 Compatibility

- Unify the camera keys for consistent transformation between coordinate systems on different datasets. The modification change the key names to `lidar2img`, `depth2img`, `cam2img`, etc. for easier understanding. Customized codes using legacy keys may be influenced.
- The next release will begin to move files of CUDA ops to `MMCV`. It will influence the way to import related functions. We will not break the compatibility but will raise a warning first and please prepare to migrate it.

30.15.2 Highlights

- Support 3D object detection on the S3DIS dataset
- Support compilation on Windows
- Full benchmark for PConv on S3DIS
- Further enhancement for documentation, especially on the Chinese documentation

30.15.3 New Features

- Support 3D object detection on the S3DIS dataset (#835)

30.15.4 Improvements

- Support point sampling based on distance metric (#667, #840)
- Update PointFusion to support unified camera keys (#791)
- Add Chinese documentation for customized dataset (#792), data pipeline (#827), customized runtime (#829), 3D Detection on ScanNet (#836), nuScenes (#854) and Waymo (#859)
- Unify camera keys used in transformation between different systems (#805)
- Add a script to support benchmark regression (#808)
- Benchmark PConvCUDA on S3DIS (#847)
- Add a tutorial for 3D detection on the Lyft dataset (#849)
- Support to download pdf and epub documentation (#850)
- Change the `repeat` setting in Group-Free-3D configs to reduce training epochs (#855)

30.15.5 Bug Fixes

- Fix compiling errors on Windows (#766)
- Fix the deprecated nms setting in the ImVoteNet config (#828)
- Use the latest `wrap_fp16_model` import from mmcv (#861)
- Remove 2D annotations generation on Lyft (#867)
- Update index files for the Chinese documentation to be consistent with the English version (#873)
- Fix the nested list transpose in the CenterPoint head (#879)
- Fix deprecated pretrained model loading for RegNet (#889)

30.15.6 Contributors

A total of 11 developers contributed to this release.

@THU17cyz, @wHao-Wu, @wangruohui, @Wuziyi616, @filaPro, @ZwwWayne, @Tai-Wang, @DCNSW, @xieenze, @robin-karlsson0, @ZCMax

30.16 v0.16.0 (1/8/2021)

30.16.1 Compatibility

- Remove the rotation and dimension hack in the monocular 3D detection on nuScenes by applying corresponding transformation in the pre-processing and post-processing. The modification only influences nuScenes coco-style json files. Please re-run the data preparation scripts if necessary. See more details in the PR #744.
- Add a new pre-processing module for the ScanNet dataset in order to support multi-view detectors. Please run the updated scripts to extract the RGB data and its annotations. See more details in the PR #696.

30.16.2 Highlights

- Support to use [MIM](#) with pip installation
- Support PAAConv [models and benchmarks](#) on S3DIS
- Enhance the documentation especially on dataset tutorials

30.16.3 New Features

- Support RGB images on ScanNet for multi-view detectors (#696)
- Support FLOPs and number of parameters calculation (#736)
- Support to use [MIM](#) with pip installation (#782)
- Support PAAConv models and benchmarks on the S3DIS dataset (#783, #809)

30.16.4 Improvements

- Refactor Group-Free-3D to make it inherit BaseModule from MMCV (#704)
- Modify the initialization methods of FCOS3D to be consistent with the refactored approach (#705)
- Benchmark the Group-Free-3D [models](#) on ScanNet (#710)
- Add Chinese documentation for Getting Started (#725), FAQ (#730), Model Zoo (#735), Demo (#745), Quick Run (#746), Data Preparation (#787) and Configs (#788)
- Add documentation for semantic segmentation on ScanNet and S3DIS (#743, #747, #806, #807)
- Add a parameter `max_keep_ckpts` to limit the maximum number of saved Group-Free-3D checkpoints (#765)
- Add documentation for 3D detection on SUN RGB-D and nuScenes (#770, #793)
- Remove mmpycocotools in the Dockerfile (#785)

30.16.5 Bug Fixes

- Fix versions of OpenMMLab dependencies (#708)
- Convert `rt_mat` to `torch.Tensor` in coordinate transformation for compatibility (#709)
- Fix the `bev_range` initialization in `ObjectRangeFilter` according to the `gt_bboxes_3d` type (#717)
- Fix Chinese documentation and incorrect doc format due to the incompatible Sphinx version (#718)
- Fix a potential bug when setting `interval == 1` in `analyze_logs.py` (#720)
- Update the structure of Chinese documentation (#722)
- Fix FCOS3D FPN BC-Breaking caused by the code refactoring in MMDetection (#739)
- Fix wrong `in_channels` when `with_distance=True` in the [Dynamic VFE Layers](#) (#749)
- Fix the dimension and yaw hack of FCOS3D on nuScenes (#744, #794, #795, #818)
- Fix the missing default `bbox_mode` in the `show_multi_modality_result` (#825)

30.16.6 Contributors

A total of 12 developers contributed to this release.

@yinchimaoliang, @gopi231091, @filaPro, @ZwwWayne, @ZCMax, @hjin2902, @wHao-Wu, @Wuziyi616, @xiliu8006, @THU17cyz, @DCNSW, @Tai-Wang

30.17 v0.15.0 (1/7/2021)

30.17.1 Compatibility

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](#).

30.17.2 Highlights

- Support [PACnv](#)
- Support monocular/multi-view 3D detector [ImVoxelNet](#) on KITTI
- Support Transformer-based 3D detection method [Group-Free-3D](#) on ScanNet
- Add documentation for tasks including LiDAR-based 3D detection, vision-only 3D detection and point-based 3D semantic segmentation
- Add dataset documents like ScanNet

30.17.3 New Features

- Support Group-Free-3D on ScanNet (#539)
- Support PACnv modules (#598, #599)
- Support ImVoxelNet on KITTI (#627, #654)

30.17.4 Improvements

- Add unit tests for pipeline functions `LoadImageFromFileMono3D`, `ObjectNameFilter` and `ObjectRangeFilter` (#615)
- Enhance [IndoorPatchPointSample](#) (#617)
- Refactor model initialization methods based MMCV (#622)
- Add Chinese docs (#629)
- Add documentation for LiDAR-based 3D detection (#642)
- Unify intrinsic and extrinsic matrices for all datasets (#653)
- Add documentation for point-based 3D semantic segmentation (#663)
- Add documentation of ScanNet for 3D detection (#664)
- Refine docs for tutorials (#666)
- Add documentation for vision-only 3D detection (#669)
- Refine docs for Quick Run and Useful Tools (#686)

30.17.5 Bug Fixes

- Fix the bug of [BackgroundPointsFilter](#) using the bottom center of ground truth (#609)
- Fix [LoadMultiViewImageFromFiles](#) to unravel stacked multi-view images to list to be consistent with `DefaultFormatBundle` (#611)
- Fix the potential bug in [analyze_logs](#) when the training resumes from a checkpoint or is stopped before evaluation (#634)
- Fix test commands in docs and make some refinements (#635)
- Fix wrong config paths in unit tests (#641)

