Torch Points 3D

Thomas Chaton and Nicolas Chaulet

Apr 30, 2021

CONTENTS:

Torch Points 3D is a framework for developing and testing common deep learning models to solve tasks related to unstructured 3D spatial data i.e. Point Clouds. The framework currently integrates some of the best published architectures and it integrates the most common public datasests for ease of reproducibility. It heavily relies on Pytorch Geometric and Facebook Hydra library thanks for the great work!

We aim to build a tool which can be used for benchmarking SOTA models, while also allowing practitioners to efficiently pursue research into point cloud analysis, with the end-goal of building models which can be applied to real-life applications.

You can easily install Torch Points3D with `pip`

```
pip install torch
pip install torch-points3d
```

but first make sure that the following dependencies are met

- CUDA 10 or higher (if you want GPU version)
- Python 3.6 or higher + headers (python-dev)
- PyTorch 1.7 or higher
- MinkowskiEngine (optional) see here for installation instructions

CHAPTER

ONE

CORE FEATURES

- **Task** driven implementation with dynamic model and dataset resolution from arguments.
- **Core** implementation of common components for point cloud deep learning - greatly simplifying the creation of new models:
  - **Core Architectures** - Unet
  - **Core Modules** - Residual Block, Down-sampling and Up-sampling convolutions
  - **Core Transforms** - Rotation, Scaling, Jitter
  - **Core Sampling** - FPS, Random Sampling, Grid Sampling
  - **Core Neighbour Finder** - Radius Search, KNN
- 4 **Base Convolution** base classes to simplify the implementation of new convolutions. Each base class supports a different data format (B = number of batches, C = number of features):
  - **DENSE** (B, num_points, C)
  - **PARTIAL DENSE** (B * num_points, C)
  - **MESSAGE PASSING** (B * num_points, C)
  - **SPARSE** (B * num_points, C)
- Models can be completely specified using a YAML file, greatly easing reproducability.
- Several visualiation tools **(tensorboard, wandb)** and **dynamic metric-based model checkpointing** , which is easily customizable.
- **Dynamic customized placeholder resolution** for smart model definition.

CHAPTER

TWO

SUPPORTED MODELS

The following models have been tested and validated:

- Relation-Shape Convolutional (RSConv) Neural Network for Point Cloud Analysis
- KPConv: Flexible and Deformable Convolution for Point Clouds
- PointCNN: Convolution On X-Transformed Points
- PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space
- 4D Spatio-Temporal ConvNets: Minkowski Convolutional Neural Networks
- Deep Hough Voting for 3D Object Detection in Point Clouds

We are actively working on adding the following ones to the framework:

- RandLA-Net: Efficient Semantic Segmentation of Large-Scale Point Clouds - implemented but not completely tested

and much more to come ...

CHAPTER

THREE

SUPPORTED TASKS

- Segmentation
- Registration
- Classification
- Object detection

3.1 Getting Started

You're reading this because the API wasn't cracking it and you would like to extend the framework for your own task or use some of the deeper layers of our codebase. This set of pages will take you from setting up the code for local development all the way to adding a new task or a new dataset to the framework. For using Torch Points3D as a library please refer to *this section*.

3.1.1 Installation

Install Python 3.6 or higher

Start by installing Python > 3.6. You can use pyenv by doing the following:

```
curl -L https://github.com/pyenv/pyenv-installer/raw/master/bin/pyenv-installer | bash
```

Add these three lines to your `.bashrc`

```
export PATH="$HOME/.pyenv/bin:$PATH"
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"
```

Finaly you can install `python 3.6.10` by running the following command

```
pyenv install 3.6.10
```

Install dependencies using poetry

Start by installing poetry:

```
pip install poetry
```

You can clone the repository and install all the required dependencies as follow:

```
git clone https://github.com/nicolas-chaulet/torch-points3d.git
cd torch-points3d
pyenv local 3.6.10
poetry install --no-root
```

You can check that the install has been successful by running

```
poetry shell
python -m unittest -v
```

Minkowski engine support

The repository is supporting Minkowski Engine which requires *openblas-dev* and *nvcc* if you have a CUDA device on your machine. First install *openblas*

```
sudo apt install libopenblas-dev
```

then make sure that *nvcc* is in your path:

```
nvcc -V
```

If it's not then locate it (*locate nvcc*) and add its location to your *PATH* variable. On my machine:

```
export PATH="/usr/local/cuda-10.2/bin:$PATH"
```

You are now in a position to install MinkowskiEngine with GPU support:

```
poetry install -E MinkowskiEngine --no-root
```

Installation within a virtual environment

We try to maintain a `requirements.txt` file for those who want to use plain old `pip`. Start by cloning the repo:

```
git clone https://github.com/nicolas-chaulet/torch-points3d.git
cd torch-points3d
```

We still recommend that you first create a virtual environment and activate it before installing the dependencies:

```
python3 -m virtualenv pcb
source pcb/bin/activate
```

Install all dependencies:

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

You should now be able to run the tests successfully:

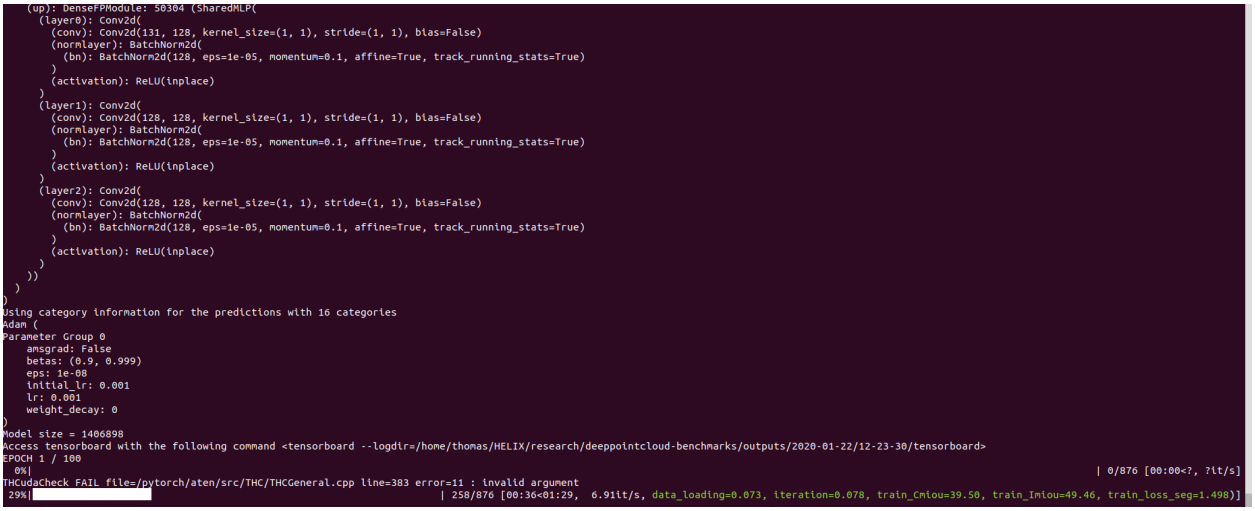
```
python -m unittest -v
```

3.1.2 Train!

You should now be in a position to train your first model. Here is how is goes to train pointnet++ on part segmentation task for dataset shapenet, simply run the following:

```
python train.py  \
    task=segmentation model_type=pointnet2 model_name=pointnet2_charlesssg␣
↪dataset=shapenet-fixed
```

And you should see something like that



The config for pointnet++ is a good example starting point to understand how models are defined:

```
                ]
            skip: True
        mlp_cls:
            nn: [128, 128]
            dropout: 0.5

    pointnet2_charlesmsg:
        class: pointnet2.PointNet2_D
        conv_type: "DENSE"
        use_category: ${data.use_category}
        down_conv:
            module_name: PointNetMSGDown
            npoint: [512, 128]
            radii: [[0.1, 0.2, 0.4], [0.4, 0.8]]
            nsamples: [[32, 64, 128], [64, 128]]
            down_conv_nn:
                [
                    [
                        [FEAT+3, 32, 32, 64],
                        [FEAT+3, 64, 64, 128],
                        [FEAT+3, 64, 96, 128],
                    ],
                    [
```

(continues on next page)

(continued from previous page)

```
                    [64 + 128 + 128+3, 128, 128, 256],
                    [64 + 128 + 128+3, 128, 196, 256],
                ],
            ]
    innermost:
        module_name: GlobalDenseBaseModule
        nn: [256 * 2 + 3, 256, 512, 1024]
    up_conv:
        module_name: DenseFPModule
        up_conv_nn:
            [
                [1024 + 256*2, 256, 256],
                [256 + 128 * 2 + 64, 256, 128],
```

Once the training is complete, you can access the model checkpoint as well as any visualisation and graphs that you may have generated in the `outputs/<date>/<time>` folder where date and time correspond to the time where you launched the training.

3.1.3 Visualise your results

We provide a notebook based on pyvista and panel that allows you to explore your past experiments visually. When using jupyter lab you will have to install an extension:

```
jupyter labextension install @pyviz/jupyterlab_pyviz
```

Once this is done you can launch jupyter lab from the root directory and run through the notebook. You should see a dashboard starting that looks like the following:

3.1.4 Project structure

The ambition of the project is to be a base for all point cloud related deep learning research. As such we wanted to make it scalable and also ensure that components could be reused. Below is the overall structure of the project:

```
├── benchmark                # Output from various benchmark runs
├── conf                     # All configurations for training nad evaluation leave␣
↪there
├── notebooks                # A collection of notebooks that allow result␣
↪exploration and network debugging
├── docker                   # Docker image that can be used for inference or␣
↪training
├── docs                     # All the doc
├── eval.py                  # Eval script
├── find_neighbour_dist.py   # Script that helps find the optimal number of␣
↪neighbours for neighbour search operations
├── forward_scripts          # Script that runs a forward pass on possibly non␣
↪annotated data
├── outputs                  # All outputs from your runs sorted by date
├── scripts                  # Some scripts to help manage the project
├── torch_points3d
    ├── core                 # Core components
    ├── datasets             # All code related to datasets
    ├── metrics              # All metrics and trackers
```

(continues on next page)

(continued from previous page)

```
    ├── models                  # All models
    ├── modules                 # Basic modules that can be used in a modular way
    ├── utils                   # Various utils
    └── visualization           # Visualization
├── test
└── train.py                    # Main script to launch a training
```

Note: As a general philosophy we have split datasets and models by task. For example, datasets has three subfolders:

- segmentation
- classification
- registration

where each folder contains the dataset related to each task.

