d3d

Release 0.5.0b1.dev38

Jacob Zhong

Feb 17, 2022

GET STARTED

D3D is a collections of tools built for 3D Machine Learning, currently it's mainly designed for 3D object detection and tracking tasks.

Please consider siting my work if you find this library useful in your research :)

```
@article{zhong2020uncertainty,
  title={Uncertainty-Aware Voxel based 3D Object Detection and Tracking with von-Mises␣
↪Loss},
  author={Zhong, Yuanxin and Zhu, Minghan and Peng, Huei},
  journal={arXiv preprint arXiv:2011.02553},
  year={2020}
}
```

CHAPTER

ONE

REQUIREMENTS

Installation requirements:

- `python >= 3.6`
- `numpy >= 1.17.0`
- `scipy >= 1.4`
- `addict`
- `pillow`
- `tqdm`
- `msgpack`
- `filterpy`
- `PyYAML`
- `sortedcontainers`

Build requirements:

- `cython >= 0.29.16`
- `scikit-build`
- `setuptools-scm`

Optional requirements:

- `utm`: support converting GPS coordinate to local frame
- `pytorch >= 1.4`: support custom pytorch operators
- `pcl.py`: support visualization in PCL
- `matplotlib`: support visualization in 2D figures
- `waymo_open_dataset`: support converting Waymo Dataset
- `intervaltree`: support indexing in some datasets
- `pyyaml`: support calibration loading in some datasets
- `filelock`: support indexing in some datasets
- `scikit-learn`: support indexing in some datasets

CHAPTER

TWO

BUILD

- create build environment in conda: `conda create -f conda/env-dev.yaml`
- build and install: `python setup.py install`
- build wheel: `python setup.py bdist_wheel`
- build in-place: `python setup.py develop`
- build debug: `python setup.py develop --build-type Debug`

2.1 Build on cluster

Some tips about building the library in a cluster: The default behavior of building is using all the CPU cores, so if you find the compiler crashed during compilation, that's usually due to insufficient memory. You can choose the number of parallel building by using `-jN` switch along with those building commands

2.2 Wheels

Prebuilt wheels will be distributed in the future, through either release page or conda channel. Only source distribution will be uploaded to PyPI.

2.3 Versioning

- Major version will be increased when big feature is added
- Minor version will be increased when API compatibility is broken
- Patch version will be increased when new feature is completed.

CHAPTER

THREE

FEATURES

- Unified data representation
- Support loading KITTI, Waymo, Nuscenes dataset
- Rotated 2D IoU, NMS with clear CUDA implementations
- Point Cloud Voxelization
- Visualization
- Benchmarking

3.1 Package structure

- `d3d.abstraction`: Common interface definitions
- `d3d.benchmark`: Implementation of benchmarks
- `d3d.box`: Modules for bounding box related calculations
- `d3d.dataset`: Modules for dataset loading
- `d3d.math`: Implementation of some special math functions
- `d3d.point`: Modules for point array related components
- `d3d.vis`: Modules for visualizations
- `d3d.voxel`: Moduels for voxel related components

CHAPTER

FOUR

DATA ABSTRACTION

CHAPTER

FIVE

DATASETS

The support status of various 3D datasets are listed below.

CHAPTER

SIX

OPERATORS

CHAPTER

SEVEN

TARGET TRACKING

CHAPTER

EIGHT

UTILITIES

CHAPTER

NINE

D3D.ABSTRACTION

This module contains definition of basic data structures.

class d3d.abstraction.**CameraMetadata**(*int width*, *int height*, *ndarray distort_coeffs*, *ndarray intri_matrix*, *float mirror_coeff*)

Bases: object

This class represents intrinsic parameters of a camera

Parameters

- **width** – Width of the image
- **height** – Height of the image
- **distort_coeffs** – Distortion coefficients
- **intri_matrix** – Intrisic matrix of the camera
- **mirror_coeff** – Mirror coefficient for stereo setup

distort_coeffs
numpy.ndarray

Coefficients of camera distortion model, follow OpenCV format

Type distort_coeffs

height
'int'

Height of the camera image

Type height

intri_matrix
numpy.ndarray

Original intrinsic matrix used for cv2.undistortPoints

Type intri_matrix

mirror_coeff
'float'

Coefficient of mirror equation (as used in MEI camera model)

Type mirror_coeff

width
'int'

Width of the camera image

Type width

class d3d.abstraction.**EgoPose**(*position*, *orientation*, *position_var=None*, *orientation_var=None*)
Bases: object

This object is used to store dynamic state of ego vehicle. All value is represented in earth-fixed coordinate (absolute coordinate).

Parameters

- **position** – position of ego sensor, [x, y, z]
- **orientation** – orientation of ego sensor, in format of [x, y, z, w] quaternion of scipy Rotation object
- **position_var** – positional variance of ego sensor, [var_x, var_y, var_z]
- **orientation_var** – orientation variance of ego sensor

homo(*self*)
Convert the pose to a homogeneous matrix representation

orientation
The orientation of the ego sensor

orientation_var
numpy.ndarray

Variance of the estimation of the sensor orientation

Type orientation_var

position
numpy.ndarray

The position of the ego sensor

Type position

position_var
numpy.ndarray

Variance of the estimation of the sensor position

Type position_var

class d3d.abstraction.**LidarMetadata**
Bases: object

This class represents intrinsic parameters of a lidar

class d3d.abstraction.**ObjectTag**(*labels*, *mapping=None*, *scores=None*)
Bases: object

This class stands for label tags associate with object target. This class can contains multiple estimated classes with separate confidence scores.

Parameters

- **labels** – A label or list of labels as enum object, enum name or enum id.
- **mapping** – The enum object defining the label classes.
- **scores** – Scores corresponding to the input labels.

deserialize(*type cls*, *data*)
Deserialize data from python primitives

labels
'vector[int]'

The ids of the labels, sorted by score in descending order

Type labels

mapping
object

The Enum class defining the label classes

Type mapping

scores
'vector[float]'

The ids of the labels, sorted by score in descending order

Type scores

serialize(*self*)
Serialize this object to primitives

class d3d.abstraction.**ObjectTarget3D**(*position*, *orientation*, *dimension*, *tag*, *tid=0*, *position_var=None*, *orientation_var=None*, *dimension_var=None*, *aux=None*)

Bases: object

This class stands for a target in cartesian coordinate. The body coordinate is FLU (front-left-up).

Parameters

- **position** – Position of object center (x,y,z)
- **orientation** – Object heading (direction of x-axis attached on body) with regard to x-axis of the world at the object center.
- **dimension** – Length of the object in 3 dimensions (lx,ly,lz)
- **tag** – Classification information of the object
- **tid** – ID of the object used for tracking (optional), 0 means no tracking id assigned
- **position_var** – The uncertainty of target position
- **orientation_var** – The uncertainty of target orientation
- **dimension_var** – The uncertainty of target dimension

aux
dict

Auxiliary data attached to the object

Type aux

box_iou(*self*, *ObjectTarget3D other*)

corners
Convert the bounding box representation to coorindate of 8 corner points

crop_points(*self*, *ndarray cloud*)

deserialize(*type cls*, *data*)
Deserialize data from python primitives

dimension
Dimension of the target

dimension_var
Variance of dimension estimation of the target

orientation
Orientation of the target

orientation_var
'float'

Variance of orientation of the target. This API may be changed in future

Type orientation_var

points_distance(*self*, *ndarray cloud*)

position
Position of the (center of) target

position_var
Positional variance of the (center of) target

serialize(*self*)
Serialize this object to python primitives

tag
d3d.abstraction.ObjectTag

The tag attached to the target

Type tag

tag_top
Return the object of the target's top tag

tag_top_score
Return the score of the target's top tag

tid
'unsigned long long'

The unique id of the target (across frames). tid = 0 means no id assigned, so valid tid should be greater than 1.

Type tid

tid64
Return base64 represented tracking id

to_numpy(*self*, *unicode box_type=u'ground'*) → ndarray
Convert the object to numpy array representation

Parameters **box_type** – The type of box representation * ground: use the representation of bird's eye view 2D projection

yaw
Return the rotation angle around z-axis (ignoring rotations in other two directions)

class d3d.abstraction.**PinMetadata**(*float lon*, *float lat*)
Bases: object

This class represents a ground-fixed coordinate. The coordinate can be in WGS-84 or local UTM coordinate system.

Parameters

- **lon** – Longitude coordinate value

- **lat** – Latitude coordinate value

lat
'float'

Latitude coordinate of the pin

Type lat

lon
'float'

Longitude coordinate of the pin

Type lon

class d3d.abstraction.**RadarMetadata**
Bases: object

This class represents intrinsic parameters of a radar

class d3d.abstraction.**Target3DArray**(*iterable=[]*, *frame=None*, *timestamp=0*)
Bases: list

Target3DArray stores an array of ObjectTarget3D or TrackingTarget3D represented in the frame of certain senser at certain time.

Parameters

- **iterable** – List of targets
- **frame** – Sensor frame that the box parameters used. None means base frame (consistent with TransformSet)
- **timestamp** – The timestamp of the target properties

crop_points(*self*, *ndarray cloud*)

deserialize(*type cls*, *data*)
Deserialize data from python primitives

dump(*self*, *output*)
Serialize the array and dump it into file

Parameters **output** – output file-like object or file path

filter(*self*, *predicate*)
Filter the list of objects by predicate

filter_position(*self*, *float x_min=float(u'nan')*, *float x_max=float(u'nan')*, *float y_min=float(u'nan')*, *float y_max=float(u'nan')*, *float z_min=float(u'nan')*, *float z_max=float(u'nan')*)
Filter the list of objects by the center position

Parameters

- **x_min** – Minimum x coordinate
- **x_max** – Maximum x coordinate
- **y_min** – Minimum y coordinate
- **y_max** – Maximum y coordinate
- **z_min** – Minimum z coordinate
- **z_max** – Maximum z coordinate

filter_score(*self*, *score*)
Filter the list by select only objects higher than certain score

Parameters **score** – The minimun score for tag_top_score field

filter_tag(*self*, *tags*)
Filter the list by select only objects with given tags

Parameters **tags** – None means no filter, otherwise str/enum or list of str/enum

frame
unicode

The transform frame which the targets lie in

Type frame

load(*type cls*, *file*)
Load the array form a binary file created by dump()

Parameters **file** – path of input file or file-like object to be loaded

paint_label(*self*, *ndarray cloud*, *ndarray semantics*)

serialize(*self*)
Serialize this object to python primitives

sort_by_score(*self*, *reverse=False*)
Sort the box list (in place) by the score

Parameters **reverse** – sorting is done ascendingly by default, reverse means descending

timestamp
'unsigned long long'

The timestamp of when the targets are annotated or reported. It's represented by unix timestamp in milliseconds

Type timestamp

to_numpy(*self*, *unicode box_type=u'ground'*) → ndarray
Convert the object array to numpy array representation

- ground: use the representation of bird's eye view 2D projection

to_torch(*self*, *box_type='ground'*)
Convert the object array to PyTorch Tensor representation

class d3d.abstraction.**TrackingTarget3D**(*position*, *orientation*, *dimension*, *velocity*, *angular_velocity*, *tag*, *tid=0*, *position_var=None*, *orientation_var=None*, *dimension_var=None*, *velocity_var=None*, *angular_velocity_var=None*, *history=None*, *aux=None*)
Bases: *d3d.abstraction.ObjectTarget3D*

This class stands for a tracked target in cartesian coordinate. The body coordinate is FLU (front-left-up).