30.18 v0.14.0 (1/6/2021)

30.18.1 Highlights

- Support the point cloud segmentation method [PointNet++](#)

30.18.2 New Features

- Support PointNet++ (#479, #528, #532, #541)
- Support RandomJitterPoints transform for point cloud segmentation (#584)
- Support RandomDropPointsColor transform for point cloud segmentation (#585)

30.18.3 Improvements

- Move the point alignment of ScanNet from data pre-processing to pipeline (#439, #470)
- Add compatibility document to provide detailed descriptions of BC-breaking changes (#504)
- Add MMSegmentation installation requirement (#535)
- Support points rotation even without bounding box in GlobalRotScaleTrans for point cloud segmentation (#540)
- Support visualization of detection results and dataset browse for nuScenes Mono-3D dataset (#542, #582)
- Support faster implementation of KNN (#586)
- Support RegNetX models on Lyft dataset (#589)
- Remove a useless parameter `label_weight` from segmentation datasets including `Custom3DSegDataset`, `ScanNetSegDataset` and `S3DISSegDataset` (#607)

30.18.4 Bug Fixes

- Fix a corrupted lidar data file in Lyft dataset in [data_preparation](#) (#546)
- Fix evaluation bugs in nuScenes and Lyft dataset (#549)
- Fix converting points between coordinates with specific transformation matrix in the [coord_3d_mode.py](#) (#556)
- Support PointPillars models on Lyft dataset (#578)
- Fix the bug of demo with pre-trained VoteNet model on ScanNet (#600)

30.19 v0.13.0 (1/5/2021)

30.19.1 Highlights

- Support a monocular 3D detection method [FCOS3D](#)
- Support ScanNet and S3DIS semantic segmentation dataset
- Enhancement of visualization tools for dataset browsing and demos, including support of visualization for multi-modality data and point cloud segmentation.

30.19.2 New Features

- Support ScanNet semantic segmentation dataset (#390)
- Support monocular 3D detection on nuScenes (#392)
- Support multi-modality visualization (#405)
- Support nuimages visualization (#408)
- Support monocular 3D detection on KITTI (#415)
- Support online visualization of semantic segmentation results (#416)
- Support ScanNet test results submission to online benchmark (#418)
- Support S3DIS data pre-processing and dataset class (#433)
- Support FCOS3D (#436, #442, #482, #484)
- Support dataset browse for multiple types of datasets (#467)
- Adding paper-with-code (PWC) metafile for each model in the model zoo (#485)

30.19.3 Improvements

- Support dataset browsing for SUNRGBD, ScanNet or KITTI points and detection results (#367)
- Add the pipeline to load data using file client (#430)
- Support to customize the type of runner (#437)
- Make pipeline functions process points and masks simultaneously when sampling points (#444)
- Add waymo unit tests (#455)
- Split the visualization of projecting points onto image from that for only points (#480)
- Efficient implementation of PointSegClassMapping (#489)
- Use the new model registry from mmcv (#495)

30.19.4 Bug Fixes

- Fix Pytorch 1.8 Compilation issue in the `scatter_points_cuda.cu` (#404)
- Fix `dynamic_scatter` errors triggered by empty point input (#417)
- Fix the bug of missing points caused by using break incorrectly in the voxelization (#423)
- Fix the missing `coord_type` in the waymo dataset `config` (#441)
- Fix errors in four unittest functions of `configs`, `test_detectors.py`, `test_heads.py` (#453)
- Fix 3DSSD training errors and simplify configs (#462)
- Clamp 3D votes projections to image boundaries in ImVoteNet (#463)
- Update out-of-date names of pipelines in the `config` of pointpillars benchmark (#474)
- Fix the lack of a placeholder when unpacking RPN targets in the `h3d_bbox_head.py` (#508)
- Fix the incorrect value of K when creating pickle files for SUN RGB-D (#511)

30.20 v0.12.0 (1/4/2021)

30.20.1 Highlights

- Support a new multi-modality method [ImVoteNet](#).
- Support PyTorch 1.7 and 1.8
- Refactor the structure of tools and [train.py/test.py](#)

30.20.2 New Features

- Support LiDAR-based semantic segmentation metrics (#332)
- Support [ImVoteNet](#) (#352, #384)
- Support the KNN GPU operation (#360, #371)

30.20.3 Improvements

- Add FAQ for common problems in the documentation (#333)
- Refactor the structure of tools (#339)
- Refactor [train.py](#) and [test.py](#) (#343)
- Support demo on nuScenes (#353)
- Add 3DSSD checkpoints (#359)
- Update the Bibtex of CenterPoint (#368)
- Add citation format and reference to other OpenMMLab projects in the README (#374)
- Upgrade the mmcv version requirements (#376)
- Add numba and numpy version requirements in FAQ (#379)
- Avoid unnecessary for-loop execution of vfe layer creation (#389)
- Update SUNRGBD dataset documentation to stress the requirements for training [ImVoteNet](#) (#391)
- Modify vote head to support 3DSSD (#396)

30.20.4 Bug Fixes

- Fix missing keys `coord_type` in database sampler config (#345)
- Rename H3DNet configs (#349)
- Fix CI by using ubuntu 18.04 in github workflow (#350)
- Add assertions to avoid 4-dim points being input to [points_in_boxes](#) (#357)
- Fix the SECOND results on Waymo in the corresponding [README](#) (#363)
- Fix the incorrect adopted pipeline when adding val to workflow (#370)
- Fix a potential bug when indices used in the backwarding in ThreeNN (#377)
- Fix a compilation error triggered by [scatter_points_cuda.cu](#) in PyTorch 1.7 (#393)

30.21 v0.11.0 (1/3/2021)

30.21.1 Highlights

- Support more friendly visualization interfaces based on open3d
- Support a faster and more memory-efficient implementation of DynamicScatter
- Refactor unit tests and details of configs

30.21.2 New Features

- Support new visualization methods based on open3d (#284, #323)

30.21.3 Improvements

- Refactor unit tests (#303)
- Move the key `train_cfg` and `test_cfg` into the model configs (#307)
- Update [README](#) with [Chinese version](#) and [instructions for getting started](#). (#310, #316)
- Support a faster and more memory-efficient implementation of DynamicScatter (#318, #326)

30.21.4 Bug Fixes

- Fix an unsupported bias setting in the unit test for centerpoint head (#304)
- Fix errors due to typos in the centerpoint head (#308)
- Fix a minor bug in `points_in_boxes.py` when tensors are not in the same device. (#317)
- Fix warning of deprecated usages of nonzero during training with PyTorch 1.6 (#330)

30.22 v0.10.0 (1/2/2021)

30.22.1 Highlights

- Preliminary release of API for SemanticKITTI dataset.
- Documentation and demo enhancement for better user experience.
- Fix a number of underlying minor bugs and add some corresponding important unit tests.