3.2 Tutorials

Here you will learn how you can extend the framework to serve your needs, we will cover

- *Create a new dataset*
- *Create a new model*
- *Launch an experiment*
- *Train and Test on tasks already implemented*

3.2.1 Create a new dataset

Let's add support for the version of S3DIS that **Pytorch Geometric** provides: https://pytorch-geometric.readthedocs.io/en/latest/modules/datasets.html#torch_geometric.datasets.S3DIS

We are going to go through the successive steps to do so:

- *Create a dataset that the framework recognises*
- *Create a new configuration file*

Let's go through those steps together and in order to go further we highly recommend that you take a look at Before starting, we strongly advice to read the Creating Your Own Datasets from **Pytorch Geometric**.

Create a dataset that the framework recognises

The framework provides a base class for datasets that needs to be sub classed when you add your own. We also follow the convention that the `.py` file that describes a dataset for segmentation will leave in the `torch_points3d/datasets/segmentation` folder. For another task such as classification it would go in `torch_points3d/datasets/classification`.

Start by creating a new file `torch_points3d/datasets/segmentation/s3dis.py` with the class `S3DISDataset`, it should inherit from `BaseDataset`.

```
from torch_geometric.datasets import S3DIS

from torch_points3d.datasets.base_dataset import BaseDataset
from torch_points3d.metrics.segmentation_tracker import SegmentationTracker

class S3DISDataset(BaseDataset):
    def __init__(self, dataset_opt):
        super().__init__(dataset_opt)

        self.train_dataset = S3DIS(
            self._data_path,
            test_area=self.dataset_opt.fold,
            train=True,
            pre_transform=self.pre_transform,
            transform=self.train_transform,
        )
        self.test_dataset = S3DIS(
            self._data_path,
            test_area=self.dataset_opt.fold,
            train=False,
            pre_transform=self.pre_transform,
            transform=self.test_transform,
        )


    def get_tracker(self, wandb_log: bool, tensorboard_log: bool):
        """Factory method for the tracker

        Arguments:
            wandb_log - Log using weight and biases
            tensorboard_log - Log using tensorboard
        Returns:
            [BaseTracker] -- tracker
        """
        return SegmentationTracker(self, wandb_log=wandb_log, use_
↪tensorboard=tensorboard_log)
```

Let's explain the code more in details there.

```
class S3DISDataset(BaseDataset):
     def __init__(self, dataset_opt):
         super().__init__(dataset_opt)
```

This instantiates the parent class based on a given configuration `dataset_opt` (see *Create a new configuration file*) and this does few things for you:

- Sets the path to the data, by convention it will be `dataset_opt.dataroot/s3dis/` in our case (name of the class without Dataset)

- Extracts from the configuration the transforms that should be applied to you data before giving it to the model

Next comes the instantiation of the actual datasets that will be used for training and testing.

```
self.train_dataset = S3DIS(
    self._data_path,
    test_area=self.dataset_opt.fold,
    train=True,
    pre_transform=self.pre_transform,
    transform=self.train_transform,
)
self.test_dataset = S3DIS(
    self._data_path,
    test_area=self.dataset_opt.fold,
    train=False,
    pre_transform=self.pre_transform,
    transform=self.test_transform,
)
```

You can see that we use the `pre_transform`, `test_transform` and `train_transform` from the base class, they have been set based on the configuration that you have provided. The base class will then use those datasets to create the dataloaders that will be used in the training script.

The final step is to associate a metric tracker to your dataset, in this case we will use a SegmentationTracker that tracks IoU metrics as well as accuracy, mean accuracy and loss.

```
def get_tracker(self, wandb_log: bool, tensorboard_log: bool):
    """Factory method for the tracker

    Arguments:
        wandb_log - Log using weight and biases
        tensorboard_log - Log using tensorboard
    Returns:
        [BaseTracker] -- tracker
    """
    return SegmentationTracker(self, wandb_log=wandb_log, use_tensorboard=tensorboard_
↪log)
```

Create a new configuration file

Let's move to the next step, the definition of the configuration file that will control the behaviour of our dataset. The configuration file mainly controls the following things:

- Location of the data
- Transforms that will be applied to the data
- Python class that will be used for creating the actual python object used during training.

Let's create a `conf/data/segmentation/s3disfused.yaml` file with our own setting to setup the dataset

```
data:
  task: segmentation
  class: s3dis.S3DISFusedDataset
  dataroot: data
  fold: 5
  first_subsampling: 0.04
  use_category: False
```

(continues on next page)

(continued from previous page)

```
  pre_collate_transform:
      - transform: PointCloudFusion   # One point cloud per area
      - transform: SaveOriginalPosId    # Required so that one can recover the␣
↪original point in the fused point cloud
      - transform: GridSampling3D       # Samples on a grid
        params:
            size: ${data.first_subsampling}
  train_transforms:
    - transform: RandomNoise
      params:
        sigma: 0.001
    - transform: RandomRotate
      params:
        degrees: 180
        axis: 2
    - transform: RandomScaleAnisotropic
      params:
        scales: [0.8, 1.2]
    - transform: RandomSymmetry
      params:
        axis: [True, False, False]
    - transform: DropFeature
      params:
        drop_proba: 0.2
        feature_name: rgb
    - transform: XYZFeature
      params:
        add_x: False
        add_y: False
        add_z: True
    - transform: AddFeatsByKeys
      params:
        list_add_to_x: [True, True]
        feat_names: [rgb, pos_z]
        delete_feats: [True, True]
    - transform: Center
  test_transform:
    - transform: XYZFeature
      params:
        add_x: False
        add_y: False
        add_z: True
    - transform: AddFeatsByKeys
      params:
        list_add_to_x: [True, True]
        feat_names: [rgb, pos_z]
        delete_feats: [True, True]
    - transform: Center
  val_transform: ${data.test_transform}
```

Note:

- `task` needs to be specified. Currently, the arguments provided by the command line are lost and therefore we need the extra information.
- `class` needs to be specified. In that case, since we solve a classification task, the code will look for a class named `S3DISDataset` within the `torch_points3d/datasets/segmentation/s3dis.py` file.

For more details about the tracker please refer to the source code

3.2.2 Create a new model

Let's add PointNet++ model implemented within the "DENSE" format type to the project. Model definitions are separated between the definition of the core "convolution" operation equivalent to a Conv2D on images (see *Create the basic modules*) and the overall model that combines all those convolutions (see *Assemble all the basic blocks*).

We are going to go through the successive steps to do so:

Create the basic modules

Let's create `torch_points3d/modules/pointnet2/` directory and `dense.py` file within.

Note: Remember to create a `__init__.py` file within that directory that will contain the multiscale convolution proposed in pointnet++.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch_points_kernels as tp

from torch_points3d.core.base_conv.dense import *
from torch_points3d.core.spatial_ops import DenseRadiusNeighbourFinder,␣
↪DenseFPSSampler
from torch_points3d.utils.model_building_utils.activation_resolver import get_
↪activation


class PointNetMSGDown(BaseDenseConvolutionDown):
    def __init__(
        self,
        npoint=None,
        radii=None,
        nsample=None,
        down_conv_nn=None,
        bn=True,
        activation=torch.nn.LeakyReLU(negative_slope=0.01),
        use_xyz=True,
        normalize_xyz=False,
        **kwargs
    ):
        assert len(radii) == len(nsample) == len(down_conv_nn)
```

(continues on next page)

(continued from previous page)

```
        super(PointNetMSGDown, self).__init__(
            DenseFPSSampler(num_to_sample=npoint), DenseRadiusNeighbourFinder(radii,␣
↪nsample), **kwargs
        )
        self.use_xyz = use_xyz
        self.npoint = npoint
        self.mlps = nn.ModuleList()
        for i in range(len(radii)):
            self.mlps.append(MLP2D(down_conv_nn[i], bn=bn, activation=activation,␣
↪bias=False))
        self.radii = radii
        self.normalize_xyz = normalize_xyz

    def _prepare_features(self, x, pos, new_pos, idx, scale_idx):
        new_pos_trans = pos.transpose(1, 2).contiguous()
        grouped_pos = tp.grouping_operation(new_pos_trans, idx)  # (B, 3, npoint,␣
↪nsample)
        grouped_pos -= new_pos.transpose(1, 2).unsqueeze(-1)

        if self.normalize_xyz:
            grouped_pos /= self.radii[scale_idx]

        if x is not None:
            grouped_features = tp.grouping_operation(x, idx)
            if self.use_xyz:
                new_features = torch.cat([grouped_pos, grouped_features], dim=1)  #␣
↪(B, C + 3, npoint, nsample)
            else:
                new_features = grouped_features
        else:
            assert self.use_xyz, "Cannot have not features and not use xyz as a␣
↪feature!"
            new_features = grouped_pos

        return new_features

    def conv(self, x, pos, new_pos, radius_idx, scale_idx):
        """ Implements a Dense convolution where radius_idx represents
        the indexes of the points in x and pos to be agragated into the new feature
        for each point in new_pos

        Arguments:
            x -- Previous features [B, N, C]
            pos -- Previous positions [B, N, 3]
            new_pos  -- Sampled positions [B, npoints, 3]
            radius_idx -- Indexes to group [B, npoints, nsample]
            scale_idx -- Scale index in multiscale convolutional layers
        Returns:
            new_x -- Features after passing trhough the MLP [B, mlp[-1], npoints]
        """
        assert scale_idx < len(self.mlps)
        new_features = self._prepare_features(x, pos, new_pos, radius_idx, scale_idx)
        new_features = self.mlps[scale_idx](new_features)  # (B, mlp[-1], npoint,␣
↪nsample)
        new_features = F.max_pool2d(new_features, kernel_size=[1, new_features.
↪size(3)])  # (B, mlp[-1], npoint, 1)
        new_features = new_features.squeeze(-1)  # (B, mlp[-1], npoint)
```

(continues on next page)

(continued from previous page)

```
        return new_features
```

Let's dig in.