Parameters

- **velocity** – Velocity of the object (vx,vy,vz)
- **angular_velocity** – Angular velocity of the object (wx,wy,wz)
- **velocity_var** – The uncertainty of the target velocity
- **angular_velocity** – The uncertainty of the target angular velocity
- **history** – The time of the object being tracked

angular_velocity
Angular velocity of the target

angular_velocity_var
Variance of angular velocity estimation of the target

deserialize(*type cls*, *data*)

history
'float'

Tracked time of this target in seconds

Type history

serialize(*self*)

to_numpy(*self*, *unicode box_type=u'ground'*) → ndarray

velocity
Velocity of the (center of) target

velocity_var
Variance of velocity estimation of the target

class d3d.abstraction.**TransformSet**(*unicode base_frame*)
Bases: object

This object load a collection of intrinsic and extrinsic parameters All extrinsic parameters are stored as transform from base frame to its frame In this class, we require all frames to use FLU coordinate including camera frame

Parameters **base_frame** – name of base frame used by extrinsics

base_frame
unicode

The default frame for the transform set

Type base_frame

dump(*self*, *output*)
Serialize the transform collection and dump it into file

Parameters **output** – output file-like object or file path

extrinsics
dict

Type extrinsics

frames
Report registered frame names (excluding base_frame)

get_extrinsic(*self*, *unicode frame_to=None*, *unicode frame_from=None*) → ndarray

Parameters **frame_from** – If set to None, then the source frame is base frame

intrinsics
dict

This dictionary defines all the valid sensor frames and store their projection matrix (for camera)

Type intrinsics

intrinsics_meta
dict

This dictionary defines the related metadata for sensors

Type intrinsics_meta

load(*type cls*, *file*)
Load the transform collection form a binary file created by dump()

Parameters **file** – path of input file or file-like object to be loaded

project_points_to_camera(*self*, *ndarray points*, *unicode frame_to*, *unicode frame_from=None*, *bool remove_outlier=True*, *bool return_dmask=False*) → tuple

Parameters

- **remove_outlier** – If set to True, the mask will be applied, i.e. only points that fall into image view will be returned
- **return_dmask** – also return the mask for z > 0 only

Returns return points, mask and dmask if required. The masks are array of indices

set_extrinsic(*self*, *transform*, *unicode frame_to=None*, *unicode frame_from=None*) → void
All extrinsics are stored as transform convert point from *frame_from* to *frame_to* :param frame_from: If set to None, then the source frame is base frame :param frame_to: If set to None, then the target frame is base frame

set_intrinsic_camera(*self*, *unicode frame_id*, *ndarray transform*, *size*, *bool rotate=True*, *distort_coeffs=[]*, *ndarray intri_matrix=None*, *float mirror_coeff=float(u'nan')*) → void
Set camera intrinsics :param size: (width, height) :param rotate: if True, then transform will append an axis rotation (Front-Left-Up to Right-Down-Front) :param distort_coeffs: distortion coefficients, see [OpenCV](https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html) for details :param intri_matrix: intrinsic matrix (in general camera model) :param mirror_coeff: the xi coefficient in MEI camera model. Reference: Single View Point OmnidirectionalCamera Calibration from Planar Grids

set_intrinsic_general(*self*, *unicode frame_id*, *metadata=None*) → void
Set intrinsic for a general sensor. This is used for marking existence of a frame

set_intrinsic_lidar(*self*, *unicode frame_id*) → void

set_intrinsic_map_pin(*self*, *unicode frame_id*, *lon=float(u'nan')*, *lat=float(u'nan')*) → void

set_intrinsic_pinhole(*self*, *unicode frame_id*, *size*, *float cx*, *float cy*, *float fx*, *float fy*, *float s=0*, *distort_coeffs=[]*) → void
Set camera intrinsics with pinhole model parameters :param s: skew coefficient

set_intrinsic_radar(*self*, *unicode frame_id*) → void

transform_objects(*self*, *Target3DArray objects*, *unicode frame_to=None*) → *Target3DArray*
Change the representing frame of a object array

transform_points(*self*, *ndarray points*, *unicode frame_to*, *unicode frame_from=None*) → ndarray
Convert point cloud from *frame_from* to *frame_to*

CHAPTER

TEN

D3D.BENCHMARKS

This module contains benchmark utilities

class d3d.benchmarks.**DetectionEvalStats**

Bases: object

Detection stats summary of a evaluation step

acc_angular

'unordered_map[int,vector[float]]'

Type acc_angular

acc_box

'unordered_map[int,vector[float]]'

Type acc_box

acc_dist

'unordered_map[int,vector[float]]'

Type acc_dist

acc_iou

'unordered_map[int,vector[float]]'

Type acc_iou

acc_var

'unordered_map[int,vector[float]]'

Type acc_var

as_object(*self*)

fn

'unordered_map[int,vector[int]]'

Type fn

fp

'unordered_map[int,vector[int]]'

Type fp

ndt

'unordered_map[int,vector[int]]'

Type ndt

ngt

'unordered_map[int,int]'

Type ngt

tp

'unordered_map[int,vector[int]]'

Type tp

class d3d.benchmarks.**DetectionEvaluator**(*classes*, *min_overlaps*, *int pr_sample_count=40*, *float min_score=0*, *unicode pr_sample_scale=u'log10'*)

Bases: object

Benchmark for object detection

Object detection benchmark. Targets association is done by score sorting.

Parameters

- **classes** – Object classes to consider
- **min_overlaps** – Min overlaps per class for two boxes being considered as overlap. If single value is provided, all class will use the same overlap threshold
- **min_score** – Min score for precision-recall samples
- **pr_sample_count** – Number of precision-recall sample points (expect for p=1,r=0 and p=0,r=1)
- **pr_sample_scale** – PR sample type, {lin: linspace, log: logspace 1~10, logX: logspace 1~X}

acc_angular(*self*, *float score=NAN*)

acc_box(*self*, *float score=NAN*)

acc_dist(*self*, *float score=NAN*)

acc_iou(*self*, *float score=NAN*)

add_stats(*self*, *DetectionEvalStats stats*) → void
Add statistics from calc_stats into database

ap(*self*)
Calculate (mean) average precision

calc_stats(*self*, *Target3DArray gt_boxes*, *Target3DArray dt_boxes*, *TransformSet calib=None*) → *DetectionEvalStats*

dt_count(*self*, *float score=NAN*)

fn(*self*, *float score=NAN*)
Return false negative count. If score is not specified, return the median value

fp(*self*, *float score=NAN*)
Return false positive count. If score is not specified, return the median value

fscore(*self*, *float score=NAN*, *float beta=1*, *bool return_all=False*)

get_stats(*self*) → *DetectionEvalStats*
Summarize current state of the benchmark counters

gt_count(*self*)

precision(*self*, *float score=NAN*, *bool return_all=False*)

recall(*self*, *float score=NAN*, *bool return_all=False*)

reset(*self*) → void

summary(*self*, *float score_thres=0.8*, *bool verbose=False*)
Print default summary (into returned string)

tp(*self*, *float score=NAN*)
Return true positive count. If score is not specified, return the median value

class d3d.benchmarks.**SegmentationEvaluator**(*classes*, *background=0*, *min_points=0*)
Bases: object

Benchmark for semgentation

Parameters

- **classes** – classes to be considered during evaluation, other classes are all considered as background
- **background** – class to be considered as background class
- **min_points** – minimum number of points when calculating segments in panoptic evaluation

add_stats(*self*, *SegmentationStats stats*) → void

calc_stats(*self*, *ndarray gt_labels*, *ndarray pred_labels*, *ndarray gt_ids=None*, *ndarray pred_ids=None*) → *SegmentationStats*
Please make sure the id are 0 if the label is in stuff category

fn(*self*, *bool instance=False*)

fp(*self*, *bool instance=False*)

get_stats(*self*) → *SegmentationStats*
Summarize current state of the benchmark counters

iou(*self*, *bool instance=False*)

pq(*self*)
Panoptic Quality (PQ) in panoptic segmentation

reset(*self*) → void

rq(*self*)
Recognition Quality (RQ) in panoptic segmentation

sq(*self*)
Segmentation Quality (SQ) in panoptic segmentation

summary(*self*)

tp(*self*, *bool instance=False*)

class d3d.benchmarks.**SegmentationStats**
Bases: object

Tracking stats summary of a data frame

as_object(*self*)

cumiou
'unordered_map[uint8_t,float]' Summation of IoU of TP segments in instance segmentation

Type cumiou

fn
'unordered_map[uint8_t,int]' Number of false negative data points in semantic segmentation

Type fn

fp
'unordered_map[uint8_t,int]' Number of false positive data points in semantic segmentation

Type fp

ifn
'unordered_map[uint8_t,int]' Number of false negative data segments in instance segmentation

Type ifn

ifp
'unordered_map[uint8_t,int]' Number of false positives data segments in instance segmentation

Type ifp

itp
'unordered_map[uint8_t,int]' Number of true negative data segments in instance segmentation

Type itp

tp
'unordered_map[uint8_t,int]' Number of true negative data points in semantic segmentation

Type tp

class d3d.benchmarks.**TrackingEvalStats**
Bases: *d3d.benchmarks.DetectionEvalStats*

Tracking stats summary of a evaluation step

as_object(*self*)

fragments
'unordered_map[int,vector[int]]' Number of ground-truth trajectory matched to different tracked tracjetories

Type fragments

id_switches
'unordered_map[int,vector[int]]' Number of tracked trajectory matched to different ground-truth trajectories

Type id_switches

ndt_ids
'unordered_map[int,vector[unordered_map[uint64_t,int]]]' Frame count of all proposal targets (represented by their IDs)

Type ndt_ids

ngt_ids
'unordered_map[int,unordered_map[uint64_t,int]]' Frame count of all ground-truth targets (represented by their IDs)

Type ngt_ids

ngt_tracked
'unordered_map[int,vector[unordered_map[uint64_t,int]]]' Frame count of ground-truth targets being tracked

Type ngt_tracked

class d3d.benchmarks.**TrackingEvaluator**(*classes*, *min_overlaps*, *int pr_sample_count=40*, *float min_score=0*, *unicode pr_sample_scale=u'log10'*)
Bases: *d3d.benchmarks.DetectionEvaluator*

Benchmark for object tracking

Object tracking benchmark. Targets association is done by score sorting.

Parameters

- **classes** – Object classes to consider
- **min_overlaps** – Min overlaps per class for two boxes being considered as overlap. If single value is provided, all class will use the same overlap threshold
- **min_score** – Min score for precision-recall samples
- **pr_sample_count** – Number of precision-recall sample points (expect for p=1,r=0 and p=0,r=1)
- **pr_sample_scale** – PR sample type, {lin: linspace, log: logspace 1~10, logX: logspace 1~X}

add_stats(*self*, *DetectionEvalStats stats*) → void

calc_stats(*self*, *Target3DArray gt_boxes*, *Target3DArray dt_boxes*, *TransformSet calib=None*) → *TrackingEvalStats*

fragments(*self*, *float score=NAN*)
Return fragments count. If score is not specified, return the median value

get_stats(*self*) → *TrackingEvalStats*
Summarize current state of the benchmark counters

gt_traj_count(*self*)
Return total ground-truth trajectory count. gt() will return total bounding box count

id_switches(*self*, *float score=NAN*)
Return ID switch count. If score is not specified, return the median value

lost_ratio(*self*, *float score=NAN*, *float frame_ratio_threshold=0.2*, *bool return_all=False*)
Return the ratio of mostly lost trajectories.

Parameters **frame_ratio_threshold** – The threshold of ratio of tracked frames over total frames. A trajectory with lower tracked frames ratio will be counted as mostly tracked

mota(*self*, *float score=NAN*)
Return the MOTA metric defined by the CLEAR MOT metrics. For MOTP equivalents, see acc_* properties

reset(*self*) → void

summary(*self*, *float score_thres=0.8*, *float tracked_ratio_thres=0.8*, *float lost_ratio_thres=0.2*, *unicode note=None*, *bool verbose=False*)
Print default summary (into returned string)

tracked_ratio(*self*, *float score=NAN*, *float frame_ratio_threshold=0.8*, *bool return_all=False*)
Return the ratio of mostly tracked trajectories.