30.22.2 New Features

- Support SemanticKITTI dataset preliminarily (#287)

30.22.3 Improvements

- Add tag to README in configurations for specifying different uses (#262)
- Update instructions for evaluation metrics in the documentation (#265)
- Add nuImages entry in [README.md](#) and gif demo (#266, #268)
- Add unit test for voxelization (#275)

30.22.4 Bug Fixes

- Fixed the issue of unpacking size in [furthest_point_sample.py](#) (#248)
- Fix bugs for 3DSSD triggered by empty ground truths (#258)
- Remove models without checkpoints in model zoo statistics of documentation (#259)
- Fix some unclear installation instructions in [getting_started.md](#) (#269)
- Fix relative paths/links in the documentation (#271)
- Fix a minor bug in [scatter_points_cuda.cu](#) when num_features != 4 (#275)
- Fix the bug about missing text files when testing on KITTI (#278)
- Fix issues caused by inplace modification of tensors in `BaseInstance3DBBoxes` (#283)
- Fix log analysis for evaluation and adjust the documentation accordingly (#285)

30.23 v0.9.0 (31/12/2020)

30.23.1 Highlights

- Documentation refactoring with better structure, especially about how to implement new models and customized datasets.
- More compatible with refactored point structure by bug fixes in ground truth sampling.

30.23.2 Improvements

- Documentation refactoring (#242)

30.23.3 Bug Fixes

- Fix point structure related bugs in ground truth sampling (#211)
- Fix loading points in ground truth sampling augmentation on nuScenes (#221)
- Fix channel setting in the SeparateHead of CenterPoint (#228)
- Fix evaluation for indoors 3D detection in case of less classes in prediction (#231)
- Remove unreachable lines in nuScenes data converter (#235)
- Minor adjustments of numpy implementation for perspective projection and prediction filtering criterion in KITTI evaluation (#241)

30.24 v0.8.0 (30/11/2020)

30.24.1 Highlights

- Refactor points structure with more constructive and clearer implementation.
- Support axis-aligned IoU loss for VoteNet with better performance.
- Update and enhance [SECOND](#) benchmark on Waymo.

30.24.2 New Features

- Support axis-aligned IoU loss for VoteNet. (#194)
- Support points structure for consistent processing of all the point related representation. (#196, #204)

30.24.3 Improvements

- Enhance [SECOND](#) benchmark on Waymo with stronger baselines. (#205)
- Add model zoo statistics and polish the documentation. (#201)

30.25 v0.7.0 (1/11/2020)

30.25.1 Highlights

- Support a new method [SSN](#) with benchmarks on nuScenes and Lyft datasets.
- Update benchmarks for [SECOND](#) on Waymo, CenterPoint with TTA on nuScenes and models with mixed precision training on KITTI and nuScenes.
- Support semantic segmentation on nuImages and provide [HTC](#) models with configurations and performance for reference.

30.25.2 New Features

- Modified primitive head which can support the setting on SUN-RGBD dataset (#136)
- Support semantic segmentation and [HTC](#) with models for reference on nuImages dataset (#155)
- Support [SSN](#) on nuScenes and Lyft datasets (#147, #174, #166, #182)
- Support double flip for test time augmentation of CenterPoint with updated benchmark (#143)

30.25.3 Improvements

- Update [SECOND](#) benchmark with configurations for reference on Waymo (#166)
- Delete checkpoints on Waymo to comply its specific license agreement (#180)
- Update models and instructions with [mixed precision training](#) on KITTI and nuScenes (#178)

30.25.4 Bug Fixes

- Fix incorrect code weights in anchor3d_head when introducing mixed precision training (#173)
- Fix the incorrect label mapping on nuImages dataset (#155)

30.26 v0.6.1 (11/10/2020)

30.26.1 Highlights

- Support mixed precision training of voxel-based methods
- Support docker with PyTorch 1.6.0
- Update baseline configs and results ([CenterPoint](#) on nuScenes and [PointPillars](#) on Waymo with full dataset)
- Switch model zoo to download.openmmlab.com

30.26.2 New Features

- Support dataset pipeline `VoxelBasedPointSampler` to sample multi-sweep points based on voxelization. (#125)
- Support mixed precision training of voxel-based methods (#132)
- Support docker with PyTorch 1.6.0 (#160)

30.26.3 Improvements

- Reduce requirements for the case exclusive of Waymo (#121)
- Switch model zoo to download.openmmlab.com (#126)
- Update docs related to Waymo (#128)
- Add version assertion in the `init` file (#129)
- Add evaluation interval setting for CenterPoint (#131)
- Add unit test for CenterPoint (#133)
- Update [PointPillars](#) baselines on Waymo with full dataset (#142)
- Update [CenterPoint](#) results with models and logs (#154)

30.26.4 Bug Fixes

- Fix a bug of visualization in multi-batch case (#120)
- Fix bugs in dcn unit test (#130)
- Fix dcn bias bug in centerpoint (#137)
- Fix dataset mapping in the evaluation of nuScenes mini dataset (#140)
- Fix origin initialization in `CameraInstance3DBBoxes` (#148, #150)
- Correct documentation link in the `getting_started.md` (#159)
- Fix model save path bug in `gather_models.py` (#153)
- Fix image padding shape bug in `PointFusion` (#162)

30.27 v0.6.0 (20/9/2020)

30.27.1 Highlights

- Support new methods [H3DNet](#), [3DSSD](#), [CenterPoint](#).
- Support new dataset [Waymo](#) (with [PointPillars](#) baselines) and [nuImages](#) (with Mask R-CNN and Cascade Mask R-CNN baselines).
- Support Batch Inference
- Support Pytorch 1.6
- Start to publish `mmdet3d` package to PyPI since v0.5.0. You can use `mmdet3d` through `pip install mmdet3d`.

30.27.2 Backwards Incompatible Changes

- Support Batch Inference (#95, #103, #116): MMDetection3D v0.6.0 migrates to support batch inference based on MMDetection >= v2.4.0. This change influences all the test APIs in MMDetection3D and downstream code-bases.
- Start to use collect environment function from MMCV (#113): MMDetection3D v0.6.0 migrates to use `collect_env` function in MMCV. `get_compiler_version` and `get_compiling_cuda_version` compiled in `mm3d.ops.utils` are removed. Please import these two functions from `mmcv.ops`.

30.27.3 New Features

- Support `nuImages` dataset by converting them into coco format and release Mask R-CNN and Cascade Mask R-CNN baseline models (#91, #94)
- Support to publish to PyPI in github-action (#17, #19, #25, #39, #40)
- Support CBGSDataset and make it generally applicable to all the supported datasets (#75, #94)
- Support `H3DNet` and release models on ScanNet dataset (#53, #58, #105)
- Support Fusion Point Sampling used in `3DSSD` (#66)
- Add `BackgroundPointsFilter` to filter background points in data pipeline (#84)
- Support pointnet2 with multi-scale grouping in backbone and refactor pointnets (#82)
- Support dilated ball query used in `3DSSD` (#96)
- Support `3DSSD` and release models on KITTI dataset (#83, #100, #104)
- Support `CenterPoint` and release models on nuScenes dataset (#49, #92)
- Support `Waymo` dataset and release PointPillars baseline models (#118)
- Allow `LoadPointsFromMultiSweeps` to pad empty sweeps and select multiple sweeps randomly (#67)

30.27.4 Improvements

- Fix all warnings and bugs in PyTorch 1.6.0 (#70, #72)
- Update issue templates (#43)
- Update unit tests (#20, #24, #30)
- Update documentation for using ply format point cloud data (#41)
- Use points loader to load point cloud data in ground truth (GT) samplers (#87)
- Unify version file of OpenMMLab projects by using `version.py` (#112)
- Remove unnecessary data preprocessing commands of SUN RGB-D dataset (#110)

30.27.5 Bug Fixes

- Rename CosineAnealing to CosineAnnealing (#57)
- Fix device inconsistent bug in 3D IoU computation (#69)
- Fix a minor bug in json2csv of lyft dataset (#78)
- Add missed test data for pointnet modules (#85)
- Fix use_valid_flag bug in CustomDataset (#106)

30.28 v0.5.0 (9/7/2020)

MMDetection3D is released.