```
class PointNetMSGDown(BaseDenseConvolutionDown):
    def __init__(
        ...
    ):
        super(PointNetMSGDown, self).__init__(
            DenseFPSSampler(num_to_sample=npoint), DenseRadiusNeighbourFinder(radii,␣
↪nsample), **kwargs
        )
```

- The `PointNetMSGDown` inherit from `BaseDenseConvolutionDown`:
  - `BaseDenseConvolutionDown` takes care of all the sampling and search logic for you.
  - Therefore, a `sampler` and a `neighbour finder` have to be provided.
  - Here, we provide `DenseFPSSampler` (furthest point sampling) and `DenseRadiusNeighbourFinder` (neighbour search within a given radius)
- The `PointNetMSGDown` class just needs to implement the `conv` method which implements the actual logic for deriving the features that will come out of this layer. Here the features of a given point are obtained by passing the neighbours of that point through an MLP.

Assemble all the basic blocks

Let's create a new file `/torch_points3d/models/segmentation/pointnet2.py` with its associated class `PointNet2_D`

```
import torch

import torch.nn.functional as F
from torch_geometric.data import Data
import logging

from torch_points3d.modules.pointnet2 import * # This part is extremely important.␣
↪Always important the associated modules within your this file
from torch_points3d.core.base_conv.dense import DenseFPModule
from torch_points3d.models.base_architectures import UnetBasedModel

log = logging.getLogger(__name__)

class PointNet2_D(UnetBasedModel):
    def __init__(self, option, model_type, dataset, modules):
        """Initialize this model class.
        Parameters:
            opt -- training/test options
        A few things can be done here.
        - (required) call the initialization function of BaseModel
        - define loss function, visualization images, model names, and optimizers
        """
        UnetBasedModel.__init__(
            self, option, model_type, dataset, modules
        )  # call the initialization method of UnetBasedModel
```

(continues on next page)

(continued from previous page)

```
        # Create the mlp to classify data
        nn = option.mlp_cls.nn
        self.dropout = option.mlp_cls.get("dropout")
        self.lin1 = torch.nn.Linear(nn[0], nn[1])
        self.lin2 = torch.nn.Linear(nn[2], nn[3])
        self.lin3 = torch.nn.Linear(nn[4], dataset.num_classes)

        self.loss_names = ["loss_seg"] # This will be used the automatically get loss_
↪seg from self

    def set_input(self, data, device):
        """Unpack input data from the dataloader and perform necessary pre-processing␣
↪steps.
        Parameters:
            input: a dictionary that contains the data itself and its metadata␣
↪information.
        """
        data = data.to(device)
        self.input = data
        self.labels = data.y
        self.batch_idx = torch.arange(0, data.pos.shape[0]).view(-1, 1).repeat(1,␣
↪data.pos.shape[1]).view(-1)

    def forward(self) -> Any:
        """Run forward pass. This will be called by both functions <optimize_
↪parameters> and <test>."""
        data = self.model(self.input)
        x = F.relu(self.lin1(data.x))
        x = F.dropout(x, p=self.dropout, training=self.training)
        x = self.lin2(x)
        x = F.dropout(x, p=self.dropout, training=self.training)
        x = self.lin3(x)
        self.output = F.log_softmax(x, dim=-1)
        return self.output

    def backward(self):
        """Calculate losses, gradients, and update network weights; called in every␣
↪training iteration"""
        # caculate the intermediate results if necessary; here self.output has been␣
↪computed during function <forward>
        # calculate loss given the input and intermediate results
        self.loss_seg = F.nll_loss(self.output, self.labels) + self.get_internal_
↪loss()

        self.loss_seg.backward()  # calculate gradients of network G w.r.t. loss_G
```

Note:

- Make sure that you import all the required modules
- You need to inherit from `BaseModel`. That class contains all the core logic that enables training (see base_model.py for more details)

Create a new configuration

We create a new file `conf/models/segmentation/pointnet2.yaml`. This file will contain all the **different versions** of pointnet++.

```
models:
                ]
            skip: True
        mlp_cls:
            nn: [128, 128]
            dropout: 0.5

    pointnet2_charlesmsg:
        class: pointnet2.PointNet2_D
        conv_type: "DENSE"
        use_category: ${data.use_category}
        down_conv:
            module_name: PointNetMSGDown
            npoint: [512, 128]
            radii: [[0.1, 0.2, 0.4], [0.4, 0.8]]
            nsamples: [[32, 64, 128], [64, 128]]
            down_conv_nn:
                [
                    [
                        [FEAT+3, 32, 32, 64],
                        [FEAT+3, 64, 64, 128],
                        [FEAT+3, 64, 96, 128],
                    ],
                    [
                        [64 + 128 + 128+3, 128, 128, 256],
                        [64 + 128 + 128+3, 128, 196, 256],
                    ],
                ]
        innermost:
            module_name: GlobalDenseBaseModule
            nn: [256 * 2 + 3, 256, 512, 1024]
        up_conv:
            module_name: DenseFPModule
            up_conv_nn:
                [
                    [1024 + 256*2, 256, 256],
                    [256 + 128 * 2 + 64, 256, 128],
```

Here is `PointNet++` Multi-Scale original version by Charles R. Qi.

Let's dig in the definition.

Required arguments

- `pointnet2_charlesmsg` is model_name and should be provided from the command line in order to load this file configuration.
- `architecture: pointnet2.PointNet2_D`. It indicates where to find the Model Logic. The framework backend will look for the file `/torch_points3d/models/segmentation/pointnet2.py` and the `PointNet2_D` class.
- `conv_type: "DENSE"`

"Optional" arguments

When I say optional, I mean those parameters could be defined differently for your own model. We don't want to force any particular configuration format however, the simpler is always better !

The format above is used across models that leverage our Unet architecture builder base class torch_points3d/models/base_architectures/unet.py with `UnetBasedModel` and `UnwrappedUnetBasedModel`. The following arguments are required by those classes:

- `down_conv`: parameters of each down convolution layer
- `innermost`: parameters of the innermost layer
- `up_conv`: parameters of each up convolution layer

Those elements need to contain a `module_name` which will be used to create the associated Module.

Those Unet builder classes will do the followings:

- If provided a list, it will use the index to access the value
- If provided something else, it will broadcast the arguments to all convolutions.

Understanding the model

From the configuration written above, we can infer that

- The model has got two down convolutions, one inner module and three up convolutions
- Each down convolutions is a multiscale pointnet++ convolution implemented with the class `PointNetMSGDown`
- The first down convolution uses the following parameters:
  - only 512 points are kept after this layer,
  - three scales with radii 0.1, 0.2 and 0.4 are used,
  - 32, 64 and 128 neighbours are kept for each scale
  - the multi layer perceptrons for each scale are of size: [FEAT+3, 32, 32, 64], [FEAT+3, 64, 64, 128] and [FEAT+3, 64, 96, 128] respectively
- The up convolution uses `DenseFPModule` and the first layer has got an MLP of size [1024 + 256*2, 256, 256]
- The final classifier has got two layers and uses a dropout of 0.5

Another example with RSConcv

Here is an example with the `RSConv` implementation in `MESSAGE_TYPE ConvType`.

```
class RSConvDown(BaseConvolutionDown):
    def __init__(self, ratio=None, radius=None, local_nn=None, down_conv_nn=None,␣
↪*args, **kwargs):
        super(RSConvDown, self).__init__(FPSSampler(ratio),␣
↪RadiusNeighbourFinder(radius), *args, **kwargs)

        self._conv = Convolution(local_nn=local_nn, global_nn=down_conv_nn)

    def conv(self, x, pos, edge_index, batch):
        return self._conv(x, pos, edge_index)
```

We can see this convolution needs the followings arguments

```
ratio=None, radius=None, local_nn=None, down_conv_nn=None
```

Here is an extract from the model architecture config:

```
down_conv: # For the encoder part convolution
    module_name: RSConvDown # We will be using the RSConvDown Module