Parameters **frame_ratio_threshold** – The threshold of ratio of tracked frames over total frames. A trajectory with higher tracked frames ratio will be counted as mostly tracked

CHAPTER

ELEVEN

D3D.BOX

This module contains bounding box related autograd operations implemented with CUDA support.

class d3d.box.**Iou2D**
Bases: torch.autograd.function.Function

Differentiable axis aligned IoU function for 2D boxes

class d3d.box.**Iou2DR**
Bases: torch.autograd.function.Function

Differentiable rotated IoU function for 2D boxes

class d3d.box.**DIou2DR**
Bases: torch.autograd.function.Function

Differentiable rotated DIoU function for 2D boxes

class d3d.box.**GIou2DR**
Bases: torch.autograd.function.Function

Differentiable rotated GIoU function for 2D boxes

d3d.box.**box2d_iou**(*boxes1*, *boxes2*, *method='box'*, *precise=True*)
Differentiable IoU on axis-aligned or rotated 2D boxes

Parameters

- **boxes1** – Input boxes, shape is N x 5 (x,y,w,h,r)
- **boxes2** – Input boxes, shape is N x 5 (x,y,w,h,r)
- **method** – 'box' - axis-aligned box, 'rbox' - rotated box, 'grbox' - giou for rotated box, 'drbox' - diou for rotated box
- **precise** – force using double precision to calculate iou

d3d.box.**box2d_nms**(*boxes*, *scores*, *iou_method='box'*, *supression_method='hard'*, *iou_threshold=0*, *score_threshold=0*, *supression_param=0*, *precise=True*)
NMS on axis-aligned or rotated 2D boxes

Parameters

- **method** – 'box' - axis-aligned box, 'rbox' - rotated box
- **precise** – force using double precision to calculate iou
- **iou_threshold** – IoU threshold for two boxes to be considered as overlapped
- **score_threshold** – Minimum score for a box to be considered as valid

- **suppression_param** – Type of suppression. {0: hard, 1: linear, 2: gaussian}. See reference below for details ..

  Soft-NMS: Bodla, Navaneeth, et al. "Soft-NMS–improving object detection with one line of code." Proceedings of the IEEE international conference on computer vision. 2017.

d3d.box.**seg1d_iou**(*seg1*, *seg2*)

Calculate IoU of 1D segments The input should be n*2, where the last dim is [center, width]

Parameters

- **boxes1** – Input segments, shape is Nx2
- **boxes2** – Input segments, shape is Nx2

CHAPTER

TWELVE

D3D.DATASET

This module contains loaders for various datasets.

class d3d.dataset.base.**NumberPool**(*processes*, *offset=0*, **args*, ***kargs*)

Bases: object

This class is a utility for multiprocessing using tqdm, define the task as

```
def task(ntqdm, ...):
    ...
    for data in tqdm(..., position=ntqdm, leave=False):
        ...
```

Then the parallel progress bars will be displayed in place.

Parameters

- **processes** (*int*) – Number of processes available in the pool. If processes < 1, then functions will be executed in current thread.
- **offset** (*int*) – The offset added to the *ntqdm* value of all process. This is useful when you want to display a progress bar in outer loop.

apply_async(*func*, *args=()*, *callback=None*)

wait_for_once(*margin=0*)

Block current thread and wait for one available process

Parameters **margin** (*int*) – Define when a process is available. The method will return when there is `nprocess + margin` processes in the pool.

class d3d.dataset.base.**DatasetBase**(*base_path*, *inzip=False*, *phase='training'*, *trainval_split=1.0*, *trainval_random=False*)

Bases: object

This class acts as the base for all dataset loaders

Parameters

- **base_path** (*Union[str, pathlib.Path]*) – directory containing the zip files, or the required data
- **inzip** (*bool*) – whether the dataset is store in original zip archives or unzipped
- **phase** (*str*) – training, validation or testing
- **trainval_split** (*Union[float, List[int]]*) – the ratio to split training dataset. See documentation of *split_trainval()* for detail.

- **trainval_random** (*Union[bool, int, str]*) – whether select the train/val split randomly. See documentation of *split_trainval()* for detail.

identity(*idx*)

Return something that can track the data back to original dataset. The result tuple can be passed to any accessors above and directly access given data.

Parameters **idx** (*int*) – index of requested frame to be parsed

Return type tuple

return_path()

Make the dataset return the raw paths to the data instead of parsing it. This method returns a context manager.

Return type *AbstractContextManager*

class d3d.dataset.base.**DetectionDatasetBase**(*base_path*, *inzip=False*, *phase='training'*, *trainval_split=1.0*, *trainval_random=False*)

Bases: *d3d.dataset.base.DatasetBase*, *d3d.dataset.base.MultiModalDatasetMixin*

This class defines basic interface for object detection datasets

Parameters

- **base_path** (*Union[str, pathlib.Path]*) – directory containing the zip files, or the required data
- **inzip** (*bool*) – whether the dataset is store in original zip archives or unzipped
- **phase** (*str*) – training, validation or testing
- **trainval_split** (*Union[float, List[int]]*) – the ratio to split training dataset. See documentation of *split_trainval()* for detail.
- **trainval_random** (*Union[bool, int, str]*) – whether select the train/val split randomly. See documentation of *split_trainval()* for detail.

VALID_OBJ_CLASSES: enum.Enum

List of valid object labels

analyze_3dobject()

Report statistics on 3D object labels

Returns Statistics containing mean dimension

Return type dict

annotation_3dobject(*idx*, *raw=None*)

Return list of converted ground truth targets in lidar frame.

Parameters

- **idx** (*Union[int, tuple]*) – index of requested frame
- **raw** (*Optional[bool]*) – if false, targets will be converted to d3d *d3d.abstraction.Target3DArray* format, otherwise raw data will be returned in original format

Return type *Union*[*d3d.abstraction.Target3DArray*, *Any*]

class d3d.dataset.base.**MultiModalDatasetMixin**

Bases: object

This class defines basic interface for multi-modal datasets

VALID_CAM_NAMES: List[str]
List of valid sensor names of camera

VALID_LIDAR_NAMES: List[str]
List of valid sensor names of lidar

calibration_data(*idx*, *raw=None*)
Return the calibration data

Parameters

- **idx** (*Union[int, tuple]*) – index of requested frame
- **raw** (*Optional[bool]*) – if false, converted *d3d.abstraction.TransformSet* will be returned, otherwise raw data will be returned in original format.

Return type *Union*[*d3d.abstraction.TransformSet*, *Any*]

camera_data(*idx*, *names=None*)
Return the camera image data

Parameters

- **names** (*Optional[Union[str, List[str]]]*) – name of requested camera sensors. The default sensor is the first element in *VALID_CAM_NAMES*.
- **idx** (*Union[int, tuple]*) – index of requested image frames

Return type *Union*[PIL.Image.Image, *List*[PIL.Image.Image]]

lidar_data(*idx*, *names=None*, *formatted=False*)
Return the lidar point cloud data

Parameters

- **names** (*Optional[Union[str, List[str]]]*) – name of requested lidar sensors. The default sensor is the first element in *VALID_LIDAR_NAMES*.
- **idx** (*Union[int, tuple]*) – index of requested lidar frames
- **formatted** (*bool*) – if true, the point cloud wrapped in a numpy record array will be returned

Return type *Union*[numpy.ndarray, *List*[numpy.ndarray]]

class d3d.dataset.base.**MultiModalSequenceDatasetMixin**
Bases: object

This class defines basic interface for multi-modal datasets of sequences

calibration_data(*idx*, *raw=False*)
Return the calibration data. Notices that we assume the calibration is fixed among one squence, so it always return a single object.

Parameters

- **idx** (*Union[int, tuple]*) – index of requested lidar frames
- **raw** (*Optional[bool]*) – If false, converted *d3d.abstraction.TransformSet* will be returned, otherwise raw data will be returned in original format

Return type *Union*[*d3d.abstraction.TransformSet*, *Any*]

camera_data(*idx*, *names=None*)
Return the camera image data

Parameters

- **names** (*Optional[Union[str, List[str]]]*) – name of requested camera sensors. The default sensor is the first element in `VALID_CAM_NAMES`.
- **idx** (*Union[int, tuple]*) – index of requested image frames, see description in *lidar_data()* method.

Return type *Union*[PIL.Image.Image, *List*[PIL.Image.Image], *List*[*List*[PIL.Image.Image]]]

lidar_data(*idx*, *names=None*, *formatted=False*)

If multiple frames are requested, the results will be a list of list. Outer list corresponds to frame names and inner list corresponds to time sequence. So len(names) × len(frames) data objects will be returned

Parameters

- **names** (*Optional[Union[str, List[str]]]*) – name of requested lidar sensors. The default frame is the first element in `VALID_LIDAR_NAMES`.
- **idx** (*Union[int, tuple]*) – index of requested lidar frames
- **formatted** (*bool*) – if true, the point cloud wrapped in a numpy record array will be returned
  - If single index is given, then the frame indexing is done on the whole dataset with trainval split
  - If a tuple is given, it's considered to be a unique id of the frame (from `identity()` method), trainval split is ignored in this way and nframes offset is not added

Return type *Union*[numpy.ndarray, *List*[numpy.ndarray], *List*[*List*[numpy.ndarray]]]

class d3d.dataset.base.**SegmentationDatasetMixin**

Bases: `object`

This class define basic interface for point cloud segmentation datasets

VALID_PTS_CLASSES: *enum.Enum*

List of valid points labels

annotation_3dpoints(*idx*, *names=None*, *formatted=False*)

Return list of point-wise labels in lidar frame.

Parameters

- **idx** (*Union[int, tuple]*) – index of requested frame
- **formatted** (*Optional[bool]*) – if True, the point labels will be represented as a numpy record array, otherwise, the returned object will be a dictionary of numpy arrays.
- **names** (*Optional[Union[str, List[str]]]*) –

class d3d.dataset.base.**SequenceDatasetBase**(*base_path*, *inzip=False*, *phase='training'*, *trainval_split=1.0*, *trainval_random=False*, *trainval_byseq=False*, *nframes=0*)

Bases: *d3d.dataset.base.DatasetBase*

This class defines basic interface for datasets of sequences

Parameters

- **base_path** (*Union[str, pathlib.Path]*) – directory containing the zip files, or the required data
- **inzip** (*bool*) – whether the dataset is store in original zip archives or unzipped
- **phase** (*str*) – training, validation or testing

- **trainval_split** (*Union[float, List[int]]*) – the ratio to split training dataset. See documentation of *split_trainval_seq()* for detail.
- **trainval_random** (*Union[bool, int, str]*) – whether select the train/val split randomly. See documentation of *split_trainval_seq()* for detail.
- **nframes** (*int*) – number of consecutive frames returned from the accessors
  - If it's a positive number, then it returns adjacent frames with total number reduced
  - If it's a negative number, absolute value of it is consumed
  - If it's zero, then it act like object detection dataset, which means the methods will return unpacked data
- **trainval_byseq** – Whether split trainval partitions by sequences instead of frames

_locate_frame(*idx*)

Subclass should implement this function to convert overall index to (sequence_id, frame_idx) to support decorator *expand_idx()* and *expand_idx_name()*

Returns (seq_id, frame_idx) where frame_idx is the index of starting frame

Parameters **idx** (*int*) –

Return type *Tuple*[*Any*, int]

identity(*idx*)

Return something that can track the data back to original dataset

Parameters **idx** (*int*) – index of requested frame to be parsed

Returns if `nframes` > 0, then the function return a list of ids which are consistent with other functions.