CHANGELOG OF V1.1

31.1 v1.1.0 (6/4/2023)

31.1.1 Highlights

- Support [Cylinder3D](#) (#2291, #2344, #2350)
- Support [MinkUnet](#) (#2294, #2358)
- Support [SPVCNN](#) (#2320#2372)
- Support [TR3D](#) detector in projects (#2274)
- Support the inference of [BEVFusion](#) in projects (#2175)
- Support [DETR3D](#) in projects (#2173)

31.1.2 New Features

- Support [Cylinder3D](#) (#2291, #2344, #2350)
- Support [MinkUnet](#) (#2294, #2358)
- Support [SPVCNN](#) (#2320#2372)
- Support [TR3D](#) detector in projects (#2274)
- Support the inference of [BEVFusion](#) in projects (#2175)
- Support [DETR3D](#) in projects (#2173)
- Support PolarMix and LaserMix augmentation (#2265, #2302)
- Support loading annotation of panoptic segmentation (#2223)
- Support panoptic segmentation metric (#2230)
- Add inferencer for LiDAR-based, monocular and multi-modality 3D detection (#2208, #2190, #2342)
- Add inferencer for LiDAR-based segmentation (#2304)

31.1.3 Improvements

- Support `lazy_init` for `CBGSDataset` (#2271)
- Support generating annotation files for test set on Waymo (#2180)
- Enhance the support for `SemanticKitti` (#2253, #2323)
- File I/O migration and reconstruction (#2319)
- Support `format_only` option for Lyft, NuScenes and Waymo datasets (#2333, #2151)
- Replace `np.transpose` with `torch.permute` to speed up (#2277)
- Allow setting local-rank for pytorch 2.0 (#2387)

31.1.4 Bug Fixes

- Fix the problem of reversal of length and width when drawing heatmap in `CenterFormer` (#2362)
- Deprecate old type alias due to the new version of numpy (#2339)
- Lose `trimesh` version requirements to fix numpy random state (#2340)
- Fix the device mismatch error in `CenterPoint` (#2308)
- Fix bug of visualization when there are no bboxes (#2231)
- Fix bug of counting ignore index in IOU in segmentation evaluation (#2229)

31.1.5 Contributors

A total of 14 developers contributed to this release.

@ZLTJohn, @SekiroRong, @shufanwu, @vansin, @triple-Mu, @404Vector, @filaPro, @sunjiahao1999, @Ginray, @Xiangxu-0103, @JingweiZhang12, @DezeZhao, @ZCMax, @roger-lcc

31.2 v1.1.0rc3 (7/1/2023)

31.2.1 Highlights

- Support `CenterFormer` in projects (#2175)
- Support `PETR` in projects (#2173)

31.2.2 New Features

- Support `CenterFormer` in projects (#2175)
- Support `PETR` in projects (#2173)
- Refactor `ImVoxelNet` on SUN RGB-D into `mmdet3d v1.1` (#2141)

31.2.3 Improvements

- Remove legacy builder.py (#2061)
- Update customize_dataset documentation (#2153)
- Update tutorial of LiDAR-based detection (#2120)

31.2.4 Bug Fixes

- Fix the configs of FCOS3D and PGD (#2191)
- Fix numpy's ValueError in update_infos_to_v2.py (#2162)
- Fix parameter missing in Det3DVisualizationHook (#2118)
- Fix memory overflow in the rotated box IoU calculation (#2134)
- Fix lidar2cam error in update_infos_to_v2.py for nus and lyft dataset (#2110)
- Fix error of data type in Waymo metrics (#2109)
- Update bbox_3d information in cam_instances for mono3d detection task (#2046)
- Fix label saving of Waymo dataset (#2096)

31.2.5 Contributors

A total of 10 developers contributed to this release.

@SekiroRong, @ZLTJohn, @vansin, @shanmo, @VVsssssk, @ZCMax, @Xiangxu-0103, @JingweiZhang12, @Tai-Wang, @lianqing11

31.3 v1.1.0rc2 (2/12/2022)

31.3.1 Highlights

- Support [PV-RCNN](#)
- Speed up evaluation on Waymo dataset

31.3.2 New Features

- Support [PV-RCNN](#) (#1597, #2045)
- Speed up evaluation on Waymo dataset (#2008)
- Refactor FCAF3D into the framework of mmdet3d v1.1 (#1945)
- Refactor S3DIS dataset into the framework of mmdet3d v1.1 (#1984)
- Add Projects/ folder and the first example project (#2042)

31.3.3 Improvements

- Rename CLASSES and PALETTE to `classes` and `palette` respectively (#1932)
- Update `metainfo` in `pkl` files and add `categories` into `metainfo` (#1934)
- Show instance statistics before and after through the pipeline (#1863)
- Add configs of DGCNN for different testing areas (#1967)
- Remove testing utils from `tests/utils/` to `mmdet3d/testing/` (#2012)
- Add typehint for code in `models/layers/` (#2014)
- Refine documentation (#1891, #1994)
- Refine voxelization for better speed (#2062)

31.3.4 Bug Fixes

- Fix loop visualization error about point cloud (#1914)
- Fix image conversion of Waymo to avoid information loss (#1979)
- Fix evaluation on KITTI testset (#2005)
- Fix sampling bug in `IoUNegPiecewiseSampler` (#2017)
- Fix point cloud range in `CenterPoint` (#1998)
- Fix some loading bugs and support FOV-image-based mode on Waymo dataset (#1942)
- Fix dataset conversion utils (#1923, #2040, #1971)
- Update metafiles in all the configs (#2006)

31.3.5 Contributors

A total of 12 developers contributed to this release.