    # And provide to each convolution, the associated arguments within a list are␣
↪selected using the convolution index.
    # For the others, there are just copied for each convolution.
    activation:
        name:  "LeakyReLU"
        negative_slope: 0.2
    ratios: [0.2, 0.25]
    radius: [0.1, 0.2]
    local_nn: [[10, 8, FEAT], [10, 32, 64, 64]]
    down_conv_nn: [[FEAT, 16, 32, 64], [64, 64, 128]]
```

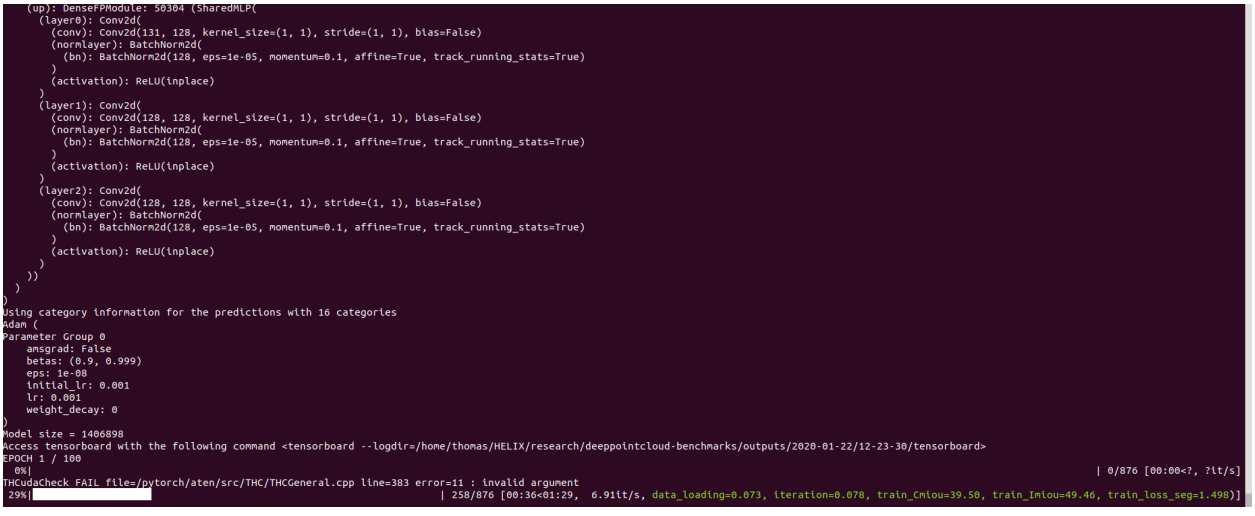
- First convolution receives `ratio=0.2, radius=0.1, local_nn_=[10, 8, 3], down_conv_nn=[3, 16, 32, 64]`
- Second convolution receives `ratio=0.25, radius=0.2, local_nn_=[10, 32, 64, 64], down_conv_nn=[64, 64, 128]`
- Both of them will also receive a dictionary `activation = {name: "LeakyReLU", negative_slope: 0.2}`

3.2.3 Launch an experiment

Now that we have our new dataset and model, it is time to launch a training. If you have followed the instructions above you should be able to simply run the following command and should run smoothly!

```
poetry run python run task=segmentation dataset=s3dis model_type=pointnet2 model_
↪name=pointnet2_charlesmsg
```

Your terminal should contain:



3.2.4 Train and Test on tasks already implemented

In this section, We will see How we can train and test model on existing datasets.

Registration Task

In registration task, the goal is to find the right transformation that align correctly pattern. Here, we will show how we can use deep learning to solve this task. Especially, we will see how we can use Fully Convolutional Geometric Feature. FCGF use a Unet architecture to compute feature per point and then we can match these features. Then to find the correct transformation, we can use algorithms such as RANSAC or Fast Global Registration. For this task, we use siamese networks, it means that the dataset provides pairs of point clouds and the networks is applied to both pairs.

To train on 3DMatch, we can type the command:

```
poetry run python train.py task=registration model_type=minkowski model_name=MinkUNet_
↪Fragment dataset=fragment3dmatch_sparse training=sparse_fragment_reg
```

the config file for models are in the `conf/models/registration/`. It automatically instantiates models written in `torch_points3d/models/registration`. The config file for the datasets are here `conf/data/registration`. preprocessing and data augmentation are defined here. So here, it will train a network with the sparse convolution from Minkowski engine, with the architecture specified in the following path on 3DMatch.

We can try an other convolution (for example KPConv):

```
poetry run train.py task=registration model_type=kpconv model_name=KPFCNN␣
↪dataset=fragment3dmatch_partial training=sparse_fragment_reg``
```

In the case of KPConv, because it's not the same convolution, the pre-processing is different. 3DMatch is a dataset containing RGBD frames and the poses from 5 different datasets.But our method need to be trained on 3D point cloud. So we need to fuse RGBD frame to create fragments. Our code will download automatically the RGBD frames with the poses. To build the fragment, we mainly rely on the code from this repository: In the yaml code, we specify the params to build the fragments for the training and the evaluation and also we provide the parameters for the evaluation.

If you want to test your model you can use the provided scripts.

```
python scripts/test_registration_scripts/evaluate.py task=registration model_
↪type=minkowski model_name=MinkUNet_Fragment dataset=fragment3dmatch_sparse training.
↪checkpoint_dir="your weights " data.sym=True
```

Where you have to replace "Your weights" by the directory containing the weights. Finally, if you want to use the networks off the shelf on your own project (using the `PretrainedRegistry`). you can check the notebooks `notebooks/demo_registration_3dm.ipynb` and `demo_registration_kitti.ipynb` for 3DMatch and KITTI.

3.3 Advanced

3.3.1 Configuration

Overview

We have chosen Facebook Hydra library as out core tool for managing the configuration of our experiments. It provides a nice and scalable interface to defining models and datasets. We encourage our users to take a look at their documentation and get a basic understanding of its core functionalities. As per their website

"Hydra is a framework for elegantly configuring complex applications"

Configuration architecture

All configurations leave in the conf folder and it is organised as follow:

```
.
├── config.yaml      # main config file for training
├── data             # contains all configurations related to datasets
├── debugging        # configs that can be used for debugging purposes
├── eval.yaml        # Main config for running a full evaluation on a given dataset
├── hydra            # hydra specific configs
├── lr_scheduler     # learning rate schedulers
├── models           # Architectures of the models
├── sota.yaml        # SOTA scores
├── training         # Training specific parameters
└── visualization    # Parameters for saving visualisation artefact
```

Understanding config.yaml

`config.yaml` is the config file that governs the behaviour of your trainings. It gathers multiple configurations into one, and it is organised as follow:

```yaml
defaults:
  - task: ??? # Task performed (segmentation, classification etc...)
    optional: True
  - model_type: ??? # Type of model to use, e.g. pointnet2, rsconv etc...
    optional: True
  - dataset: ???
    optional: True

  - visualization: default
  - lr_scheduler: multi_step
  - training: default
  - eval

  - debugging: default.yaml
  - models: ${defaults.0.task}/${defaults.1.model_type}
  - data: ${defaults.0.task}/${defaults.2.dataset}
  - sota # Contains current SOTA results on different datasets (extracted from papers␣
↪!).
  - hydra/job_logging: custom

model_name: ??? # Name of the specific model to load

selection_stage: ""
pretty_print: False
```

Hydra is expecting the followings arguments from the command line:

- task
- model_type
- dataset
- model_name

The provided `task` and `dataset` will be used to load the configuration for the dataset at `conf/data/{task}/{dataset}.yaml` while the `model_type` argument will be used to load the model config at `conf/models/{task}/{model_type}.yaml`. Finally `model_name` is used to pull the appropriate model from the model configuration file.

Training arguments

```yaml
# Those arguments defines the training hyper-parameters
training:
    epochs: 100
    num_workers: 6
    batch_size: 16
    shuffle: True
    cuda: 0 # -1 -> no cuda otherwise takes the specified index
    precompute_multi_scale: False # Compute multiscate features on cpu for faster training / inference
    optim:
        base_lr: 0.001
        # accumulated_gradient: -1 # Accumulate gradient accumulated_gradient * batch_size
        grad_clip: -1
        optimizer:
            class: Adam
            params:
                lr: ${training.optim.base_lr} # The path is cut from training
        lr_scheduler: ${lr_scheduler}
        bn_scheduler:
            bn_policy: "step_decay"
            params:
                bn_momentum: 0.1
                bn_decay: 0.9
                decay_step : 10
                bn_clip : 1e-2
    weight_name: "latest" # Used during resume, select with model to load from [miou, macc, acc..., latest]
    enable_cudnn: True
    checkpoint_dir: ""

# Those arguments within experiment defines which model, dataset and task to be created for benchmarking
# parameters for Weights and Biases
wandb:
    entity: ""
    project: default
    log: True
    notes:
    name:
    public: True # It will be display the model within wandb log, else not.
    config:
        model_name: ${model_name}

    # parameters for TensorBoard Visualization
tensorboard:
    log: True
```

- `precompute_multi_scale`: Computes spatial queries such as grid sampling and neighbour search on cpu for faster. Currently this is only supported for KPConv.

Eval arguments

```
num_workers: 1
batch_size: 1
cuda: 6
weight_name: "latest" # Used during resume, select with model to load from [miou,␣
↪macc, acc..., latest]
enable_cudnn: True
checkpoint_dir: "/home/ChauletN/torch-points3d/outputs/2020-09-22/17-06-39" # "{your_
↪path}/outputs/2020-01-28/11-04-13" for example
model_name: Res16UNet34C
precompute_multi_scale: True # Compute multiscate features on cpu for faster training␣
↪/ inference
enable_dropout: False
voting_runs: 1

tracker_options: # Extra options for the tracker
  full_res: False
  make_submission: False