Return type *Union*[tuple, *List*[tuple]]

intermediate_data(*idx*, *names=None*, *ninter_frames=1*)

Return the intermediate data (and annotations) between keyframes. For key frames data, please use corresponding function to load them

Parameters

- **idx** (*Union[int, tuple]*) – index of requested data frames
- **names** (*Optional[Union[str, List[str]]]*) – name of requested sensors.
- **ninter_frames** (*int*) – number of intermediate frames. If set to None, then all frames will be returned.

Return type dict

property sequence_ids: List[Any]

Return the list of sequence ids

property sequence_sizes: Dict[Any, int]

Return the mapping from sequence id to sequence sizes

timestamp(*idx*, *names=None*)

Return the timestamp of frame specified by the index, represented by Unix timestamp in macroseconds (usually 16 digits integer)

Parameters

- **idx** (*Union[int, tuple]*) – index of requested frame

- **names** (*Optional[Union[str, List[str]]]*) – specify the sensor whose pose is requested. This option only make sense when the dataset contains separate timestamps for data from each sensor.

Return type *Union*[int, *List*[int]]

class d3d.dataset.base.**TrackingDatasetBase**(*base_path*, *inzip=False*, *phase='training'*, *trainval_split=1.0*, *trainval_random=False*, *trainval_byseq=False*, *nframes=0*)

Bases: *d3d.dataset.base.SequenceDatasetBase*, *d3d.dataset.base.MultiModalSequenceDatasetMixin*

Tracking dataset is similarly defined with detection dataset. The two major differences are 1. Tracking dataset use (sequence_id, frame_id) as identifier. 2. Tracking dataset provide unique object id across time frames.

Parameters

- **base_path** (*Union[str, pathlib.Path]*) – directory containing the zip files, or the required data
- **inzip** (*bool*) – whether the dataset is store in original zip archives or unzipped
- **phase** (*str*) – training, validation or testing
- **trainval_split** (*Union[float, List[int]]*) – the ratio to split training dataset. See documentation of *split_trainval_seq()* for detail.
- **trainval_random** (*Union[bool, int, str]*) – whether select the train/val split randomly. See documentation of *split_trainval_seq()* for detail.
- **nframes** (*int*) – number of consecutive frames returned from the accessors
  - If it's a positive number, then it returns adjacent frames with total number reduced
  - If it's a negative number, absolute value of it is consumed
  - If it's zero, then it act like object detection dataset, which means the methods will return unpacked data
- **trainval_byseq** – Whether split trainval partitions by sequences instead of frames

annotation_3dobject(*idx*, *raw=False*)

Return list of converted ground truth targets in lidar frame.

Parameters

- **idx** (*Union[int, tuple]*) – index of requested frame
- **raw** (*Optional[bool]*) – if false, targets will be converted to d3d *d3d.abstraction.Target3DArray* format, otherwise raw data will be returned in original format.

Return type *Union*[*d3d.abstraction.Target3DArray*, *List*[*d3d.abstraction.Target3DArray*]]

pose(*idx*, *raw=False*, *names=None*)

Return (relative) pose of the vehicle for the frame. The base frame should be ground attached which means the base frame will follow a East-North-Up axis order.

Parameters

- **idx** (*Union[int, tuple]*) – index of requested frame
- **names** (*Optional[Union[str, List[str]]]*) – specify the sensor whose pose is requested. This option only make sense when the dataset contains separate timestamps for data from each sensor. In this case, the pose either comes from dataset, or from interpolation.

- **raw** (*Optional[bool]*) – if false, targets will be converted to d3d *d3d.abstraction.EgoPose* format, otherwise raw data will be returned in original format.

Return type *Union*[*d3d.abstraction.EgoPose*, *Any*]

property pose_name: str
Return the sensor frame name whose coordinate the pose is reported in. This frame can be different from the default frame in the calibration TransformSet.

d3d.dataset.base.**check_frames**(*names*, *valid*)
Check wether names is inside valid options.

Parameters

- **names** (*Union[List[str], str]*) – Names to be checked
- **valid** (*List[str]*) – List of the valid names

Returns unpack_result: whether need to unpack results names: frame names converted to list

d3d.dataset.base.**expand_idx**(*func*)

This decorator wraps SequenceDatasetBase member functions with index input. It will delegates the situation where self.nframe > 0 to the original function so that the original function can support only one index.

There is a parameter :obj:bypass` added to decorated function, which is used to call the original underlying method without expansion.

d3d.dataset.base.**expand_idx_name**(*valid_names*)
This decorator works similar to expand_idx with support to distribute both indices and frame names. Note that this function acts as a decorator factory instead of decorator

There is a parameter :obj:bypass` added to decorated function, which is used to call the original underlying method without expansion.

Parameters **valid_names** (*List[str]*) – List of valid sensor names

Return type *Callable*[[*Callable*], *Callable*]

d3d.dataset.base.**expand_name**(*valid_names*)
This decorator works similar to expand_idx with support to distribute frame names. Note that this function acts as a decorator factory instead of decorator

Parameters **valid_names** (*List[str]*) – List of valid sensor names

Return type *Callable*[[*Callable*], *Callable*]

d3d.dataset.base.**split_trainval**(*phase*, *total_count*, *trainval_split*, *trainval_random*)
Split frames for training or validation set

Parameters

- **phase** (*str*) – training or validation
- **total_count** (*int*) – total number of frames in trainval part of the dataset
- **trainval_split** (*Union[float, List[int]]*) – the ratio to split training dataset.
  - If it's a number, then it's the ratio to split training dataset.
  - If it's 1, then the validation set is empty; if it's 0, then training set is empty
  - If it's a list of number, then it directly defines the indices to report (ignoring trainval_random)

- **trainval_random** (*Union[bool, int, str]*) – whether select the train/val split randomly.
  - If it's a bool, then trainval is split with or without shuffle
  - If it's a number, then it's used as the seed for random shuffling
  - If it's a string, then predefined order is used. {r: reverse}

Return type *Iterable*[int]

d3d.dataset.base.**split_trainval_seq**(*phase*, *seq_counts*, *trainval_split*, *trainval_random*, *by_seq=False*)
Split frames for training or validation by frames or by sequence TODO: consider nframes

Parameters

- **phase** (*str*) – training or validation
- **total_count** – total number of frames in trainval part of the dataset
- **trainval_split** (*Union[float, List[int]]*) – the ratio to split training dataset.
  - If it's a number, then it's the ratio to split training dataset.
  - If it's 1, then the validation set is empty; if it's 0, then training set is empty
  - If it's a list of number and by_seq is True, then it directly defines the indices to report (ignoring `trainval_random`)
  - If it's a list of sequence name and by_seq is False, then it directly defines the sequences to be chosen
- **trainval_random** (*Union[bool, int, str]*) – whether select the train/val split randomly.
  - If it's a bool, then trainval is split with or without shuffle
  - If it's a number, then it's used as the seed for random shuffling
  - If it's a string, then predefined order is used. {r: reverse}
- **by_seq** (*bool*) – Whether split trainval partitions by sequences instead of frames
- **seq_counts** (*SortedCountDict*) –

Return type *Iterable*[int]

class d3d.dataset.zip.**PatchedZipFile**(*file*, *mode='r'*, *compression=0*, *allowZip64=True*, *to_extract=[]*)
Bases: zipfile.ZipFile

This class is based on build-in ZipFile class, which is further patched for better reading speed. The improvement is achieved by skip reading metadata of files not interested in.

Parameters **to_extract** (*Union[List[str], str]*) – specify the path (inside zip) of files to be extracted

Open the ZIP file with mode read 'r', write 'w', exclusive create 'x', or append 'a'.

CHAPTER

THIRTEEN

D3D.DATASET.KITTI

This module contains loaders for KITTI 3d object, KITTI tracking, KITTI raw datasets.

class d3d.dataset.kitti.**KittiObjectLoader**(*base_path*, *inzip=False*, *phase='training'*, *trainval_split=0.8*, *trainval_random=False*)

Bases: *d3d.dataset.base.DetectionDatasetBase*

Loader for KITTI object detection dataset, please organize the files into following structure

- Zip Files:

```
- data_object_calib.zip
- data_object_image_2.zip
- data_object_image_3.zip
- data_object_label_2.zip
- data_object_velodyne.zip
```

- Unzipped Structure:

```
- <base_path directory>
    - training
        - calib
        - image_2
        - label_2
        - velodyne
    - testing
        - calib
        - image_2
        - velodyne
```

Note that the 3d objects labelled as *DontCare* are removed from the result of *annotation_3dobject()*.

For description of constructor parameters, please refer to *d3d.dataset.base.DetectionDatasetBase*

Parameters

- **base_path** – directory containing the zip files, or the required data
- **inzip** – whether the dataset is store in original zip archives or unzipped
- **phase** – training, validation or testing
- **trainval_split** – the ratio to split training dataset. See documentation of `split_trainval()` for detail.
- **trainval_random** – whether select the train/val split randomly. See documentation of `split_trainval()` for detail.

VALID_OBJ_CLASSES
alias of *d3d.dataset.kitti.utils.KittiObjectClass*

annotation_3dobject(*idx*, *raw=False*)
Return list of converted ground truth targets in lidar frame.

Parameters

- **idx** – index of requested frame
- **raw** – if false, targets will be converted to d3d *d3d.abstraction.Target3DArray* format, otherwise raw data will be returned in original format

calibration_data(*idx*, *raw=False*)
Return the calibration data

Parameters

- **idx** – index of requested frame
- **raw** – if false, converted *d3d.abstraction.TransformSet* will be returned, otherwise raw data will be returned in original format.

camera_data(*idx*, *names='cam2'*)
Return the camera image data

Parameters

- **names** – name of requested camera sensors. The default sensor is the first element in VALID_CAM_NAMES.
- **idx** – index of requested image frames

dump_detection_output(*idx*, *detections*, *fout*)
Save the detection in KITTI output format. We need raw calibration for R0_rect

Parameters

- **idx** (*Union[int, tuple]*) –
- **detections** (d3d.abstraction.Target3DArray) –
- **fout** (*Union[str, pathlib.Path, io.RawIOBase]*) –

Return type None

identity(*idx*)

Return something that can track the data back to original dataset. The result tuple can be passed to any accessors above and directly access given data.

Parameters **idx** – index of requested frame to be parsed

lidar_data(*idx*, *names='velo'*, *formatted=False*)
Return the lidar point cloud data

Parameters

- **names** – name of requested lidar sensors. The default sensor is the first element in VALID_LIDAR_NAMES.
- **idx** – index of requested lidar frames
- **formatted** – if true, the point cloud wrapped in a numpy record array will be returned

class d3d.dataset.kitti.**KittiRawLoader**(*base_path*, *datatype='sync'*, *inzip=True*, *phase='training'*, *trainval_split=1*, *trainval_random=False*, *trainval_byseq=False*, *nframes=0*)

Bases: *d3d.dataset.base.TrackingDatasetBase*

Load and parse raw data into a usable format, please organize the files into following structure

- Zip Files:

```
- 2011_09_26_calib.zip [required]
- 2011_09_26_drive_0001_extract.zip
- ...
- 2011_09_26_drive_0001_sync.zip
- ...
- 2011_09_26_drive_0001_tracklets.zip
- ...
```

- Unzipped Structure:

```
- <base_path directory>
    - 2011_09_26
        - calib_cam_to_cam.txt
        - calib_imu_to_velo.txt
        - calib_velo_to_cam.txt
        - 2011_09_26_drive_0001_extract
            - image_00
            - image_01
            - image_02
            - image_03
            - oxts
            - velodyne_points
        - ...
        - 2011_09_26_drive_0001_sync
            - image_00
            - image_01
            - image_02
            - image_03
            - oxts
            - velodyne_points
            - tracklet_labels.xml
        - ...
```

For description of constructor parameters, please refer to *d3d.dataset.base.TrackingDatasetBase* Note that the 3d objects labelled as *DontCare* are removed from the result of *annotation_3dobject()*.