@vavanade, @oyel, @thinkthinking, @PeterH0323 @274869388, @cxiang26, @lianqing11, @VVsssssk, @ZCMax, @Xiangxu-0103, @JingweiZhang12, @Tai-Wang

31.4 v1.1.0rc1 (11/10/2022)

31.4.1 Highlights

- Support a camera-only 3D detection baseline on Waymo, [MV-FCOS3D++](#)

31.4.2 New Features

- Support a camera-only 3D detection baseline on Waymo, [MV-FCOS3D++](#), with new evaluation metrics and transformations (#1716)
- Refactor PointRCNN in the framework of mmdet3d v1.1 (#1819)

31.4.3 Improvements

- Add `auto_scale_lr` in config to support training with auto-scale learning rates (#1807)
- Fix CI (#1813, #1865, #1877)
- Update `browse_dataset.py` script (#1817)
- Update SUN RGB-D and Lyft datasets documentation (#1833)
- Rename `convert_to_datasample` to `add_pred_to_datasample` in detectors (#1843)
- Update customized dataset documentation (#1845)
- Update `Det3DLocalVisualization` and visualization documentation (#1857)
- Add the code of generating `cam_sync_labels` for Waymo dataset (#1870)
- Update dataset transforms typehints (#1875)

31.4.4 Bug Fixes

- Fix missing registration of models in `setup_env.py` (#1808)
- Fix the data base sampler bugs when using the ground plane data (#1812)
- Add output directory existing check during visualization (#1828)
- Fix bugs of nuScenes dataset for monocular 3D detection (#1837)
- Fix visualization hook to support the visualization of different data modalities (#1839)
- Fix monocular 3D detection demo (#1864)
- Fix the lack of `num_pts_feats` key in nusenes dataset and complete docstring (#1882)

31.4.5 Contributors

A total of 10 developers contributed to this release.

@ZwwWayne, @Tai-Wang, @lianqing11, @VVsssssk, @ZCMax, @Xiangxu-0103, @JingweiZhang12, @tpoi-sonooo, @ice-tong, @jshilong

31.5 v1.1.0rc0 (1/9/2022)

We are excited to announce the release of MMDetection3D 1.1.0rc0. MMDet3D 1.1.0rc0 is the first version of MMDetection3D 1.1, a part of the OpenMMLab 2.0 projects. Built upon the new [training engine](#) and [MMDet 3.x](#), MMDet3D 1.1 unifies the interfaces of dataset, models, evaluation, and visualization with faster training and testing speed. It also provides a standard data protocol for different datasets, modalities, and tasks for 3D perception. We will support more strong baselines in the future release, with our latest exploration on camera-only 3D detection from videos.

31.6 Highlights

1. **New engines.** MMDet3D 1.1 is based on [MMEngine](#) and [MMDet 3.x](#), which provides a universal and powerful runner that allows more flexible customizations and significantly simplifies the entry points of high-level interfaces.
2. **Unified interfaces.** As a part of the OpenMMLab 2.0 projects, MMDet3D 1.1 unifies and refactors the interfaces and internal logics of train, testing, datasets, models, evaluation, and visualization. All the OpenMMLab 2.0 projects share the same design in those interfaces and logics to allow the emergence of multi-task/modality algorithms.
3. **Standard data protocol for all the datasets, modalities, and tasks for 3D perception.** Based on the unified base datasets inherited from MMEngine, we also design a standard data protocol that defines and unifies the common keys across different datasets, tasks, and modalities. It significantly simplifies the usage of multiple datasets and data modalities for multi-task frameworks and eases dataset customization. Please refer to the [documentation of customized datasets](#) for details.
4. **Strong baselines.** We will release strong baselines of many popular models to enable fair comparisons among state-of-the-art models.
5. **More documentation and tutorials.** We add a bunch of documentation and tutorials to help users get started more smoothly. Read it [here](#).

31.7 Breaking Changes

MMDet3D 1.1 has undergone significant changes to have better design, higher efficiency, more flexibility, and more unified interfaces. Besides the changes of API, we briefly list the major breaking changes in this section. We will update the [migration guide](#) to provide complete details and migration instructions. Users can also refer to the [compatibility documentation](#) and [API doc](#) for more details.

31.7.1 Dependencies

- MMDet3D 1.1 runs on PyTorch \geq 1.6. We have deprecated the support of PyTorch 1.5 to embrace the mixed precision training and other new features since PyTorch 1.6. Some models can still run on PyTorch 1.5, but the full functionality of MMDet3D 1.1 is not guaranteed.
- MMDet3D 1.1 relies on MMEngine to run. MMEngine is a new foundational library for training deep learning models of OpenMMLab and are widely depended by OpenMMLab 2.0 projects. The dependencies of file IO and training are migrated from MMCV 1.x to MMEngine.
- MMDet3D 1.1 relies on MMCV \geq 2.0.0rc0. Although MMCV no longer maintains the training functionalities since 2.0.0rc0, MMDet3D 1.1 relies on the data transforms, CUDA operators, and image processing interfaces in MMCV. Note that the package `mmcv` is the version that provides pre-built CUDA operators and `mmcv-lite` does not since MMCV 2.0.0rc0, while `mmcv-full` has been deprecated since 2.0.0rc0.

- MMDet3D 1.1 is based on MMDet 3.x, which is also a part of OpenMMLab 2.0 projects.

31.7.2 Training and testing

- MMDet3D 1.1 uses Runner in [MMEngine](#) rather than that in MMCV. The new Runner implements and unifies the building logic of dataset, model, evaluation, and visualizer. Therefore, MMDet3D 1.1 no longer relies on the building logics of those modules in `mmdet3d.train.apis` and `tools/train.py`. Those code have been migrated into [MMEngine](#). Please refer to the [migration guide of Runner in MMEngine](#) for more details.
- The Runner in MMEngine also supports testing and validation. The testing scripts are also simplified, which has similar logic as that in training scripts to build the runner.
- The execution points of hooks in the new Runner have been enriched to allow more flexible customization. Please refer to the [migration guide of Hook in MMEngine](#) for more details.
- Learning rate and momentum scheduling has been migrated from Hook to [Parameter Scheduler in MMEngine](#). Please refer to the [migration guide of Parameter Scheduler in MMEngine](#) for more details.

31.7.3 Configs

- The [Runner in MMEngine](#) uses a different config structure to ease the understanding of the components in runner. Users can read the [config example of MMDet3D 1.1](#) or refer to the [migration guide in MMEngine](#) for migration details.
- The file names of configs and models are also refactored to follow the new rules unified across OpenMMLab 2.0 projects. The names of checkpoints are not updated for now as there is no BC-breaking of model weights between MMDet3D 1.1 and 1.0.x. We will progressively replace all the model weights by those trained in MMDet3D 1.1. Please refer to the [user guides of config](#) for more details.

31.7.4 Dataset

The Dataset classes implemented in MMDet3D 1.1 all inherits from the `Det3DDataset` and `Seg3DDataset`, which inherits from the [BaseDataset in MMEngine](#). In addition to the changes of interfaces, there are several changes of Dataset in MMDet3D 1.1.

- All the datasets support to serialize the internal data list to reduce the memory when multiple workers are built for data loading.
- The internal data structure in the dataset is changed to be self-contained (without losing information like class names in MMDet3D 1.0.x) while keeping simplicity.
- Common keys across different datasets and data modalities are defined and all the info files are unified into a standard protocol.
- The evaluation functionality of each dataset has been removed from dataset so that some specific evaluation metrics like KITTI AP can be used to evaluate the prediction on other datasets.