hydra:
  run:
    dir: ${checkpoint_dir}/eval/${now:%Y-%m-%d_%H-%M-%S}
```

3.3.2 Data formats for point cloud

While developing this project, we discovered there are several ways to implement a convolution.

- “DENSE”
- “PARTIAL_DENSE”
- “MESSAGE_PASSING”
- “SPARSE”

Dense

This format is very similar to what you would be used to with images, during the assembling of a batch the B tensors of shape (num_points, feat_dim) will be concatenated on a new dimension [(num_points, feat_dim), . . . , (num_points, feat_dim)] -> (B, num_points, feat_dim).

This format forces each sample to have exactly the same number of points.

Advantages

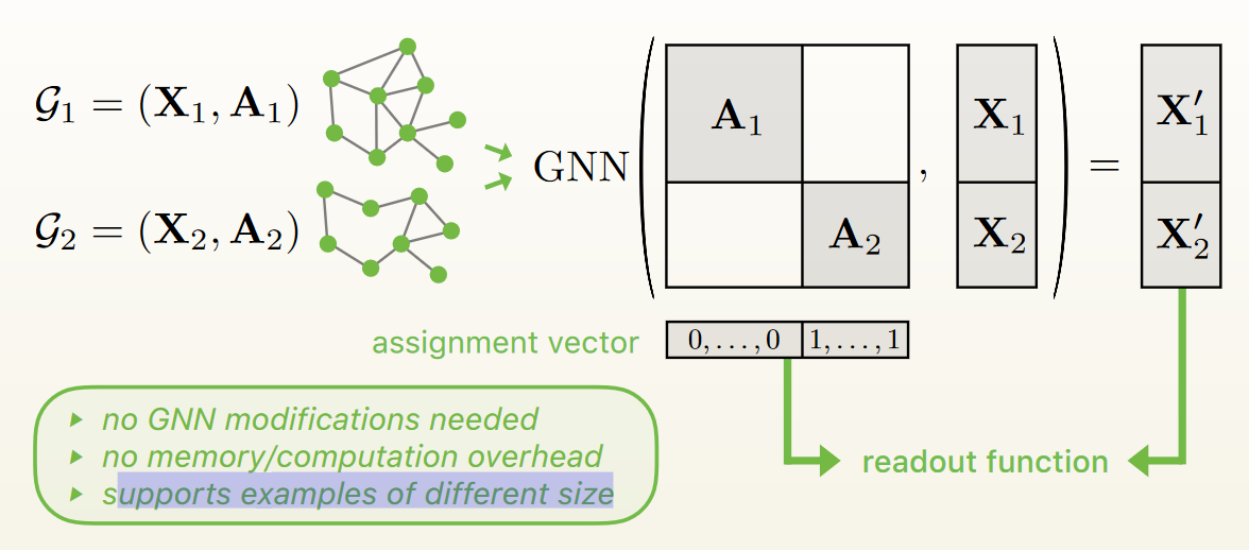
- The format is dense and therefore aggregation operation are fast

Drawbacks

- Handling variability in the number of neighbours happens through padding which is not very efficient
- Each sample needs to have the same number of points, as a consequence points are duplicated or removed from a sample during the data loading phase using a FixedPoints transform

Sparse formats

The second family of convolution format is based on a sparse data format meaning that each sample can have a variable number of points and the collate function handles the complexity behind the scene. For those intersted in learning more about it Batch.from_data_list



Given `N` tensors with their own `num_points_{i}`, the collate function does:

```
[(num_points_1, feat_dim), ..., (num_points_n, feat_dim)]
    -> (num_points_1 + ... + num_points_n, feat_dim)
```

It also creates an associated `batch tensor` of size `(num_points_1 + ... + num_points_n)` with indices of the corresponding batch.

Note: Example

- A with shape (2, 2)
- B with shape (3, 2)

`C = Batch.from_data_list([A, B])`

C is a tensor of shape `(5, 2)` and its associated batch will contain `[0, 0, 1, 1, 1]`

PARTIAL_DENSE ConvType format

This format is used by KPConv original implementation.

Same as dense format, it forces each point to have the same number of neighbors. It is why we called it partially dense.

MESSAGE_PASSING ConvType Format

This ConvType is Pytorch Geometric base format. Using Message Passing API class, it deploys the graph created by `neighbour finder` using internally the `torch.index_select` operator.

Therefore, the `[PointNet++]` internal convolution looks like that.

```
import torch
from torch_geometric.nn.conv import MessagePassing
from torch_geometric.utils import remove_self_loops, add_self_loops

from ..inits import reset

class PointConv(MessagePassing):
    r"""The PointNet set layer from the `"PointNet: Deep Learning on Point Sets
    for 3D Classification and Segmentation"
    <https://arxiv.org/abs/1612.00593>`_ and `"PointNet++: Deep Hierarchical
    Feature Learning on Point Sets in a Metric Space"
    <https://arxiv.org/abs/1706.02413>`_ papers
    """

    def __init__(self, local_nn=None, global_nn=None, **kwargs):
        super(PointConv, self).__init__(aggr='max', **kwargs)

        self.local_nn = local_nn
        self.global_nn = global_nn

        self.reset_parameters()

    def reset_parameters(self):
        reset(self.local_nn)
        reset(self.global_nn)


    def forward(self, x, pos, edge_index):
        r"""
        Args:
            x (Tensor): The node feature matrix. Allowed to be :obj:`None`.
            pos (Tensor or tuple): The node position matrix. Either given as
                tensor for use in general message passing or as tuple for use
                in message passing in bipartite graphs.
            edge_index (LongTensor): The edge indices.
        """
        if torch.is_tensor(pos):  # Add self-loops for symmetric adjacencies.
            edge_index, _ = remove_self_loops(edge_index)
            edge_index, _ = add_self_loops(edge_index, num_nodes=pos.size(0))

        return self.propagate(edge_index, x=x, pos=pos)


    def message(self, x_j, pos_i, pos_j):
        msg = pos_j - pos_i
        if x_j is not None:
            msg = torch.cat([x_j, msg], dim=1)
        if self.local_nn is not None:
            msg = self.local_nn(msg)
        return msg
```

(continues on next page)

(continued from previous page)

```
    def update(self, aggr_out):
        if self.global_nn is not None:
            aggr_out = self.global_nn(aggr_out)
        return aggr_out

    def __repr__(self):
        return '{}(local_nn={}, global_nn={})'.format(
            self.__class__.__name__, self.local_nn, self.global_nn)
```

SPARSE ConvType Format

The sparse conv type is used by project like SparseConv or Minkowski Engine, therefore, the points have to be converted into indices living within a grid.

3.3.3 Backbone Architectures

Several unet could be built using different convolution or blocks. However, the final model will still be a UNet.

In the `base_architectures` folder, we intend to provide base architecture builder which could be used across tasks and datasets.

We provide two UNet implementations:

- UnetBasedModel
- UnwrappedUnetBasedModel

The main difference between them if `UnetBasedModel` implements the forward function and `UnwrappedUnetBasedModel` doesn't.

UnetBasedModel

```
def forward(self, data):
    if self.innermost:
        data_out = self.inner(data)
        data = (data_out, data)
        return self.up(data)
    else:
        data_out = self.down(data)
        data_out2 = self.submodule(data_out)
        data = (data_out2, data)
        return self.up(data)
```

The UNet will be built recursively from the middle using the `UnetSkipConnectionBlock` class.

UnetSkipConnectionBlock .. code-block:

```
Defines the Unet submodule with skip connection.
X -------------------identity----------------------
-- downsampling -- |submodule| -- upsampling --|
```

UnwrappedUnetBasedModel

The `UnwrappedUnetBasedModel` will create the model based on the configuration and add the created layers within the followings `ModuleList`

```
self.down_modules = nn.ModuleList()
self.inner_modules = nn.ModuleList()
self.up_modules = nn.ModuleList()
```

3.3.4 Datasets

Segmentation

Preprocessed S3DIS

We support a couple of flavours or S3DIS. The dataset used for `S3DIS1x1` is coming from https://pytorch-geometric.readthedocs.io/en/latest/_modules/torch_geometric/datasets/s3dis.html.

It is a preprocessed version of the original data where each sample is a 1mx1m extraction of the original data. It was initially used in PointNet.

Raw S3DIS

The dataset used for S3DIS is the original dataset without any pre-processing applied. Here is the area_1 if you want to visualize it. We provide some data transform for combining each area back together and split the dataset into digestible chunks. Please refer to code base and associated configuration file for more details:

```
data:
  task: segmentation
  class: s3dis.S3DISFusedDataset
  dataroot: data
  fold: 5
  first_subsampling: 0.04
  use_category: False
  pre_collate_transform:
      - transform: PointCloudFusion   # One point cloud per area
      - transform: SaveOriginalPosId    # Required so that one can recover the␣
↪original point in the fused point cloud
      - transform: GridSampling3D       # Samples on a grid
        params:
            size: ${data.first_subsampling}
  train_transforms:
    - transform: RandomNoise
      params:
        sigma: 0.001
    - transform: RandomRotate
      params:
        degrees: 180
        axis: 2
    - transform: RandomScaleAnisotropic
      params:
        scales: [0.8, 1.2]
    - transform: RandomSymmetry
      params:
```

(continues on next page)

(continued from previous page)

```
        axis: [True, False, False]
    - transform: DropFeature
      params:
        drop_proba: 0.2
        feature_name: rgb
    - transform: XYZFeature
      params:
        add_x: False
        add_y: False
        add_z: True
    - transform: AddFeatsByKeys
      params:
        list_add_to_x: [True, True]
        feat_names: [rgb, pos_z]
        delete_feats: [True, True]
    - transform: Center
  test_transform:
    - transform: XYZFeature
      params:
        add_x: False
        add_y: False
        add_z: True
    - transform: AddFeatsByKeys
      params:
        list_add_to_x: [True, True]
        feat_names: [rgb, pos_z]
        delete_feats: [True, True]
    - transform: Center
  val_transform: ${data.test_transform}
```

Shapenet

Shapenet is a simple dataset that allows quick prototyping for segmentation models. When used in single class mode, for part segmentation on airplanes for example, it is a good way to figure out if your implementation is correct.

Classification

ModelNet

The dataset used for `ModelNet` comes in two format:

- ModelNet10
- ModelNet40 Their website is here https://modelnet.cs.princeton.edu/.

Registration

3D Match

http://3dmatch.cs.princeton.edu/

IRALab Benchmark

https://arxiv.org/abs/2003.12841 composed of data from:

- the ETH datasets (https://projects.asl.ethz.ch/datasets/doku.php?id=laserregistration:laserregistration)
- the Canadian Planetary Emulation Terrain 3D Mapping datasets (http://asrl.utias.utoronto.ca/datasets/3dmap/index.html)
- the TUM Vision Groud RGBD datasets (https://vision.in.tum.de/data/datasets/rgbd-dataset)
- the KAIST Urban datasets (https://irap.kaist.ac.kr/dataset)

3.3.5 Model checkpoint

Model Saving

Our custom `Checkpoint` class keeps track of the models for `every metric`, the stats for `"train", "test", "val"`, `optimizer` and `its learning params`.