Parameters

- **datatype** (*str*) – 'sync' (synced) / 'extract' (unsynced)
- **base_path** – directory containing the zip files, or the required data
- **inzip** – whether the dataset is store in original zip archives or unzipped
- **phase** – training, validation or testing
- **trainval_split** – the ratio to split training dataset. See documentation of `split_trainval_seq()` for detail.

- **trainval_random** – whether select the train/val split randomly. See documentation of `split_trainval_seq()` for detail.
- **nframes** – number of consecutive frames returned from the accessors
  - If it's a positive number, then it returns adjacent frames with total number reduced
  - If it's a negative number, absolute value of it is consumed
  - If it's zero, then it act like object detection dataset, which means the methods will return unpacked data
- **trainval_byseq** – Whether split trainval partitions by sequences instead of frames

VALID_OBJ_CLASSES
alias of *d3d.dataset.kitti.utils.KittiObjectClass*

annotation_3dobject(*idx*)
Return list of converted ground truth targets in lidar frame.

Parameters

- **idx** – index of requested frame
- **raw** – if false, targets will be converted to d3d *d3d.abstraction.Target3DArray* format, otherwise raw data will be returned in original format.

calibration_data(*idx*, *raw=False*)
Return the calibration data. Notices that we assume the calibration is fixed among one squence, so it always return a single object.

Parameters

- **idx** – index of requested lidar frames
- **raw** – If false, converted *d3d.abstraction.TransformSet* will be returned, otherwise raw data will be returned in original format

camera_data(*idx*, *names='cam2'*)
Return the camera image data

Parameters

- **names** – name of requested camera sensors. The default sensor is the first element in `VALID_CAM_NAMES`.
- **idx** – index of requested image frames, see description in *lidar_data()* method.

identity(*idx*)
Return something that can track the data back to original dataset

Parameters **idx** – index of requested frame to be parsed

Returns if `nframes` > 0, then the function return a list of ids which are consistent with other functions.

lidar_data(*idx*, *names='velo'*, *formatted=False*)
If multiple frames are requested, the results will be a list of list. Outer list corresponds to frame names and inner list corresponds to time sequence. So len(names) × len(frames) data objects will be returned

Parameters

- **names** – name of requested lidar sensors. The default frame is the first element in `VALID_LIDAR_NAMES`.
- **idx** – index of requested lidar frames

- **formatted** – if true, the point cloud wrapped in a numpy record array will be returned
  - If single index is given, then the frame indexing is done on the whole dataset with trainval split
  - If a tuple is given, it's considered to be a unique id of the frame (from *identity()* method), trainval split is ignored in this way and nframes offset is not added

pose(*idx*, *raw=False*)

Return (relative) pose of the vehicle for the frame. The base frame should be ground attached which means the base frame will follow a East-North-Up axis order.

Parameters

- **idx** – index of requested frame
- **names** – specify the sensor whose pose is requested. This option only make sense when the dataset contains separate timestamps for data from each sensor. In this case, the pose either comes from dataset, or from interpolation.
- **raw** – if false, targets will be converted to d3d *d3d.abstraction.EgoPose* format, otherwise raw data will be returned in original format.

property pose_name

Return the sensor frame name whose coordinate the pose is reported in. This frame can be different from the default frame in the calibration TransformSet.

property sequence_ids

Return the list of sequence ids

property sequence_sizes

Return the mapping from sequence id to sequence sizes

timestamp(*idx*, *names='velo'*)

Return the timestamp of frame specified by the index, represented by Unix timestamp in macroseconds (usually 16 digits integer)

Parameters

- **idx** – index of requested frame
- **names** – specify the sensor whose pose is requested. This option only make sense when the dataset contains separate timestamps for data from each sensor.

class d3d.dataset.kitti.**KittiTrackingLoader**(*base_path*, *inzip=False*, *phase='training'*, *trainval_split=0.8*, *trainval_random=False*, *trainval_byseq=False*, *nframes=0*)

Bases: *d3d.dataset.base.TrackingDatasetBase*

Loader for KITTI multi-object tracking dataset, please organize the files into following structure

- Zip Files:

```
- data_tracking_calib.zip
- data_tracking_image_2.zip
- data_tracking_image_3.zip
- data_tracking_label_2.zip
- data_tracking_velodyne.zip
- data_tracking_oxts.zip
```

- Unzipped Structure:

```
- <base_path directory>
    - training
        - calib
        - image_02
        - label_02
        - oxts
        - velodyne
    - testing
        - calib
        - image_02
        - oxts
        - velodyne
```

For description of constructor parameters, please refer to *d3d.dataset.base.TrackingDatasetBase*. Note that the 3d objects labelled as *DontCare* are removed from the result of *annotation_3dobject()*.

Parameters

- **base_path** – directory containing the zip files, or the required data
- **inzip** – whether the dataset is store in original zip archives or unzipped
- **phase** – training, validation or testing
- **trainval_split** – the ratio to split training dataset. See documentation of `split_trainval_seq()` for detail.
- **trainval_random** – whether select the train/val split randomly. See documentation of `split_trainval_seq()` for detail.
- **nframes** – number of consecutive frames returned from the accessors
  - If it's a positive number, then it returns adjacent frames with total number reduced
  - If it's a negative number, absolute value of it is consumed
  - If it's zero, then it act like object detection dataset, which means the methods will return unpacked data
- **trainval_byseq** – Whether split trainval partitions by sequences instead of frames

VALID_OBJ_CLASSES
alias of *d3d.dataset.kitti.utils.KittiObjectClass*

annotation_3dobject(*idx*, *raw=False*)
Return list of converted ground truth targets in lidar frame.

Parameters

- **idx** – index of requested frame
- **raw** – if false, targets will be converted to d3d *d3d.abstraction.Target3DArray* format, otherwise raw data will be returned in original format.

calibration_data(*idx*, *raw=False*)
Return the calibration data. Notices that we assume the calibration is fixed among one squence, so it always return a single object.

Parameters

- **idx** – index of requested lidar frames

- **raw** – If false, converted *d3d.abstraction.TransformSet* will be returned, otherwise raw data will be returned in original format

camera_data(*idx*, *names='cam2'*)
Return the camera image data

Parameters

- **names** – name of requested camera sensors. The default sensor is the first element in VALID_CAM_NAMES.
- **idx** – index of requested image frames, see description in *lidar_data()* method.

identity(*idx*)
Return something that can track the data back to original dataset

Parameters **idx** – index of requested frame to be parsed

Returns if nframes > 0, then the function return a list of ids which are consistent with other functions.

lidar_data(*idx*, *names='velo'*, *formatted=False*)
If multiple frames are requested, the results will be a list of list. Outer list corresponds to frame names and inner list corresponds to time sequence. So len(names) × len(frames) data objects will be returned

Parameters

- **names** – name of requested lidar sensors. The default frame is the first element in VALID_LIDAR_NAMES.
- **idx** – index of requested lidar frames
- **formatted** – if true, the point cloud wrapped in a numpy record array will be returned
  - If single index is given, then the frame indexing is done on the whole dataset with trainval split
  - If a tuple is given, it's considered to be a unique id of the frame (from *identity()* method), trainval split is ignored in this way and nframes offset is not added

pose(*idx*, *raw=False*)
Return (relative) pose of the vehicle for the frame. The base frame should be ground attached which means the base frame will follow a East-North-Up axis order.

Parameters

- **idx** – index of requested frame
- **names** – specify the sensor whose pose is requested. This option only make sense when the dataset contains separate timestamps for data from each sensor. In this case, the pose either comes from dataset, or from interpolation.
- **raw** – if false, targets will be converted to d3d *d3d.abstraction.EgoPose* format, otherwise raw data will be returned in original format.

property pose_name
Return the sensor frame name whose coordinate the pose is reported in. This frame can be different from the default frame in the calibration TransformSet.

property sequence_ids
Return the list of sequence ids

property sequence_sizes
Return the mapping from sequence id to sequence sizes

timestamp(*idx*, *names='velo'*)

Return the timestamp of frame specified by the index, represented by Unix timestamp in macroseconds (usually 16 digits integer)

Parameters

- **idx** – index of requested frame
- **names** – specify the sensor whose pose is requested. This option only make sense when the dataset contains separate timestamps for data from each sensor.

class d3d.dataset.kitti.**KittiObjectClass**(*value*)

Bases: enum.Enum

Category of objects in KITTI dataset

Car = 1

Cyclist = 6

DontCare = 0

Misc = 8

Pedestrian = 4

Person = 5

Person_sitting = 5

Tram = 7

Truck = 3

Van = 2

d3d.dataset.kitti.object.**execute_official_evaluator**(*exec_path*, *label_path*, *result_path*, *output_path*, *model_name=None*, *show_output=True*)

Execute official evaluator from KITTI devkit

Parameters

- **label_path** – path to the extracted labels of kitti dataset
- **result_path** – path to the results created by dump_detection_output
- **model_name** – unique name of your model. KITTI tool requires sha1 as model name, but that's not mandatory.
- **show_output** – show_output option passed to Kitti evaluator

Parma output_path path to the output evaluation results

Note: to install prerequisites *sudo apt install gnuplot texlive-extra-utils*

CHAPTER

FOURTEEN

D3D.DATASET.KITTI360

class d3d.dataset.kitti360.**KITTI360Loader**(*base_path*, *phase='training'*, *inzip=False*, *trainval_split=1*, *trainval_random=False*, *trainval_byseq=False*, *nframes=0*, *interpolate_pose=True*, *compression=0*)

Bases: *d3d.dataset.base.TrackingDatasetBase*

Load KITTI-360 dataset into a usable format. The dataset structure should follow the official documents.

- Zip Files:

```
- calibration.zip
- data_3d_bboxes.zip
- data_3d_semantics.zip
- data_poses.zip
- data_timestamps_sick.zip
- data_timestamps_velodyne.zip
- 2013_05_28_drive_0000_sync_sick.zip
- 2013_05_28_drive_0000_sync_velodyne.zip
- ...
```

- Unzipped Structure:

```
- <base_path directory>
    - calibration
    - data_2d_raw
        - 2013_05_28_drive_0000_sync
        - ...
    - data_2d_semantics
        - 2013_05_28_drive_0000_sync
        - ...
    - data_3d_raw
        - 2013_05_28_drive_0000_sync
        - ...
    - data_3d_semantics
        - 2013_05_28_drive_0000_sync
        - ...
```

For description of constructor parameters, please refer to *d3d.dataset.base.TrackingDatasetBase*

Parameters

- **interpolate_pose** (*bool*) – Not all frames contain pose data in KITTI-360. The loader returns interpolated pose if this param is set as True, otherwise returns None

- **compression** (*int*) – The compression type of the created zip for semantic files. It should be one of the compression types specified in `zipfile` module.
- **base_path** – directory containing the zip files, or the required data
- **inzip** – whether the dataset is store in original zip archives or unzipped
- **phase** – training, validation or testing
- **trainval_split** – the ratio to split training dataset. See documentation of `split_trainval_seq()` for detail.
- **trainval_random** – whether select the train/val split randomly. See documentation of `split_trainval_seq()` for detail.
- **nframes** – number of consecutive frames returned from the accessors
  - If it's a positive number, then it returns adjacent frames with total number reduced
  - If it's a negative number, absolute value of it is consumed
  - If it's zero, then it act like object detection dataset, which means the methods will return unpacked data
- **trainval_byseq** – Whether split trainval partitions by sequences instead of frames

VALID_OBJ_CLASSES
alias of *d3d.dataset.kitti360.utils.Kitti360Class*

annotation_3dobject(*idx*, *raw=False*, *visible_range=80*)

Parameters **visible_range** – range for visible objects. Objects beyond that distance will be removed when reporting

calibration_data(*idx*)
Return the calibration data. Notices that we assume the calibration is fixed among one squence, so it always return a single object.