31.7.5 Data Transforms

The data transforms in MMDet3D 1.1 all inherits from `BaseTransform` in `MMCV>=2.0.0rc0`, which defines a new convention in OpenMMLab 2.0 projects. Besides the interface changes, there are several changes listed as below:

- The functionality of some data transforms (e.g., `Resize`) are decomposed into several transforms to simplify and clarify the usages.
- The format of data dict processed by each data transform is changed according to the new data structure of dataset.
- Some inefficient data transforms (e.g., normalization and padding) are moved into data preprocessor of model to improve data loading and training speed.
- The same data transforms in different OpenMMLab 2.0 libraries have the same augmentation implementation and the logic given the same arguments, i.e., `Resize` in MMDet 3.x and MMSeg 1.x will resize the image in the exact same manner given the same arguments.

31.7.6 Model

The models in MMDet3D 1.1 all inherits from `BaseModel` in `MMEngine`, which defines a new convention of models in OpenMMLab 2.0 projects. Users can refer to [the tutorial of model in MMEngine](#) for more details. Accordingly, there are several changes as the following:

- The model interfaces, including the input and output formats, are significantly simplified and unified following the new convention in MMDet3D 1.1. Specifically, all the input data in training and testing are packed into `inputs` and `data_samples`, where `inputs` contains model inputs like a dict contain a list of image tensors and the point cloud data, and `data_samples` contains other information of the current data sample such as ground truths, region proposals, and model predictions. In this way, different tasks in MMDet3D 1.1 can share the same input arguments, which makes the models more general and suitable for multi-task learning and some flexible training paradigms like semi-supervised learning.
- The model has a data preprocessor module, which are used to pre-process the input data of model. In MMDet3D 1.1, the data preprocessor usually does necessary steps to form the input images into a batch, such as padding. It can also serve as a place for some special data augmentations or more efficient data transformations like normalization.
- The internal logic of model have been changed. In MMDet3D 1.1, model uses `forward_train`, `forward_test`, `simple_test`, and `aug_test` to deal with different model forward logics. In MMDet3D 1.1 and OpenMMLab 2.0, the forward function has three modes: 'loss', 'predict', and 'tensor' for training, inference, and tracing or other purposes, respectively. The forward function calls `self.loss`, `self.predict`, and `self._forward` given the modes 'loss', 'predict', and 'tensor', respectively.

31.7.7 Evaluation

The evaluation in MMDet3D 1.0.x strictly binds with the dataset. In contrast, MMDet3D 1.1 decomposes the evaluation from dataset, so that all the detection dataset can evaluate with KITTI AP and other metrics implemented in MMDet3D 1.1. MMDet3D 1.1 mainly implements corresponding metrics for each dataset, which are manipulated by [Evaluator](#) to complete the evaluation. Users can build evaluator in MMDet3D 1.1 to conduct offline evaluation, i.e., evaluate predictions that may not produced in MMDet3D 1.1 with the dataset as long as the dataset and the prediction follows the dataset conventions. More details can be find in the [tutorial in mmengine](#).

31.7.8 Visualization

The functions of visualization in MMDet3D 1.1 are removed. Instead, in OpenMMLab 2.0 projects, we use [Visualizer](#) to visualize data. MMDet3D 1.1 implements `Det3DLocalVisualizer` to allow visualization of 2D and 3D data, ground truths, model predictions, and feature maps, etc., at any place. It also supports to send the visualization data to any external visualization backends such as Tensorboard.

31.8 Planned changes

We list several planned changes of MMDet3D 1.1.0rc0 so that the community could more comprehensively know the progress of MMDet3D 1.1. Feel free to create a PR, issue, or discussion if you are interested, have any suggestions and feedbacks, or want to participate.

1. Test-time augmentation: which is supported in MMDet3D 1.0.x, is not implemented in this version due to limited time slot. We will support it in the following releases with a new and simplified design.
2. Inference interfaces: a unified inference interfaces will be supported in the future to ease the use of released models.
3. Interfaces of useful tools that can be used in notebook: more useful tools that implemented in the `tools` directory will have their python interfaces so that they can be used through notebook and in downstream libraries.
4. Documentation: we will add more design docs, tutorials, and migration guidance so that the community can deep dive into our new design, participate the future development, and smoothly migrate downstream libraries to MMDet3D 1.1.
5. Wandb visualization: MMDet 2.x supports data visualization since v2.25.0, which has not been migrated to MMDet 3.x for now. Since Wandb provides strong visualization and experiment management capabilities, a `DetWandbVisualizer` and maybe a hook are planned to fully migrated those functionalities in MMDet 2.x and a `Det3DWandbVisualizer` will be supported in MMDet3D 1.1 accordingly.
6. Will support recent new features added in MMDet3D 1.0.x and our recent exploration on camera-only 3D detection from videos: we will refactor these models and support them with benchmarks and models soon.

COMPATIBILITY

32.1 v1.1.0rc0

32.1.1 OpenMMLab v2.0 Refactoring

In this version, we make large refactoring based on MMEngine to achieve unified data elements, model interfaces, visualizers, evaluators and other runtime modules across different datasets, tasks and even codebases. A brief summary for this refactoring is as follows:

- Data Element:
 - We add `Det3DDataSample` as the common data element passing through datasets and models. It inherits from `DetDataSample` in `mmdetection` and implement `InstanceData`, `PixelData`, and `LabelData` inheriting from `BaseDataElement` in `MMEngine` to represent different types of ground truth labels or predictions.
- Datasets:
 - We add `Det3DDataset` and `Seg3DDataset` as the base datasets to inherit from the unified `BaseDataset` in `MMEngine`. They implement most functions that are commonly used across different datasets and simplify the info loading/processing in the current datasets. Re-defined input arguments and functions can be most re-used in different datasets, which are important for the implementation of customized datasets.
 - We define the common keys across different datasets and unify all the info files with a standard protocol. The same info is more clear for users because they share the same key across different dataset infos. Besides, for different settings, such as camera-only and LiDAR-only methods, we no longer need different info formats (like the previous `pkl` and `json` files). We can just revise the `parse_data_info` to read the necessary information from a complete info file.
 - We add `train_dataloader`, `val_dataloader` and `test_dataloader` to replace the original data in the config. It simplify the levels of data-related fields.
- Data Transforms
 - Based on the basic transforms and wrappers re-implemented and simplified in the latest `MMCV`, we refactor data transforms to inherit from them.
 - We also adjust the implementation of current data pipelines to make them compatible with our latest data protocol.
 - Normalization, padding of images and voxelization operations are moved to the data-preprocessing.
 - `DefaultFormatBundle3D` and `Collect3D` are replaced with `PackDet3DInputs` to pack the data into the element format as model input.
- Models

- Unify the model interface as `inputs, data_samples, return_loss=False`
- The basic pre-processing before model forward includes: 1) convert input from CPU to GPU tensors; 2) padding images; 3) normalize images; 4) voxelization.
- Return `loss_dict` during training while `list[data_sample]` during inference
- Simply function interfaces in the models
- Add `preprocess_cfg` in the model configs for pre-processing
- Visualizer
 - Design a unified visualizer, `Det3DLocalVisualizer`, based on MMEEngine for different 3D tasks and settings
 - Support browsing dataset and visualization hooks based on the `Det3DLocalVisualizer`
- Evaluator
 - Decouple evaluators from datasets to make them more flexible: the evaluation codes of each dataset are implemented as a metric class exclusively.
 - Add evaluator information to the current dataset configs
- Registry
 - Refactor all the registries to inherit from root registries in MMEEngine
 - When using modules from other codebases, it is necessary to specify the registry scope, such as `mmdet.ResNet`
- Others: Refactor logging, hooks, scheduler, runner and other runtime configs based on MMEEngine

32.2 v1.0.0rc1

32.2.1 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.