```
self._objects = {}
self._objects["models"] = {}
self._objects["stats"] = {"train": [], "test": [], "val": []}
self._objects["optimizer"] = None
self._objects["lr_params"] = None
```

Model Loading

In training.yaml and eval.yaml, you can find the followings parameters:

- weight_name
- checkpoint_dir
- resume

As the model is saved for every metric + the latest epoch. It is possible by loading any of them using `weight_name`.

Example: `weight_name: "miou"`

If the checkpoint contains weight with the key “miou”, it will set the model state to them. If not, it will try the latest if it exists. If None are found, the model will be randonmly initialized.

Adding a new metric

Within the file `torch_points3d/metrics/model_checkpoint.py`, It contains a mapping dictionnary between a sub `metric_name` and an `optimization function`.

Currently, we support the following metrics.

```
DEFAULT_METRICS_FUNC = {
    "iou": max,
    "acc": max,
    "loss": min,
    "mer": min,
}  # Those map subsentences to their optimization functions
```

3.3.6 Visualization

The framework currently support both wandb and tensorboard

```
# parameters for Weights and Biases
wandb:
    project: benchmarking
    log: False

# parameters for TensorBoard Visualization
tensorboard:
    log: True
```

3.3.7 Custom logging

We use a custom hydra logging message which you can find within `conf/hydra/job_logging/custom.yaml`