Parameters

- **idx** – index of requested lidar frames
- **raw** – If false, converted *d3d.abstraction.TransformSet* will be returned, otherwise raw data will be returned in original format

camera_data(*idx*, *names='cam1'*)
Return the camera image data

Parameters

- **names** – name of requested camera sensors. The default sensor is the first element in `VALID_CAM_NAMES`.
- **idx** – index of requested image frames, see description in *lidar_data()* method.

intermediate_data(*idx*, *names='sick'*, *ninter_frames=None*, *report_semantic=True*)
Return the intermediate data (and annotations) between keyframes. For key frames data, please use corresponding function to load them

Parameters

- **idx** – index of requested data frames
- **names** – name of requested sensors.

- **ninter_frames** – number of intermediate frames. If set to None, then all frames will be returned.

lidar_data(*idx*, *names='velo'*, *formatted=False*)

If multiple frames are requested, the results will be a list of list. Outer list corresponds to frame names and inner list corresponds to time sequence. So len(names) × len(frames) data objects will be returned

Parameters

- **names** – name of requested lidar sensors. The default frame is the first element in `VALID_LIDAR_NAMES`.
- **idx** – index of requested lidar frames
- **formatted** – if true, the point cloud wrapped in a numpy record array will be returned
  - If single index is given, then the frame indexing is done on the whole dataset with trainval split
  - If a tuple is given, it's considered to be a unique id of the frame (from `identity()` method), trainval split is ignored in this way and nframes offset is not added

pose(*idx*)

Return (relative) pose of the vehicle for the frame. The base frame should be ground attached which means the base frame will follow a East-North-Up axis order.

Parameters

- **idx** – index of requested frame
- **names** – specify the sensor whose pose is requested. This option only make sense when the dataset contains separate timestamps for data from each sensor. In this case, the pose either comes from dataset, or from interpolation.
- **raw** – if false, targets will be converted to d3d *d3d.abstraction.EgoPose* format, otherwise raw data will be returned in original format.

property pose_name

Return the sensor frame name whose coordinate the pose is reported in. This frame can be different from the default frame in the calibration TransformSet.

property sequence_ids

Return the list of sequence ids

property sequence_sizes

Return the mapping from sequence id to sequence sizes

timestamp(*idx*, *names='velo'*)

Return the timestamp of frame specified by the index, represented by Unix timestamp in macroseconds (usually 16 digits integer)

Parameters

- **idx** – index of requested frame
- **names** – specify the sensor whose pose is requested. This option only make sense when the dataset contains separate timestamps for data from each sensor.

class d3d.dataset.kitti360.**Kitti360Class**(*value*)

Bases: enum.IntFlag

Categories and attributes of an annotation in KITTI-360 dataset.

The ids are encoded into 2bytes integer:

```
0xFF
  ├└: category
  └─: label
```

bicycle = 135

box = 144

bridge = 82

building = 18

bus = 55

car = 23

caravan = 71

construction = 2

dynamic = 96

ego_vehicle = 32

fence = 50

flat = 1

garage = 112

gate = 128

ground = 112

guard_rail = 66

human = 6

lamp = 96

license_plate = 151

motorcycle = 119

nature = 4

object_ = 3

out_of_roi = 64

parking = 49

person = 22

pole = 19

polegroup = 35

rail_track = 65

rectification_border = 48

rider = 38

road = 17

sidewalk = 33

sky = 5

```
smallpole = 80
static = 80
stop = 144
terrain = 36
traffic_light = 51
traffic_sign = 67
trailer = 87
train = 103
trash_bin = 112
truck = 39
tunnel = 98
unknown_construction = 128
unknown_object = 160
unknown_vehicle = 144
unlabeled = 16
vegetation = 20
vehicle = 7
vending_machine = 128
void = 0
wall = 34
```

CHAPTER

FIFTEEN

D3D.DATASET.NUSCENES

class d3d.dataset.nuscenes.loader.**NuscenesLoader**(*base_path*, *inzip=False*, *phase='training'*, *trainval_split='official'*, *trainval_random=False*, *trainval_byseq=False*, *nframes=0*)

Bases: *d3d.dataset.base.TrackingDatasetBase*

Load Nuscenes dataset into a usable format. Please use the d3d_nuscenes_convert command to convert the dataset first into following formats

- Directory Structure:

```
- <base_path directory>
    - trainval
        - scene_xxx(.zip)
        - ...
    - test
        - scene_xxx(.zip)
        - ...
```

For description of constructor parameters, please refer to *d3d.dataset.base.TrackingDatasetBase*

Parameters

- **base_path** – directory containing the zip files, or the required data
- **inzip** – whether the dataset is store in original zip archives or unzipped
- **phase** – training, validation or testing
- **trainval_split** – the ratio to split training dataset. See documentation of `split_trainval_seq()` for detail.
- **trainval_random** – whether select the train/val split randomly. See documentation of `split_trainval_seq()` for detail.
- **nframes** – number of consecutive frames returned from the accessors
  - If it's a positive number, then it returns adjacent frames with total number reduced
  - If it's a negative number, absolute value of it is consumed
  - If it's zero, then it act like object detection dataset, which means the methods will return unpacked data
- **trainval_byseq** – Whether split trainval partitions by sequences instead of frames

VALID_OBJ_CLASSES

alias of *d3d.dataset.nuscenes.constants.NuscenesDetectionClass*

VALID_PTS_CLASSES
alias of d3d.dataset.nuscenes.constants.NuscenesSegmentationClass

annotation_3dobject(*idx*, *raw=False*, *convert_tag=True*, *with_velocity=True*)
Return list of converted ground truth targets in lidar frame.

Parameters

- **idx** – index of requested frame
- **raw** – if false, targets will be converted to d3d *d3d.abstraction.Target3DArray* format, otherwise raw data will be returned in original format.

annotation_3dpoints(*idx*, *names='lidar_top'*, *parse_tag=True*, *convert_tag=True*)

Parameters

- **parse_tag** – Parse tag from original nuscenes id defined in category.json into *NuscenesObjectClass*
- **convert_tag** – Convert tag from *NuscenesObjectClass* to NuscenesSegmentationClass

calibration_data(*idx*)
Return the calibration data. Notices that we assume the calibration is fixed among one squence, so it always return a single object.

Parameters

- **idx** – index of requested lidar frames
- **raw** – If false, converted *d3d.abstraction.TransformSet* will be returned, otherwise raw data will be returned in original format

camera_data(*idx*, *names=None*)
Return the camera image data

Parameters

- **names** – name of requested camera sensors. The default sensor is the first element in VALID_CAM_NAMES.
- **idx** – index of requested image frames, see description in *lidar_data()* method.

dump_segmentation_output(*idx*, *segmentation*, *folder_out*, *raw2seg=True*, *default_class=15*)

Parameters

- **raw2seg** (*bool*) – If set as true, input array will be considered as raw id (consistent with values stored in label)
- **default_class** (*Union[int, d3d.dataset.nuscenes.constants.NuscenesSegmentationClass]*) – Class to be selected when the label is 0 (ignore)
- **idx** (*Union[int, tuple]*) –
- **segmentation** (*numpy.ndarray*) –
- **folder_out** (*str*) –

identity(*idx*)
Return something that can track the data back to original dataset

Parameters **idx** – index of requested frame to be parsed

Returns if `nframes` > 0, then the function return a list of ids which are consistent with other functions.

`intermediate_data`(*idx*, *names=None*, *ninter_frames=None*, *formatted=False*)
Return the intermediate data (and annotations) between keyframes. For key frames data, please use corresponding function to load them

Parameters

- **idx** – index of requested data frames
- **names** – name of requested sensors.
- **ninter_frames** – number of intermediate frames. If set to None, then all frames will be returned.

`lidar_data`(*idx*, *names='lidar_top'*, *formatted=False*)
If multiple frames are requested, the results will be a list of list. Outer list corresponds to frame names and inner list corresponds to time sequence. So len(names) × len(frames) data objects will be returned

Parameters

- **names** – name of requested lidar sensors. The default frame is the first element in `VALID_LIDAR_NAMES`.
- **idx** – index of requested lidar frames
- **formatted** – if true, the point cloud wrapped in a numpy record array will be returned
  - If single index is given, then the frame indexing is done on the whole dataset with trainval split
  - If a tuple is given, it's considered to be a unique id of the frame (from *identity()* method), trainval split is ignored in this way and nframes offset is not added

`pose`(*idx*, *names='lidar_top'*, *raw=False*)
Return (relative) pose of the vehicle for the frame. The base frame should be ground attached which means the base frame will follow a East-North-Up axis order.

Parameters

- **idx** – index of requested frame
- **names** – specify the sensor whose pose is requested. This option only make sense when the dataset contains separate timestamps for data from each sensor. In this case, the pose either comes from dataset, or from interpolation.
- **raw** – if false, targets will be converted to d3d *d3d.abstraction.EgoPose* format, otherwise raw data will be returned in original format.

`property pose_name`
Return the sensor frame name whose coordinate the pose is reported in. This frame can be different from the default frame in the calibration TransformSet.

`property sequence_ids`
Return the list of sequence ids

`property sequence_sizes`
Return the mapping from sequence id to sequence sizes

`timestamp`(*idx*, *names='lidar_top'*)
Return the timestamp of frame specified by the index, represented by Unix timestamp in macroseconds (usually 16 digits integer)

Parameters

- **idx** – index of requested frame
- **names** – specify the sensor whose pose is requested. This option only make sense when the dataset contains separate timestamps for data from each sensor.

token(*idx*, *names='lidar_top'*)
Return the sample data token in original Nuscenes data given data index and sensor name

class d3d.dataset.nuscenes.loader.**NuscenesObjectClass**(*value*)
Bases: enum.IntFlag

Categories and attributes of an annotation in Nuscenes dataset.