32.2.2 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.

32.3 v1.0.0rc0

32.3.1 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 (1, w, h) instead of (w, 1, h). This is more natural since 1 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.

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 (1, w, h) instead of (w, 1, 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`.

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 .

32.4 0.16.0

32.4.1 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.

32.4.2 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.

32.4.3 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.

32.5 0.15.0

32.5.1 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](#).

32.5.2 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.

32.5.3 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.

32.5.4 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.

32.6 0.14.0

32.6.1 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.

32.6.2 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](#).

32.7 0.12.0

32.7.1 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.

32.8 0.6.0

32.8.1 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](#).

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.

33.1 MMEEngine/MMCV/MMDet/MMDet3D Installation

- Compatibility issue between MMEEngine, MMCV, MMDetection and MMDetection3D; “ConvWS is already registered in conv layer”; “AssertionError: MMCV==xxx is used but incompatible. Please install mmcv>=xxx, <=xxx.”
- The required versions of MMEEngine, MMCV and MMDetection for different versions of MMDetection3D are as below. Please install the correct version of MMEEngine, MMCV and MMDetection to avoid installation issues.

Note: If you want to install mmdet3d-v1.0.0rcx, the compatible MMDetection, MMSegmentation and MMCV versions table can be found at [here](#). Please choose 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"
```

- If you face some errors about numba in cuda-9.0 environment, you should check the version of numba. In cuda-9.0 environment, the high version of numba is not supported and we suggest you could install numba==0.53.0.

33.2 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
THIRTYFOUR

ENGLISH

CHAPTER
THIRTYFIVE

INDICES AND TABLES

- `genindex`
- `search`

PYTHON MODULE INDEX

m

`mmdet3d.structures.bbox_3d`, [163](#)
`mmdet3d.structures.points`, [180](#)
`mmdet3d.utils`, [191](#)

INDEX

A

`array_converter()` (in module `mmdet3d.utils`), 192

`ArrayConverter` (class in `mmdet3d.utils`), 191

`attribute_dims` (`mmdet3d.structures.points.BasePoints` attribute), 180

`attribute_dims` (`mmdet3d.structures.points.CameraPoints` attribute), 183

`attribute_dims` (`mmdet3d.structures.points.DepthPoints` attribute), 184

`attribute_dims` (`mmdet3d.structures.points.LiDARPoints` attribute), 184

B

`BaseInstance3DBBoxes` (class in `mmdet3d.structures.bbox_3d`), 163

`BasePoints` (class in `mmdet3d.structures.points`), 180

`bev` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` property), 163

`bev` (`mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes` property), 169

`bev` (`mmdet3d.structures.points.BasePoints` property), 180

`bev` (`mmdet3d.structures.points.CameraPoints` property), 183

`bottom_center` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` property), 163

`bottom_height` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` property), 164

`bottom_height` (`mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes` property), 170

`Box3DMode` (class in `mmdet3d.structures.bbox_3d`), 168

`box_dim` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` attribute), 163

`box_dim` (`mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes` attribute), 169

`box_dim` (`mmdet3d.structures.bbox_3d.DepthInstance3DBBoxes` attribute), 174

`box_dim` (`mmdet3d.structures.bbox_3d.LiDARInstance3DBBoxes` attribute), 176

`mmdet3d.structures.bbox_3d`), 169

`CameraPoints` (class in `mmdet3d.structures.points`), 182

`cat()` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` class method), 164

`cat()` (`mmdet3d.structures.points.BasePoints` class method), 180

`center` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` property), 164

`clone()` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` method), 164

`clone()` (`mmdet3d.structures.points.BasePoints` method), 180

`collect_env()` (in module `mmdet3d.utils`), 193

`color` (`mmdet3d.structures.points.BasePoints` property), 180

`compat_cfg()` (in module `mmdet3d.utils`), 193

`convert()` (`mmdet3d.structures.bbox_3d.Box3DMode` static method), 169

`convert()` (`mmdet3d.structures.bbox_3d.Coord3DMode` static method), 173

`convert()` (`mmdet3d.utils.ArrayConverter` method), 191

`convert_box()` (`mmdet3d.structures.bbox_3d.Coord3DMode` static method), 173

`convert_point()` (`mmdet3d.structures.bbox_3d.Coord3DMode` static method), 173

`convert_to()` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` method), 164

`convert_to()` (`mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes` method), 170

`convert_to()` (`mmdet3d.structures.bbox_3d.DepthInstance3DBBoxes` method), 174

`convert_to()` (`mmdet3d.structures.bbox_3d.LiDARInstance3DBBoxes` method), 176

`convert_to()` (`mmdet3d.structures.points.BasePoints` method), 180

`convert_to()` (`mmdet3d.structures.points.CameraPoints` method), 183

`convert_to()` (`mmdet3d.structures.points.DepthPoints` method), 184

`convert_to()` (`mmdet3d.structures.points.LiDARPoints` method), 185

`coord` (`mmdet3d.structures.points.BasePoints` property),

C

`CameraInstance3DBBoxes` (class in

181
 Coord3DMode (class in mmdet3d.structures.bbox_3d),
 172
 corners (mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes
 property), 164
 corners (mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes
 property), 170
 corners (mmdet3d.structures.bbox_3d.DepthInstance3DBBoxes
 property), 175
 corners (mmdet3d.structures.bbox_3d.LiDARInstance3DBBoxes
 property), 177
 get_surface_line_center()
 (mmdet3d.structures.bbox_3d.DepthInstance3DBBoxes
 method), 175
 gravity_center (mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes
 property), 165
 gravity_center (mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes
 property), 171
 gravity_center (mmdet3d.structures.bbox_3d.DepthInstance3DBBoxes
 property), 175
 gravity_center (mmdet3d.structures.bbox_3d.LiDARInstance3DBBoxes
 property), 177

D

DepthInstance3DBBoxes (class in mmdet3d.structures.bbox_3d), 174
 DepthPoints (class in mmdet3d.structures.points), 183
 device (mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes
 property), 164
 device (mmdet3d.structures.points.BasePoints prop-
 erty), 181
 dims (mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes
 property), 165
 height (mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes
 property), 165
 height (mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes
 property), 171
 height (mmdet3d.structures.points.BasePoints prop-
 erty), 181
 height_overlaps() (mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes
 class method), 165
 height_overlaps() (mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes
 class method), 171

E

enlarged_box() (mmdet3d.structures.bbox_3d.DepthInstance3DBBoxes
 method), 175
 enlarged_box() (mmdet3d.structures.bbox_3d.LiDARInstance3DBBoxes
 method), 177
 in_range_3d() (mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes
 method), 165
 in_range_3d() (mmdet3d.structures.points.BasePoints
 method), 181

F

flip() (mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes
 method), 165
 flip() (mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes
 method), 170
 flip() (mmdet3d.structures.bbox_3d.DepthInstance3DBBoxes
 method), 175
 flip() (mmdet3d.structures.bbox_3d.LiDARInstance3DBBoxes
 method), 177
 flip() (mmdet3d.structures.points.BasePoints method),
 181
 flip() (mmdet3d.structures.points.CameraPoints
 method), 183
 flip() (mmdet3d.structures.points.DepthPoints
 method), 184
 flip() (mmdet3d.structures.points.LiDARPoints
 method), 185
 in_range_bev() (mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes
 method), 165
 in_range_bev() (mmdet3d.structures.points.BasePoints
 method), 181

L

LiDARInstance3DBBoxes (class in mmdet3d.structures.bbox_3d), 176
 LiDARPoints (class in mmdet3d.structures.points), 184
 limit_period() (in module
 mmdet3d.structures.bbox_3d), 178
 limit_yaw() (mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes
 method), 166
 local_yaw (mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes
 property), 171

G

get_box_type() (in module
 mmdet3d.structures.bbox_3d), 178
 get_proj_mat_by_coord_type() (in module
 mmdet3d.structures.bbox_3d), 178
 mmdet3d.structures.bbox_3d
 module, 163
 mmdet3d.structures.points
 module, 180
 mmdet3d.utils
 module, 191
 module

H

height (mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes
 property), 165
 height (mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes
 property), 171
 height (mmdet3d.structures.points.BasePoints prop-
 erty), 181
 height_overlaps() (mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes
 class method), 165
 height_overlaps() (mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes
 class method), 171

I

in_range_3d() (mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes
 method), 165
 in_range_3d() (mmdet3d.structures.points.BasePoints
 method), 181
 in_range_bev() (mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes
 method), 165
 in_range_bev() (mmdet3d.structures.points.BasePoints
 method), 181

L

LiDARInstance3DBBoxes (class in mmdet3d.structures.bbox_3d), 176
 LiDARPoints (class in mmdet3d.structures.points), 184
 limit_period() (in module
 mmdet3d.structures.bbox_3d), 178
 limit_yaw() (mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes
 method), 166
 local_yaw (mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes
 property), 171

M

mmdet3d.structures.bbox_3d
 module, 163
 mmdet3d.structures.points
 module, 180
 mmdet3d.utils
 module, 191
 module

`mmdet3d.structures.bbox_3d`, 163
`mmdet3d.structures.points`, 180
`mmdet3d.utils`, 191
`mono_cam_box2vis()` (in module `mmdet3d.structures.bbox_3d`), 178

N

`nearest_bev` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` property), 166
`new_box()` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` method), 166
`new_point()` (`mmdet3d.structures.points.BasePoints` method), 181
`nonempty()` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` attribute), 184
`nonempty()` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` method), 166

O

`overlaps()` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` class method), 166

P

`points_cam2img()` (in module `mmdet3d.structures.bbox_3d`), 179
`points_dim` (`mmdet3d.structures.points.BasePoints` attribute), 180
`points_dim` (`mmdet3d.structures.points.CameraPoints` attribute), 183
`points_dim` (`mmdet3d.structures.points.DepthPoints` attribute), 184
`points_dim` (`mmdet3d.structures.points.LiDARPoints` attribute), 184
`points_img2cam()` (in module `mmdet3d.structures.bbox_3d`), 179
`points_in_boxes_all()` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` method), 167
`points_in_boxes_all()` (`mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes` method), 171
`points_in_boxes_part()` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` method), 167
`points_in_boxes_part()` (`mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes` method), 171

R

`recover()` (`mmdet3d.utils.ArrayConverter` method), 191
`register_all_modules()` (in module `mmdet3d.utils`), 193
`rotate()` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` method), 167
`rotate()` (`mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes` method), 172

`rotate()` (`mmdet3d.structures.bbox_3d.DepthInstance3DBBoxes` method), 175
`rotate()` (`mmdet3d.structures.bbox_3d.LiDARInstance3DBBoxes` method), 177
`rotate()` (`mmdet3d.structures.points.BasePoints` method), 182
`rotation_3d_in_axis()` (in module `mmdet3d.structures.bbox_3d`), 179
`rotation_axis` (`mmdet3d.structures.points.BasePoints` attribute), 180
`rotation_axis` (`mmdet3d.structures.points.CameraPoints` attribute), 183
`rotation_axis` (`mmdet3d.structures.points.DepthPoints` attribute), 184
`rotation_axis` (`mmdet3d.structures.points.LiDARPoints` attribute), 184

S

`scale()` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` method), 167
`scale()` (`mmdet3d.structures.points.BasePoints` method), 182
`set_template()` (`mmdet3d.utils.ArrayConverter` method), 191
`setup_multi_processes()` (in module `mmdet3d.utils`), 193
`shape` (`mmdet3d.structures.points.BasePoints` property), 182
`shuffle()` (`mmdet3d.structures.points.BasePoints` method), 182

T

`tensor` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` attribute), 163
`tensor` (`mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes` attribute), 169
`tensor` (`mmdet3d.structures.bbox_3d.DepthInstance3DBBoxes` attribute), 174
`tensor` (`mmdet3d.structures.bbox_3d.LiDARInstance3DBBoxes` attribute), 176
`tensor` (`mmdet3d.structures.points.BasePoints` attribute), 180
`tensor` (`mmdet3d.structures.points.CameraPoints` attribute), 183
`tensor` (`mmdet3d.structures.points.DepthPoints` attribute), 183
`tensor` (`mmdet3d.structures.points.LiDARPoints` attribute), 184
`to()` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` method), 167
`to()` (`mmdet3d.structures.points.BasePoints` method), 182
`top_down_height` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes` property), 168

`top_height` (`mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes`
property), [172](#)
`translate()` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes`
method), [168](#)
`translate()` (`mmdet3d.structures.points.BasePoints`
method), [182](#)

V

`volume` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes`
property), [168](#)

W

`with_yaw` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes`
attribute), [163](#)
`with_yaw` (`mmdet3d.structures.bbox_3d.CameraInstance3DBBoxes`
attribute), [169](#)
`with_yaw` (`mmdet3d.structures.bbox_3d.DepthInstance3DBBoxes`
attribute), [174](#)
`with_yaw` (`mmdet3d.structures.bbox_3d.LiDARInstance3DBBoxes`
attribute), [176](#)

X

`xywhr2xyxyr()` (in `mmdet3d.structures.bbox_3d` module), [179](#)

Y

`yaw` (`mmdet3d.structures.bbox_3d.BaseInstance3DBBoxes`
property), [168](#)