```
hydra:
    job_logging:
        formatters:
            simple:
                format: "%(message)s"
        root:
            handlers: [debug_console_handler, file_handler]
        version: 1
        handlers:
            debug_console_handler:
                level: DEBUG
                formatter: simple
                class: logging.StreamHandler
                stream: ext://sys.stdout
            file_handler:
                level: DEBUG
                formatter: simple
                class: logging.FileHandler
                filename: train.log
        disable_existing_loggers: False
```

3.4 Models

torch_points3d.applications.sparseconv3d.**SparseConv3d**(*architecture: str = None, input_nc: int = None, num_layers: int = None, config: omegaconf.DictConfig = None, backend: str = 'minkowski', *args, **kwargs*)

Create a Sparse Conv backbone model based on architecture proposed in https://arxiv.org/abs/1904.08755

Two backends are available at the moment:

- https://github.com/mit-han-lab/torchsparse
- https://github.com/NVIDIA/MinkowskiEngine

architecture [str, optional] Architecture of the model, choose from unet, encoder and decoder

input_nc [int, optional] Number of channels for the input

output_nc [int, optional]

If specified, then we add a fully connected head at the end of the network to provide the requested dimension

num_layers [int, optional] Depth of the network

config [DictConfig, optional] Custom config, overrides the num_layers and architecture parameters

block: Type of resnet block, ResBlock by default but can be any of the blocks in modules/SparseConv3d/modules.py

backend: torchsparse or minkowski

torch_points3d.applications.kpconv.**KPConv**(*architecture: str = None, input_nc: int = None, num_layers: int = None, config: omegaconf.DictConfig = None, *args, **kwargs*)

Create a KPConv backbone model based on the architecture proposed in https://arxiv.org/abs/1904.08889

Parameters

- **architecture** (*str, optional*) – Architecture of the model, choose from unet, encoder and decoder
- **input_nc** (*int, optional*) – Number of channels for the input
- **output_nc** (*int, optional*) – If specified, then we add a fully connected head at the end of the network to provide the requested dimension
- **num_layers** (*int, optional*) – Depth of the network
- **in_grid_size** (*float, optional*) – Size of the grid at the entry of the network. It is divided by two at each layer
- **in_feat** (*int, optional*) – Number of channels after the first convolution. Doubles at each layer
- **config** (*DictConfig, optional*) – Custom config, overrides the num_layers and architecture parameters

torch_points3d.applications.pointnet2.**PointNet2**(*architecture: str = None*, *input_nc: int = None*, *num_layers: int = None*, *config: omegaconf.DictConfig = None*, *multiscale=False*, **args*, ***kwargs*)

Create a PointNet2 backbone model based on the architecture proposed in https://arxiv.org/abs/1706.02413

architecture [str, optional] Architecture of the model, choose from unet, encoder and decoder

input_nc [int, optional] Number of channels for the input

output_nc [int, optional]

If specified, then we add a fully connected head at the end of the network to provide the requested dimension

num_layers [int, optional] Depth of the network

config [DictConfig, optional] Custom config, overrides the num_layers and architecture parameters

torch_points3d.applications.rsconv.**RSConv**(*architecture: str = None*, *input_nc: int = None*, *num_layers: int = None*, *config: omegaconf.DictConfig = None*, **args*, ***kwargs*)

Create a RSConv backbone model based on the architecture proposed in https://arxiv.org/abs/1904.07601

Parameters

- **architecture** (*str, optional*) – Architecture of the model, choose from unet, encoder and decoder
- **input_nc** (*int, optional*) – Number of channels for the input
- **output_nc** (*int, optional*) – If specified, then we add a fully connected head at the end of the network to provide the requested dimension
- **num_layers** (*int, optional*) – Depth of the network
- **config** (*DictConfig, optional*) – Custom config, overrides the num_layers and architecture parameters

3.5 Datasets

Below is a list of the datasets we support as part of the framework. They all inherit from Pytorch Geometric dataset and they can be accessed either as raw datasets or wrapped into a base class that builds test, train and validations data loaders for you. This base class also provides a helper functions for pre-computing neighboors and point cloud sampling at data loading time.

3.5.1 ShapeNet

Raw dataset

class torch_points3d.datasets.segmentation.**ShapeNet**(*root*, *categories=None*, *include_normals=True*, *split='trainval'*, *transform=None*, *pre_transform=None*, *pre_filter=None*, *is_test=False*)

The ShapeNet part level segmentation dataset from the "A Scalable Active Framework for Region Annotation in 3D Shape Collections" paper, containing about 17,000 3D shape point clouds from 16 shape categories. Each category is annotated with 2 to 6 parts.

Parameters

- **root** (*string*) – Root directory where the dataset should be saved.
- **categories** (*string or [string], optional*) – The category of the CAD models (one or a combination of `"Airplane"`, `"Bag"`, `"Cap"`, `"Car"`, `"Chair"`, `"Earphone"`, `"Guitar"`, `"Knife"`, `"Lamp"`, `"Laptop"`, `"Motorbike"`, `"Mug"`, `"Pistol"`, `"Rocket"`, `"Skateboard"`, `"Table"`). Can be explicitly set to `None` to load all categories. (default: `None`)
- **include_normals** (*bool, optional*) – If set to `False`, will not include normal vectors as input features. (default: `True`)
- **split** (*string, optional*) – If `"train"`, loads the training dataset. If `"val"`, loads the validation dataset. If `"trainval"`, loads the training and validation dataset. If `"test"`, loads the test dataset. (default: `"trainval"`)
- **transform** (*callable, optional*) – A function/transform that takes in an `torch_geometric.data.Data` object and returns a transformed version. The data object will be transformed before every access. (default: `None`)
- **pre_transform** (*callable, optional*) – A function/transform that takes in an `torch_geometric.data.Data` object and returns a transformed version. The data object will be transformed before being saved to disk. (default: `None`)
- **pre_filter** (*callable, optional*) – A function that takes in an `torch_geometric.data.Data` object and returns a boolean value, indicating whether the data object should be included in the final dataset. (default: `None`)

Wrapped dataset

class torch_points3d.datasets.segmentation.**ShapeNetDataset**(*dataset_opt*)

Wrapper around ShapeNet that creates train and test datasets.

Parameters **dataset_opt** (*omegaconf.DictConfig*) – Config dictionary that should contain

- dataroot
- category: List of categories or All
- normal: bool, include normals or not
- pre_transforms
- train_transforms
- test_transforms

- val_transforms

3.5.2 S3DIS

Raw dataset

class torch_points3d.datasets.segmentation.**S3DISOriginalFused**(*root*, *test_area=6*, *split='train'*, *transform=None*, *pre_transform=None*, *pre_collate_transform=None*, *pre_filter=None*, *keep_instance=False*, *verbose=False*, *debug=False*)

Original S3DIS dataset. Each area is loaded individually and can be processed using a pre_collate transform. This transform can be used for example to fuse the area into a single space and split it into spheres or smaller regions. If no fusion is applied, each element in the dataset is a single room by default.

http://buildingparser.stanford.edu/dataset.html

Parameters

- **root** (*str*) – path to the directory where the data will be saved
- **test_area** (*int*) – number between 1 and 6 that denotes the area used for testing
- **split** (*str*) – can be one of train, trainval, val or test
- **pre_collate_transform** – Transforms to be applied before the data is assembled into samples (apply fusing here for example)
- **keep_instance** (*bool*) – set to True if you wish to keep instance data
- **pre_transform** –
- **transform** –
- **pre_filter** –

class torch_points3d.datasets.segmentation.**S3DISSphere**(*root*, *sample_per_epoch=100*, *radius=2*, **args*, ***kwargs*)

Small variation of S3DISOriginalFused that allows random sampling of spheres within an Area during training and validation. Spheres have a radius of 2m. If sample_per_epoch is not specified, spheres are taken on a 2m grid.

http://buildingparser.stanford.edu/dataset.html

Parameters

- **root** (*str*) – path to the directory where the data will be saved
- **test_area** (*int*) – number between 1 and 6 that denotes the area used for testing
- **train** (*bool*) – Is this a train split or not
- **pre_collate_transform** – Transforms to be applied before the data is assembled into samples (apply fusing here for example)
- **keep_instance** (*bool*) – set to True if you wish to keep instance data

- **sample_per_epoch** – Number of spheres that are randomly sampled at each epoch (-1 for fixed grid)
- **radius** – radius of each sphere
- **pre_transform** –
- **transform** –
- **pre_filter** –

Wrapped dataset

class torch_points3d.datasets.segmentation.**S3DIS1x1Dataset**(*dataset_opt*)

class torch_points3d.datasets.segmentation.**S3DISFusedDataset**(*dataset_opt*)

Wrapper around S3DISSphere that creates train and test datasets.

http://buildingparser.stanford.edu/dataset.html

Parameters **dataset_opt** (*omegaconf.DictConfig*) – Config dictionary that should contain

- dataroot
- fold: test_area parameter
- pre_collate_transform
- train_transforms
- test_transforms

3.5.3 Scannet

Raw dataset

class torch_points3d.datasets.segmentation.**Scannet**(*root, split='train', transform=None, pre_transform=None, pre_filter=None, version='v2', use_instance_labels=False, use_instance_bboxes=False, donotcare_class_ids=[], max_num_point=None, process_workers=4, types=['.txt', '_vh_clean_2.ply', '_vh_clean_2.0.010000.segs.json', '.aggregation.json'], normalize_rgb=True, is_test=False*)

Scannet dataset, you will have to agree to terms and conditions by hitting enter so that it downloads the dataset.

http://www.scan-net.org/

Parameters

- **root** (*str*) – Path to the data
- **split** (*str, optional*) – Split used (train, val or test)

- **(callable, optional)** (*pre_filter*) – A function/transform that takes in an `torch_geometric.data.Data` object and returns a transformed version. The data object will be transformed before every access.
- **(callable, optional)** – A function/transform that takes in an `torch_geometric.data.Data` object and returns a transformed version. The data object will be transformed before being saved to disk.
- **(callable, optional)** – A function that takes in an `torch_geometric.data.Data` object and returns a boolean value, indicating whether the data object should be included in the final dataset.
- **version** (*str, optional*) – version of scannet, by default "v2"
- **use_instance_labels** (*bool, optional*) – Wether we use instance labels or not, by default False
- **use_instance_bboxes** (*bool, optional*) – Wether we use bounding box labels or not, by default False
- **donotcare_class_ids** (*list, optional*) – Class ids to be discarded
- **max_num_point** (*[type], optional*) – Max number of points to keep during the pre processing step
- **use_multiprocessing** (*bool, optional*) – Wether we use multiprocessing or not
- **process_workers** (*int, optional*) – Number of process workers
- **normalize_rgb** (*bool, optional*) – Normalise rgb values, by default True

Wrapped dataset

class torch_points3d.datasets.segmentation.**ScannetDataset**(*dataset_opt*)

Wrapper around Scannet that creates train and test datasets.

Parameters **dataset_opt** (*omegaconf.DictConfig*) – Config dictionary that should contain

- dataroot
- version
- max_num_point (optional)
- use_instance_labels (optional)
- use_instance_bboxes (optional)
- donotcare_class_ids (optional)
- pre_transforms (optional)
- train_transforms (optional)
- val_transforms (optional)

3.6 Transforms

class torch_points3d.core.data_transform.**PointCloudFusion**
This transform is responsible to perform a point cloud fusion from a list of data

- If a list of data is provided -> Create one Batch object with all data
- If a list of list of data is provided -> Create a list of fused point cloud

class torch_points3d.core.data_transform.**GridSphereSampling**(*radius*, *grid_size=None*, *delattr_kd_tree=True*, *center=True*)
Fits the point cloud to a grid and for each point in this grid, create a sphere with a radius r

Parameters

- **radius** (*float*) – Radius of the sphere to be sampled.
- **grid_size** (*float, optional*) – Grid_size to be used with GridSampling3D to select spheres center. If None, radius will be used
- **delattr_kd_tree** (*bool, optional*) – If True, KDTREE_KEY should be deleted as an attribute if it exists
- **center** (*bool, optional*) – If True, a centre transform is apply on each sphere.

class torch_points3d.core.data_transform.**RandomSphere**(*radius*, *strategy='random'*, *class_weight_method='sqrt'*, *center=True*)
Select points within a sphere of a given radius. The centre is chosen randomly within the point cloud.

Parameters

- **radius** (*float*) – Radius of the sphere to be sampled.
- **strategy** (*str*) – choose between *random* and *freq_class_based*. The *freq_class_based* favors points with low frequency class. This can be used to balance unbalanced datasets
- **center** (*bool*) – if True then the sphere will be moved to the origin

class torch_points3d.core.data_transform.**GridSampling3D**(*size*, *quantize_coords=False*, *mode='mean'*, *verbose=False*)
Clusters points into voxels with size `size`. :param size: Size of a voxel (in each dimension). :type size: float :param quantize_coords: If True, it will convert the points into their associated sparse coordinates within the grid and store

the value into a new *coords* attribute

Parameters **mode** (*string:*) – The mode can be either *last* or *mean*. If mode is *mean*, all the points and their features within a cell will be averaged If mode is *last*, one random points per cell will be selected with its associated features

class torch_points3d.core.data_transform.**RandomSymmetry**(*axis=[False, False, False]*)
Apply a random symmetry transformation on the data

Parameters **axis** (*Tuple[bool,bool,bool], optional*) – axis along which the symmetry is applied

class torch_points3d.core.data_transform.**RandomNoise**(*sigma=0.01*, *clip=0.05*)
Simple isotropic additive gaussian noise (Jitter)

Parameters

- **sigma** – Variance of the noise
- **clip** – Maximum amplitude of the noise

class torch_points3d.core.data_transform.**RandomScaleAnisotropic**(*scales=None*, *anisotropic=True*)
Scales node positions by a randomly sampled factor `s1, s2, s3` within a given interval, *e.g.*, resulting in the transformation matrix

$$\begin{bmatrix} s1 & 0 & 0 \\ 0 & s2 & 0 \\ 0 & 0 & s3 \end{bmatrix}$$

for three-dimensional positions.

Parameters **scales** – scaling factor interval, e.g. `(a, b)`, then scale is randomly sampled from the range `a <= b`.

class torch_points3d.core.data_transform.**MultiScaleTransform**(*strategies*)
Pre-computes a sequence of downsampling / neighboorhood search on the CPU. This currently only works on PARTIAL_DENSE formats

Parameters **strategies** (*Dict[str, object]*) – Dictionary that contains the samplers and neighbour_finder

class torch_points3d.core.data_transform.**ModelInference**(*checkpoint_dir*, *model_name*, *weight_name*, *feat_name*, *num_classes=None*, *mock_dataset=True*)
Base class transform for performing a point cloud inference using a pre_trained model Subclass and implement the `__call__` method with your own forward. See `PointNetForward` for an example implementation.

Parameters

- **checkpoint_dir** (*str*) – Path to a checkpoint directory
- **model_name** (*str*) – Model name, the file `checkpoint_dir/model_name.pt` must exist

class torch_points3d.core.data_transform.**PointNetForward**(*checkpoint_dir*, *model_name*, *weight_name*, *feat_name*, *num_classes*, *mock_dataset=True*)
Transform for running a PointNet inference on a Data object. It assumes that the model has been trained for segmentation.

Parameters

- **checkpoint_dir** (*str*) – Path to a checkpoint directory
- **model_name** (*str*) – Model name, the file `checkpoint_dir/model_name.pt` must exist
- **weight_name** (*str*) – Type of weights to load (best for iou, best for loss etc...)
- **feat_name** (*str*) – Name of the key in Data that will hold the output of the forward

- **num_classes** (*int*) – Number of classes that the model was trained on

class torch_points3d.core.data_transform.**AddFeatsByKeys**(*list_add_to_x: List[bool], feat_names: List[str], input_nc_feats: List[Optional[int]] = None, stricts: List[bool] = None, delete_feats: List[bool] = None*)

This transform takes a list of attributes names and if allowed, add them to x

Example

Before calling "AddFeatsByKeys", if data.x was empty

- transform: AddFeatsByKeys params:

  list_add_to_x: [False, True, True] feat_names: ['normal', 'rgb', "elevation"] input_nc_feats: [3, 3, 1]

After calling "AddFeatsByKeys", data.x contains "rgb" and "elevation". Its shape[-1] == 4 (rgb:3 + elevation:1) If input_nc_feats was [4, 4, 1], it would raise an exception as rgb dimension is only 3.

list_add_to_x: List[bool] For each boolean within list_add_to_x, control if the associated feature is going to be concatenated to x

feat_names: List[str] The list of features within data to be added to x

input_nc_feats: List[int], optional If provided, evaluate the dimension of the associated feature shape[-1] found using feat_names and this provided value. It allows to make sure feature dimension didn't change

stricts: List[bool], optional Recommended to be set to list of True. If True, it will raise an Exception if feat isn't found or dimension doesn t match.

delete_feats: List[bool], optional Wether we want to delete the feature from the data object. List length must match teh number of features added.

class torch_points3d.core.data_transform.**AddFeatByKey**(*add_to_x*, *feat_name*, *input_nc_feat=None*, *strict=True*)

This transform is responsible to get an attribute under feat_name and add it to x if add_to_x is True

add_to_x: bool Control if the feature is going to be added/concatenated to x

feat_name: str The feature to be found within data to be added/concatenated to x

input_nc_feat: int, optional If provided, check if feature last dimension maches provided value.

strict: bool, optional Recommended to be set to True. If False, it won't break if feat isn't found or dimension doesn t match. (default: `True`)

class torch_points3d.core.data_transform.**RemoveAttributes**(*attr_names=[]*, *strict=False*)

This transform allows to remove unnecessary attributes from data for optimization purposes

Parameters

- **attr_names** (*list*) – Remove the attributes from data using the provided *attr_name* within attr_names
- **strict** (*bool=False*) – Wether True, it will raise an execption if the provided attr_name isn t within data keys.

class torch_points3d.core.data_transform.**ShuffleData**
This transform allow to shuffle feature, pos and label tensors within data

class torch_points3d.core.data_transform.**ShiftVoxels**(*apply_shift=True*)
Trick to make Sparse conv invariant to even and odds coordinates https://github.com/chrischoy/SpatioTemporalSegmentation/blob/master/lib/train.py#L78

Parameters **apply_shift** (*bool:*) – Whether to apply the shift on indices

class torch_points3d.core.data_transform.**ChromaticTranslation**(*trans_range_ratio=0.1*)
Add random color to the image, data must contain an rgb attribute between 0 and 1

Parameters **trans_range_ratio** – ratio of translation i.e. tramnslation = 2 * ratio * rand(-0.5, 0.5) (default: 1e-1)

class torch_points3d.core.data_transform.**ChromaticAutoContrast**(*randomize_blend_factor=True*, *blend_factor=0.5*)
Rescale colors between 0 and 1 to enhance contrast

Parameters

- **randomize_blend_factor** – Blend factor is random
- **blend_factor** – Ratio of the original color that is kept

class torch_points3d.core.data_transform.**ChromaticJitter**(*std=0.01*)
Jitter on the rgb attribute of data

Parameters **std** – standard deviation of the Jitter

class torch_points3d.core.data_transform.**Jitter**(*mu=0*, *sigma=0.01*, *p=0.95*)
add a small gaussian noise to the feature. :param mu: mean of the gaussian noise :type mu: float :param sigma: standard deviation of the gaussian noise :type sigma: float :param p: probability of noise :type p: float

class torch_points3d.core.data_transform.**RandomDropout**(*dropout_ratio: float = 0.2*, *dropout_application_ratio: float = 0.5*)
Randomly drop points from the input data

Parameters

- **dropout_ratio** (*float, optional*) – Ratio that gets dropped
- **dropout_application_ratio** (*float, optional*) – chances of the dropout to be applied

class torch_points3d.core.data_transform.**DropFeature**(*drop_proba=0.2*, *feature_name='rgb'*)
Sets the given feature to 0 with a given probability

Parameters

- **drop_proba** – Probability that the feature gets dropped
- **feature_name** – Name of the feature to drop

class torch_points3d.core.data_transform.**NormalizeFeature**(*feature_name*, *standardize=False*)
Normalize a feature. By default, features will be scaled between [0,1]. Should only be applied on a dataset-level.

Parameters **standardize** (*bool: Will use standardization rather than scaling.*) –

class torch_points3d.core.data_transform.**PCACompute**

compute Principal Component Analysis of a point cloud $x_1, \ldots, x_n$. It computes the eigenvalues and the eigenvectors of the matrix C which is the covariance matrix of the point cloud:

$$x_{centered} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$C = \frac{1}{n}\sum_{i=1}^{n}(x_i - x_{centered})(x_i - x_{centered})^T$$

store the eigen values and the eigenvectors in data. in eigenvalues attribute and eigenvectors attributes. data.eigenvalues is a tensor $(\lambda_1, \lambda_2, \lambda_3)$ such that $\lambda_1 \leq \lambda_2 \leq \lambda_3$.

data.eigenvectors is a 3 x 3 matrix such that the column are the eigenvectors associated to their eigenvalues Therefore, the first column of data.eigenvectors estimates the normal at the center of the pointcloud.

class torch_points3d.core.data_transform.**ClampBatchSize**(*num_points=100000*)

Drops sample in a batch if the batch gets too large

Parameters **num_points** (*int, optional*) – Maximum number of points per batch, by default 100000

class torch_points3d.core.data_transform.**LotteryTransform**(*transform_options*)

Transforms which draw a transform randomly among several transforms indicated in transform options Examples

Parameters **Omegaconf list which contains the transform** (*transform_options*) –

class torch_points3d.core.data_transform.**RandomParamTransform**(*transform_name*, *transform_params*)

create a transform with random parameters

Example (on the yaml)