The ids are encoded into 4bytes integer:

```
0xFFFF
  ||||: level0 category
  |||└: level1 category
  ||└-: level2 category
  |└--: attribute
  └---: attribute
```

animal = 1

property attribute
The attribute of the label

property attribute_name
Name of the attribute of the label

property category
The category of the label

property category_name
Name of the category of the label

property color

cycle_with_rider = 16384

cycle_without_rider = 20480

flat = 6

flat_driveable_surface = 22

flat_other = 70

flat_sidewalk = 38

flat_terrain = 54

classmethod from_nuscenes_id(*nid*)
Get Nuscenes class object from Nuscenes ID

Parameters **nid** (*int*) –

human = 2

human_pedestrian = 18

human_pedestrian_adult = 274

human_pedestrian_child = 530

human_pedestrian_construction_worker = 786

human_pedestrian_personal_mobility = 1042

human_pedestrian_police_officer = 1298

human_pedestrian_stroller = 1554

human_pedestrian_wheelchair = 1810

movable_object = 3

movable_object_barrier = 19

movable_object_debris = 35

movable_object_pushable_pullable = 51

movable_object_trafficcone = 67

noise = 16

property nuscenes_id
Get the Nuscenes ID of the label

classmethod parse(*string*)
Parse the Nuscenes class from string

Parameters **string** (*str*) –

pedestrian_moving = 32768

pedestrian_sitting_lying_down = 24576

pedestrian_standing = 28672

property pretty_name
Get the full name of the label with category and attribute

static = 7

static_manmade = 23

static_object = 5

static_object_bicycle_rack = 21

static_other = 55

static_vegetation = 39

to_detection()
Convert the label to the class for detection

to_segmentation()
Convert the label to the class for segmentation

Reference: https://github.com/nutonomy/nuscenes-devkit/blob/master/python-sdk/nuscenes/eval/lidarseg/README.md

unknown = 0

vehicle_bicycle = 4

vehicle_bus = 20

vehicle_bus_bendy = 276

vehicle_bus_rigid = 532

vehicle_car = 36

vehicle_construction = 52

vehicle_ego = 132

vehicle_emergency = 68

vehicle_emergency_ambulance = 324

vehicle_emergency_police = 580

vehicle_motorcycle = 84

vehicle_moving = 4096

vehicle_parked = 12288

vehicle_stopped = 8192

vehicle_trailer = 100

vehicle_truck = 116

class d3d.dataset.nuscenes.loader.**NuscenesDetectionClass**(*value*)

Bases: enum.Enum

Label classes for detection in Nuscenes dataset.

barrier = 1

bicycle = 2

bus = 3

car = 4

property color

construction_vehicle = 5

ignore = 0

motorcycle = 6

pedestrian = 7

traffic_cone = 8

trailer = 9

truck = 10

CHAPTER

SIXTEEN

D3D.DATASET.WAYMO

class d3d.dataset.waymo.loader.**WaymoLoader**(*base_path*, *phase='training'*, *inzip=False*, *trainval_split=None*, *trainval_random=False*, *nframes=0*)

Bases: *d3d.dataset.base.TrackingDatasetBase*

Load Waymo dataset into a usable format. Please use the d3d_waymo_convert command to convert the dataset first into following formats

- Directory Structure:

```
- <base_path directory>
    - training
        - xxxxxxxxxxxxxxxxxxx_xxx_xxx_xxx_xxx(.zip)
        - ...
    - validation
        - xxxxxxxxxxxxxxxxxxx_xxx_xxx_xxx_xxx(.zip)
        - ...
```

For description of constructor parameters, please refer to *d3d.dataset.base.TrackingDatasetBase*

Parameters

- **base_path** – directory containing the zip files, or the required data
- **inzip** – whether the dataset is store in original zip archives or unzipped
- **phase** – training, validation or testing
- **trainval_split** – the ratio to split training dataset. See documentation of `split_trainval_seq()` for detail.
- **trainval_random** – whether select the train/val split randomly. See documentation of `split_trainval_seq()` for detail.
- **nframes** – number of consecutive frames returned from the accessors
  - If it's a positive number, then it returns adjacent frames with total number reduced
  - If it's a negative number, absolute value of it is consumed
  - If it's zero, then it act like object detection dataset, which means the methods will return unpacked data
- **trainval_byseq** – Whether split trainval partitions by sequences instead of frames

VALID_OBJ_CLASSES

alias of *d3d.dataset.waymo.loader.WaymoObjectClass*

annotation_3dobject(*idx*, *raw=False*)

Return list of converted ground truth targets in lidar frame.

Parameters

- **idx** – index of requested frame
- **raw** – if false, targets will be converted to d3d *d3d.abstraction.Target3DArray* format, otherwise raw data will be returned in original format.

calibration_data(*idx*)

Return the calibration data. Notices that we assume the calibration is fixed among one squence, so it always return a single object.

Parameters

- **idx** – index of requested lidar frames
- **raw** – If false, converted *d3d.abstraction.TransformSet* will be returned, otherwise raw data will be returned in original format

camera_data(*idx*, *names=None*)

Parameters **names** – frame names of camera to be loaded

dump_detection_output(*idx*, *detections*, *fout*)

Parameters

- **detections** (d3d.abstraction.Target3DArray) – detection result
- **ids** – auxiliary information for output, each item contains context name and timestamp
- **fout** (*io.RawIOBase*) – output file-like object
- **idx** (*Union[int, tuple]*) –

Return type None

identity(*idx*)

Return something that can track the data back to original dataset

Parameters **idx** – index of requested frame to be parsed

Returns if `nframes` > 0, then the function return a list of ids which are consistent with other functions.

lidar_data(*idx*, *names=None*, *formatted=False*)

If multiple frames are requested, the results will be a list of list. Outer list corresponds to frame names and inner list corresponds to time sequence. So len(names) × len(frames) data objects will be returned

Parameters

- **names** – name of requested lidar sensors. The default frame is the first element in `VALID_LIDAR_NAMES`.
- **idx** – index of requested lidar frames
- **formatted** – if true, the point cloud wrapped in a numpy record array will be returned
  - If single index is given, then the frame indexing is done on the whole dataset with trainval split
  - If a tuple is given, it's considered to be a unique id of the frame (from *identity()* method), trainval split is ignored in this way and nframes offset is not added

pose(*idx*, *raw=False*)
Return (relative) pose of the vehicle for the frame. The base frame should be ground attached which means the base frame will follow a East-North-Up axis order.

Parameters

- **idx** – index of requested frame
- **names** – specify the sensor whose pose is requested. This option only make sense when the dataset contains separate timestamps for data from each sensor. In this case, the pose either comes from dataset, or from interpolation.
- **raw** – if false, targets will be converted to d3d *d3d.abstraction.EgoPose* format, otherwise raw data will be returned in original format.

property pose_name
Return the sensor frame name whose coordinate the pose is reported in. This frame can be different from the default frame in the calibration TransformSet.

property sequence_ids
Return the list of sequence ids

property sequence_sizes
Return the mapping from sequence id to sequence sizes

timestamp(*idx*)
Return the timestamp of frame specified by the index, represented by Unix timestamp in macroseconds (usually 16 digits integer)

Parameters

- **idx** – index of requested frame
- **names** – specify the sensor whose pose is requested. This option only make sense when the dataset contains separate timestamps for data from each sensor.

d3d.dataset.waymo.loader.**create_submission**(*result_path*, *output_file*, *exec_path*, *meta_path*, *model_name=None*)
Execute create_submission from waymo_open_dataset :param exec_path: path to create_submission executable from waymo devkit :param result_path: path (or list of path) to detection result in binary protobuf from dump_detection_output :param meta_path: path to the metadata file (example: waymo_open_dataset/metrics/tools/submission.txtpb) :param output_path: output path for the created submission archive

d3d.dataset.waymo.loader.**execute_official_evaluator**(*exec_path*, *label_path*, *result_path*, *output_path*, *model_name=None*, *show_output=True*)
Execute compute_detection_metrics_main from waymo_open_dataset :param exec_path: path to compute_detection_metrics_main

class d3d.dataset.waymo.loader.**WaymoObjectClass**(*value*)
Bases: enum.Enum

Category of objects in KITTI dataset

Cyclist = 4

Pedestrian = 2

Sign = 3

Unknown = 0

Vehicle = 1

CHAPTER

SEVENTEEN

D3D.MATH

This module contains implementation of special math functions.

class d3d.math.**I0Exp**

Bases: torch.autograd.function.Function

d3d.math.**i0e**(x)

Pytorch Autograd Function of modified Bessel function with order 0

CHAPTER

EIGHTEEN

D3D.POINT

This module contains implementation of autograd operations performed directly on point cloud.

class d3d.point.**AlignedScatter**

Bases: torch.autograd.function.Function

d3d.point.**aligned_scatter**(*coordinates*, *feature_map*, *method='drop'*)

Gather the values given coordinates in feature_map.

Parameters

- **feature_map** – should have shape B x C x D1 x D2… x Dm, the number of remaining dimensions (m) should be consistent with coordinate dimension
- **coordinates** – should have shape N x (m+1), it should contains batch index at the first dimension
- **method** – drop - directly convert decimal coordinates to integers mean - mean of surrounding values linear - (bi)linear interpolation of surrounding values max - maximum of surrounding values nearest - value of nearest value

Returns extracted features with shape N x C

Note: right now only *drop*, *mean* and *linear* are implemented

CHAPTER

NINETEEN

D3D.TRACKING

This module contains implementation of tracking algorithms and utilities.

19.1 d3d.tracking.tracker

class d3d.tracking.tracker.**VanillaTracker**(*pose_tracker_factory=<class 'd3d.tracking.filter.Pose_3DOF_UKF_CTRA'>, feature_tracker_factory=<class 'd3d.tracking.filter.Box_KF'>, matcher_factory=<class 'd3d.tracking.matcher.HungarianMatcher'>, matcher_distance_type='position', matcher_distance_threshold=1, lost_time=1, default_position_var=array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]]), default_dimension_var=array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]]), default_orientation_var=1*)

Bases: object

Implementation of a vanilla tracker based on Kalman Filter

Parameters

- **lost_time** – determine the time length of a target being lost before it's removed from tracking
- **pose_tracker_factory** – factory function to generate a new pose tracker, takes only initial detection as input
- **feature_tracker_factory** – factory function to generate a new feature tracker, takes only initial detection as input
- **matcher_factory** – factory function to generate a new target matcher
- **matcher_distance_type** – distance type used to match targets
- **matcher_distance_threshold** – distance threshold used in target matcher
- **default_position_var** – default positional covariance assigned to targets (if not provided)
- **default_dimension_var** – default dimensional covariance assigned to targets (if not provided)
- **default_orientation_var** – default orientational covariance assigned to targets (if not provided)

report()
Return the collection of valid tracked targets

Return type *d3d.abstraction.Target3DArray*

property tracked_ids
Return a ID list of actively tracked targets

update(*detections*)
Update the filters when receiving new detections

Parameters **detections** (d3d.abstraction.Target3DArray) –

19.2 d3d.tracking.matcher

class d3d.tracking.matcher.**BaseMatcher**
Bases: object

This class is the base class of various matchers

clear_match(*self*) → void
Clear saved match results

match(*self*, *vector[int] src_subset*, *vector[int] dst_subset*, *unordered_map[int*, *float] distance_threshold*) → void

Parameters

- **src_subset** – Indices of source boxes to be considered
- **dst_subset** – Indices of destination boxes to be considered
- **distance_threshold** – threshold of maximum distance for each category. The category should be represented as its value.

num_of_matches(*self*) → int

prepare_boxes(*self*, *Target3DArray src_boxes*, *Target3DArray dst_boxes*, *DistanceTypes distance_metric*) → void
This method add two arrays of boxes and prepare related informations, it will also clean previous results.

Parameters

- **src_boxes** – boxes to match
- **dst_boxes** – fixed boxes (such as ground truth boxes)
- **distance_metric** – the metric for calculating the "distance"

query_dst_match(*self*, *int dst_idx*) → int

query_src_match(*self*, *int src_idx*) → int

class d3d.tracking.matcher.**HungarianMatcher**
Bases: *d3d.tracking.matcher.BaseMatcher*

This matcher select target correspondences using the Hungarian Algorithm

match(*self*, *vector[int] src_subset*, *vector[int] dst_subset*, *unordered_map[int*, *float] distance_threshold*) → void

class d3d.tracking.matcher.**NearestNeighborMatcher**
Bases: *d3d.tracking.matcher.BaseMatcher*

This matcher select target correspondences from closest pair to farthest pair

match(*self*, *vector[int] src_subset*, *vector[int] dst_subset*, *unordered_map[int, float] distance_threshold*) → void

class d3d.tracking.matcher.**ScoreMatcher**
Bases: *d3d.tracking.matcher.BaseMatcher*

This matcher select target correspondences from highest score to lowest score

match(*self*, *vector[int] src_subset*, *vector[int] dst_subset*, *unordered_map[int, float] distance_threshold*) → void

19.3 d3d.tracking.filter

class d3d.tracking.filter.**Box_KF**(*init*, *Q=array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])*)
Bases: *d3d.tracking.filter.PropertyFilter*

Use kalman filter (simple bayesian filter) for a box shape estimation. Use latest value for classification result

Parameters

- **init** (*Union[d3d.abstraction.ObjectTarget3D, d3d.abstraction.TrackingTarget3D]*) – initial state for the target
- **Q** (*numpy.ndarray*) – system process noise

property classification
Current classification estimation

property classification_var
Covariance for current classification estimation

property dimension
Current dimension estimation

property dimension_var
Covariance for current shape estimation

predict(*dt*)
Predict the current system state

Parameters **dt** – Time since last update

update(*target*)
Correct the current system state with observation. This method should always be called after calling *predict()*.