```
transform: RandomParamTransform
    params:
        transform_name: GridSampling3D
        transform_params:
            size:
                min: 0.1
                max: 0.3
                type: "float"
            mode:
                value: "last"
```

We can also draw random numbers for two parameters, integer or float

```
transform: RandomParamTransform
    params:
        transform_name: RandomSphereDropout
        transform_params:
            radius:
                min: 1
                max: 2
                type: "float"
            num_sphere:
                min: 1
```

(continues on next page)

(continued from previous page)

```
                max: 5
                type: "int"
```

Parameters

- **transform_name** (*string:*) – the name of the transform
- **transform_options** (*Omegaconf Dict*) – contains the name of a variables as a key and min max type as value to specify the range of the parameters and the type of the parameters or it contains the value "value" to specify a variables (see Example above)

class torch_points3d.core.data_transform.**Select**(*indices=None*)

Selects given points from a data object

Parameters **indices** (*torch.Tensor*) – indeices of the points to keep. Can also be a boolean mask

torch_points3d.core.data_transform.**NormalizeRGB**(*normalize=True*)

Normalize rgb between 0 and 1

Parameters **normalize** (*bool: Whether to normalize the rgb attributes*) –

torch_points3d.core.data_transform.**ElasticDistortion**(*apply_distorsion: bool = True, granularity: List = [0.2, 0.8], magnitude=[0.4, 1.6]*)

Apply elastic distortion on sparse coordinate space. First projects the position onto a voxel grid and then apply the distortion to the voxel grid.

Parameters

- **granularity** (*List[float]*) – Granularity of the noise in meters
- **magnitude** (*List[float]*) – Noise multiplier in meters

Returns **data** – Returns the same data object with distorted grid

Return type Data

torch_points3d.core.data_transform.**Random3AxisRotation**(*apply_rotation: bool = True, rot_x: float = None, rot_y: float = None, rot_z: float = None*)

Rotate pointcloud with random angles along x, y, z axis

The angles should be given *in degrees*.

Parameters

- **apply_rotation** (*bool:*) – Whether to apply the rotation
- **rot_x** (*float*) – Rotation angle in degrees on x axis
- **rot_y** (*float*) – Rotation anglei n degrees on y axis
- **rot_z** (*float*) – Rotation angle in degrees on z axis

torch_points3d.core.data_transform.**RandomCoordsFlip**(*ignored_axis, is_temporal=False, p=0.95*)

torch_points3d.core.data_transform.**ScalePos**(*scale=None*)

torch_points3d.core.data_transform.**RandomWalkDropout**(*dropout_ratio: float = 0.05, num_iter: int = 5000, radius: float = 0.5, max_num: int = -1, skip_keys: List = []*)

randomly drop points from input data using random walk

Parameters

- **dropout_ratio** (*float, optional*) – Ratio that gets dropped
- **num_iter** (*int, optional*) – number of iterations
- **radius** (*float, optional*) – radius of the neighborhood search to create the graph
- **max_num** (*int optional*) – max number of neighbors
- **skip_keys** (*List optional*) – skip_keys where we don't apply the mask

torch_points3d.core.data_transform.**RandomSphereDropout**(*num_sphere: int = 10, radius: float = 5, grid_size_center: float = 0.01*)

drop out of points on random spheres of fixed radius. This function takes n random balls of fixed radius r and drop out points inside these balls.

Parameters

- **num_sphere** (*int, optional*) – number of random spheres
- **radius** (*float, optional*) – radius of the spheres

torch_points3d.core.data_transform.**SphereCrop**(*radius: float = 50*)

crop the point cloud on a sphere. this function. takes a ball of radius radius centered on a random point and points outside the ball are rejected.

Parameters **radius** (*float, optional*) – radius of the sphere

torch_points3d.core.data_transform.**CubeCrop**(*c: float = 1, rot_x: float = 180, rot_y: float = 180, rot_z: float = 180, grid_size_center: float = 0.01*)

Crop cubically the point cloud. This function take a cube of size c centered on a random point, then points outside the cube are rejected.

Parameters

- **c** (*float, optional*) – half size of the cube
- **rot_x** (*float_otional*) – rotation of the cube around x axis
- **rot_y** (*float_otional*) – rotation of the cube around x axis
- **rot_z** (*float_otional*) – rotation of the cube around x axis

torch_points3d.core.data_transform.**compute_planarity**(*eigenvalues*)

compute the planarity with respect to the eigenvalues of the covariance matrix of the pointcloud let $\lambda_1, \lambda_2, \lambda_3$ be the eigenvalues st:

$$\lambda_1 \leq \lambda_2 \leq \lambda_3$$

then planarity is defined as:

$$planarity = \frac{\lambda_2 - \lambda_1}{\lambda_3}$$

3.7 Filters

class torch_points3d.core.data_transform.**PlanarityFilter**(*thresh=0.3*, *is_leq=True*)

compute planarity and return false if the planarity of a pointcloud is above or below a threshold

Parameters

- **thresh** (*float, optional*) – threshold to filter low planar pointcloud
- **is_leq** (*bool, optional*) – choose whether planarity should be lesser or equal than the threshold or greater than the threshold.

class torch_points3d.core.data_transform.**RandomFilter**(*thresh=0.3*)

Randomly select an elem of the dataset (to have smaller dataset) with a bernouilli distribution of parameter thresh.

Parameters **thresh** (*float, optional*) – the parameter of the bernouilli function

class torch_points3d.core.data_transform.**FCompose**(*list_filter*, *boolean_operation=numpy.logical_and*)

allow to compose different filters using the boolean operation

Parameters

- **list_filter** (*list*) – list of different filter functions we want to apply
- **boolean_operation** (*function, optional*) – boolean function to compose the filter (take a pair and return a boolean)

INDEX

S