Parameters **target** – New target observation

class d3d.tracking.filter.**PoseFilter**
Bases: object

Abstraction class defining the interfaces for filters on target pose

property angular_velocity: numpy.ndarray
Current angular velocity estimation

property angular_velocity_var: numpy.ndarray
Covariance for current angular velocity esimation

property orientation: numpy.ndarray
Current orientation estimation

property orientation_var: numpy.ndarray
Covariance for current orientation estimation

property position: numpy.ndarray
Current position estimation

property position_var: numpy.ndarray
Covariance for current position estimation

predict(*dt*)
Predict the current system state

Parameters **dt** (*float*) – Time since last update

update(*target*)
Correct the current system state with observation. This method should always be called after calling *predict()*.

Parameters **target** (*Union[d3d.abstraction.ObjectTarget3D, d3d.abstraction.TrackingTarget3D]*) – New target observation

property velocity: numpy.ndarray
Current linear velocity estimation

property velocity_var: numpy.ndarray
Covariance for current linear velocity estimation

class d3d.tracking.filter.**Pose_3DOF_UKF_CTRA**(*init*, *Q=array([[1.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 1.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 1.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 1.0]])*)

Bases: *d3d.tracking.filter.PoseFilter*

UKF using constant turning rate and acceleration (CTRA) model for pose estimation

States: ..

[x, y, rz, v, a, w] [0 1 2 3 4 5]

Observe: ..

[x, y, rz] [0 1 2]

Parameters

- **init** (*Union[d3d.abstraction.ObjectTarget3D, d3d.abstraction.TrackingTarget3D]*) – Initial state for the target
- **Q** (*numpy.ndarray*) – System process noise

property angular_velocity
Current angular velocity estimation

property angular_velocity_var
Covariance for current angular velocity esimation

property orientation
Current orientation estimation

property orientation_var
Covariance for current orientation estimation

property position
Current position estimation

property position_var
Covariance for current position estimation

predict(*dt*)
Predict the current system state

Parameters **dt** – Time since last update

update(*detection*)
Correct the current system state with observation. This method should always be called after calling *predict()*.

Parameters

- **target** – New target observation
- **detection** (d3d.abstraction.ObjectTarget3D) –

property velocity
Current linear velocity estimation

property velocity_var
Covariance for current linear velocity estimation

class d3d.tracking.filter.**Pose_3DOF_UKF_CTRV**
Bases: *d3d.tracking.filter.PoseFilter*

UKF using constant turning rate and velocity (CTRV) model for pose estimation

States: ..

[x, y, rz, v, w] [0 1 2 3 4]

Observe: ..

[x, y, rz] [0 1 2]

class d3d.tracking.filter.**Pose_3DOF_UKF_CV**(*init*, *Q=array([[1.0, 0.0, 0.0, 0.0], [0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0], [0.0, 0.0, 0.0, 1.0]])*)
Bases: *d3d.tracking.filter.PoseFilter*

UKF using constant velocity (CV) model for pose estimation, assuming 3DoF (x, y, yaw)

States: ..

[x, y, vx, vy] [0 1 2 3]

Observe: ..

[x, y] [0 1]

Parameters

- **init** (*Union*[d3d.abstraction.ObjectTarget3D, d3d.abstraction.TrackingTarget3D]) – initial state for the target
- **Q** (*numpy.ndarray*) – system process noise

property angular_velocity
Current angular velocity estimation

property angular_velocity_var
Covariance for current angular velocity esimation

property orientation
Current orientation estimation

property orientation_var
Covariance for current orientation estimation

property position
Current position estimation

property position_var
Covariance for current position estimation

predict(*dt*)
Predict the current system state

Parameters **dt** – Time since last update

update(*detection*)
Correct the current system state with observation. This method should always be called after calling *predict()*.

Parameters

- **target** – New target observation
- **detection** (d3d.abstraction.ObjectTarget3D) –

property velocity
Current linear velocity estimation

property velocity_var
Covariance for current linear velocity estimation

class d3d.tracking.filter.**Pose_IMM**
Bases: *d3d.tracking.filter.PoseFilter*

UKF using IMM model for pose estimation

class d3d.tracking.filter.**PropertyFilter**
Bases: object

Abstraction class defining the interfaces for filters on target property

property classification: numpy.ndarray
Current classification estimation

property classification_var: numpy.ndarray
Covariance for current classification estimation

property dimension: numpy.ndarray
Current dimension estimation

property dimension_var: numpy.ndarray
Covariance for current shape estimation

predict(*dt*)
Predict the current system state

Parameters **dt** (*float*) – Time since last update

update(*target*)
Correct the current system state with observation. This method should always be called after calling *predict()*.

Parameters **target** (*Union[d3d.abstraction.ObjectTarget3D, d3d.abstraction.TrackingTarget3D]*) – New target observation

d3d.tracking.filter.**is_pd**(*B*)
Returns true when input is positive-definite, via Cholesky

d3d.tracking.filter.**motion_CSAA**(*state*, *dt*)
Constant Steering Angle and Acceleration model.

Parameters

- **state** (*numpy.ndarray*) – original state in format [x, y, theta, v, a, c] [0 1 2 3 4 5]
- **dt** (*float*) – time difference after last update

Returns updated state

Return type numpy.ndarray

d3d.tracking.filter.**motion_CTRA**(*state*, *dt*)
Constant Turn-Rate and (longitudinal) Acceleration model. This model also assume that velocity is the same with heading angle. CV, CTRV can be modeled by assume value equals zero

Parameters

- **state** (*Union[numpy.ndarray, List[float]]*) – original state in format [x, y, theta, v, a, w]
- **dt** (*float*) – time difference after last update

Returns updated state

Return type numpy.ndarray

d3d.tracking.filter.**motion_CV**(*state*, *dt*)
Constant Velocity model

Parameters

- **state** (*Union[numpy.ndarray, List[float]]*) – original state in format [x, y, vx, vy]
- **dt** (*float*) – time difference after last update

Returns updated state

Return type numpy.ndarray

d3d.tracking.filter.**nearest_pd**(*A*)
Find the nearest positive-definite matrix to input A Python/Numpy port of John D'Errico's *nearestSPD* MATLAB code [1], which credits [2].

- [1] https://www.mathworks.com/matlabcentral/fileexchange/42885-nearestspd
- **[2] N.J. Higham, "Computing a nearest symmetric positive semidefinite** matrix" (1988): https://doi.org/10.1016/0024-3795(88)90223-6

d3d.tracking.filter.**wrap_angle**(*theta*)
Normalize the angle to [-pi, pi)

Parameters **theta** (*float*) – angle to be wrapped

Returns wrapped angle

Return type float

CHAPTER

TWENTY

D3D.VIS

This module contains utilities for visualization.

20.1 d3d.vis.image

This module contains visualization methods on image

d3d.vis.image.**visualize_detections**(*ax*, *image_frame*, *targets*, *calib*, *box_color=(0, 1, 0)*, *thickness=2*, *tags=None*)

Draw detected object on matplotlib canvas

Parameters

- **ax** (*matplotlib.axes.Axes*) –
- **image_frame** (*str*) –
- **targets** (d3d.abstraction.Target3DArray) –
- **calib** (d3d.abstraction.TransformSet) –

20.2 d3d.vis.pcl

d3d.vis.pcl.**visualize_detections**(*visualizer*, *visualizer_frame*, *targets*, *calib*, *text_scale=0.8*, *box_color=(1, 1, 1)*, *text_color=(1, 0.8, 1)*, *id_prefix=''*, *tags=None*, *text_offset=None*, *viewport=0*)

Visualize detection targets in PCL Visualizer.

Parameters

- **visualizer** (*pcl.Visualizer*) – The pcl.Visualizer instance used for visualization
- **visualizer_frame** (*str*) – The frame that visualizer is in
- **targets** (d3d.abstraction.Target3DArray) – Object array to be visualized
- **calib** (d3d.abstraction.TransformSet) – TransformSet object storing calibration information. This is mandatory if the targets are in different frames
- **text_scale** – The scale for text tags. Set to 0 or negative if you want to suppress text visualization
- **box_color** – Specifying the color of bounding boxes. If it's a tuple, then it's assumed that it contains three RGB values in range 0-1. If it's a str or matplotlib colormap object, then the color comes from applying colormap to the object id.

- **text_color** – Specifying the color of text tags.
- **id_prefix** – Prefix of actor ids in PCL Visualizer, essential when this function is called multiple times
- **text_offset** – Relative position of text tags with regard to the box center
- **viewport** – Viewport for objects to be added. This is a PCL related feature

20.3 d3d.vis.xviz

class d3d.vis.xviz.**TrackingDatasetConverter**(*loader*, *lidar_names=None*, *camera_names=None*, *lidar_colormap='hot'*)

Bases: object

This class converts tracking dataset to data blobs like https://github.com/uber/xviz-data You can derive this class and custom the visualization results

Parameters

- **lidar_names** – Frame names of lidar to be visualized
- **camera_names** – Frame names of camera to be visualized
- **lidar_colormap** – Matplotlib colormap used to color lidar points
- **loader** (d3d.dataset.base.TrackingDatasetBase) –

d3d.vis.xviz.**visualize_detections**(*builder*, *visualizer_frame*, *targets*, *calib*, *stream_prefix*, *id_prefix=''*, *tags=None*, *text_offset=None*)

Add detection results to xviz builder

Parameters

- **builder** (*xviz_avs.builder.XVIZBuilder*) –
- **visualizer_frame** (*str*) –
- **targets** (d3d.abstraction.Target3DArray) –
- **calib** (d3d.abstraction.TransformSet) –
- **stream_prefix** (*str*) –

d3d.vis.xviz.**visualize_detections_metadata**(*builder*, *tag_enum*, *stream_prefix='/tracklets'*, *box_color=(1, 1, 1)*, *text_color=(1, 1, 1)*)

Parameters

- **tag_enum** (*enum.Enum*) – Enumeration of all possible tags.
- **box_color** – tuple or dict of tuple. Define bounding box color for each category
- **text_color** – tuple or dict of tuple. Define text color for each category
- **builder** (*xviz_avs.builder.XVIZMetadataBuilder*) –

CHAPTER

TWENTYONE

D3D.VOXEL

This module contains implementation of voxel related autograd oprations.

class d3d.voxel.**VoxelGenerator**(*bounds*, *shape*, *min_points=0*, *max_points=30*, *max_voxels=20000*, *max_points_filter=None*, *max_voxels_filter=None*, *reduction=None*, *dense=False*)

Bases: object

Convert point cloud to voxels

Parameters

- **shape** – The shape of the voxel grid
- **bouds** – The boundary of the voxel grid, in format [xmin, xmax, ymin, ymax, zmin, zmax]
- **min_points** – Minimum number of points per voxel
- **max_points** – Maximum number of points per voxel
- **max_voxels** – Maximum total number of voxels
- **reduction** – Type of reduction to apply, {none, mean, max, min}. Only for dense representation
- **dense** – Whether the output is in dense representation.
- **max_voxels_filter** – Filter to be applied to filter voxels
- **max_points_filter** – Filter to be applied to filter points in a voxel

CHAPTER

TWENTYTWO

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

d

INDEX

Symbols

A

B

C

D

E

K

L

M

N

O

P

Q

R

S

T

U

V