PyNIF3D

Release 0.1

Woven Core, Inc.

Aug 18, 2021

CONTENTS

CHAPTER

ONE

PYNIF3D

PyNIF3D is an open-source PyTorch-based library for research on neural implicit functions (NIF)-based 3D geometry representation. It aims to accelerate research by providing a modular design that allows for easy extension and combination of NIF-related components, as well as readily available paper implementations and dataset loaders.

As of August 2021, the following implementations are supported:

- NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis (Mildenhall et al., 2020)
- Convolutional Occupancy Networks (Peng et al., 2020)
- Multiview Neural Surface Reconstruction by Disentangling Geometry and Appearance (Yariv et al., 2020)

1.1 Installation

To get started with PyNIF3D, you can use `pip` to install a copy of this repository on your local machine or build the provided Dockerfile.

1.1.1 Local Installation

```
pip install --user "https://github.com/pfnet/pynif3d.git"
```

The following packages need to be installed in order to ensure the proper functioning of all the PyNIF3D features:

- torch_scatter>=1.3.0
- torchsearchsorted>=1.0

A script has been provided to take care of the installation steps for you. Please download it to a directory of choice and run:

```
bash post_install.bash
```

1.1.2 Docker Build

Enabling CUDA Support

Please make sure the following dependencies are installed in order to build the Docker image with CUDA support:

- nvidia-docker
- nvidia-container-runtime

Then register the `nvidia` runtime by adding the following to `/etc/docker/daemon.json`:

```
{
    "runtimes": {
        "nvidia": {
            [...]
        }
    },
    "default-runtime": "nvidia"
}
```

Restart the Docker daemon:

```
sudo systemctl restart docker
```

You should now be able to build a Docker image with CUDA support.

Building Dockerfile

```
git clone https://github.com/pfnet/pynif3d.git
cd pynif3d && nvidia-docker build -t pynif3d .
```

Running the Container

```
nvidia-docker run -it pynif3d bash
```

1.2 Tutorials

Get started with PyNIF3D using the examples provided below:

In addition to the tutorials, pretrained models are also provided and ready to be used. Please consult this page for more information.

1.3 License

PyNIF3D is released under the MIT license. Please refer to this document for more information.

1.4 Contributing

We welcome any new contributions to PyNIF3D. Please make sure to read the contributing guidelines before submitting a pull request.

1.5 Documentation

Learn more about PyNIF3D by reading the API documentation.

CHAPTER TWO

API DOCUMENTATION

2.1 pynif3d.aggregation

class pynif3d.aggregation.**NeRFAggregator**(*background_color=None*, *noise_std=0.0*)

Bases: `torch.nn.modules.module.Module`

Color aggregation function. Takes the raw predictions obtained from a NIF model, the depth values and a ray direction to produce the final color of a pixel. Please refer to the original NeRF paper for more information.

Usage:

```
# Assume the models are given.
nif_model = NeRFModel()
renderer = PointRenderer()
aggregator = NeRFAggregator()

# Get the RGBA values for the corresponding points and view directions.
rgba_values = renderer(nif_model, query_points, view_directions)

# Aggregate the computed RGBA values to form the RGB maps, depth maps etc.
rendered_data = aggregator(rgba_values, ray_z_values, ray_directions)
rgb_map, depth_map, disparity_map, alpha_map, weight = rendered_data
```

Parameters

- **background_color** (*torch.Tensor*) – The background color to be added for rendering. If set to None, no background will be added. Default is None.
- **noise_std** (*float*) – The standard deviation of the noise to be added to alpha. Set 0 to disable the noise addition. Default is 0.

forward(*raw_model_prediction*, *z_vals*, *rays_d*)

Parameters

- **raw_model_prediction** (*torch.Tensor*) – The prediction output of a NIF model. Its shape is `(number_of_rays, number_of_points_per_ray, 4)`.
- **z_vals** (*torch.Tensor*) – Depth values of the rays. Its shape is `(number_of_rays, number_of_points_per_ray, 4)`.
- **rays_d** (*torch.Tensor*) – The direction vector of the rays. Its shape is `(number_of_rays, 3)`.

Returns

Tuple containing:

- **rgb_map** (torch.Tensor): The RGB pixel values of the rays. Its shape is `(number_of_rays, 3)`.
- **depth_map** (torch.Tensor): The depth pixel values of the rays. Its shape is `(number_of_rays,)`.
- **disparity_map** (torch.Tensor): The inverse depth pixel values of the rays. Its shape is `(number_of_rays,)`.
- **alpha_map** (torch.Tensor): The transparency/alpha value of each pixel. Its shape is `(number_of_rays,)`.
- **weights** (torch.Tensor): The weights of each sampled points across a ray which impact the final RGB/depth/disparity/transparency value of a pixel. Its shape is `(number_of_rays, number_of_points_per_ray)`.

Return type tuple

training: bool

2.2 pynif3d.camera

`class pynif3d.camera.`**`CameraRayGenerator`**`(`*height*, *width*, *focal_x*, *focal_y*, *center_x=None*, *center_y=None*`)`

Bases: `torch.nn.modules.module.Module`

Generates rays for each pixel of a pinhole camera model. Takes the spatial dimensions of the camera's image plane along with focal lengths and camera pose to generate a ray for each pixel.

Usage:

```
ray_generator = CameraRayGenerator(image_height, image_width, focal_x, focal_y)
ray_directions, ray_origins, view_directions = ray_generator(camera_poses)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`forward``(`*camera_poses*`)`

Parameters **camera_poses** (*torch.Tensor*) – Tensor containing the Rt pose matrices. Its shape is `(N, 3, 4)` or `(3, 4)`.

Returns

Tuple containing:

- **rays_o** (torch.Tensor): The origin coordinates of the rays. Its shape is `(batch_size, 3, height, width)`.
- **rays_d** (torch.Tensor): The non-normalized direction vector of the rays. Its shape is `(batch_size, 3, height, width)`.
- **view_dirs** (torch.Tensor): The unit-normalized direction vector of the rays. Its shape is `(batch_size, 3, height, width)`.

Return type tuple

training: bool

class pynif3d.camera.**SphereRayTracer**(*sdf_model*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

Determines the intersection between a set of rays and a surface defined by an implicit representation using the sphere tracing algorithm.

Usage:

```
# Assume an SDF model (torch.nn.Module) is given.
ray_tracer = SphereRayTracer(sdf_model)
points, z_vals, mask_not_converged = ray_tracer(ray_directions, ray_origins)
```

Parameters

- **sdf_model** (*instance*) – Instance of an SDF model.
- **kwargs** (*dict*) –
  - **sdf_threshold** (float): The SDF threshold that is used to determine whether points are close enough to the surface or not. The closer they are, the lower the SDF value becomes.
  - **n_iterations** (int): The number of iterations that the sphere tracing algorithm is run for.
  - **n_fix_iterations** (int): The number of iterations that the algorithm for correcting overshooting SDF values is run for.

fix_overshoot(*points*, *z_vals*, *rays_d*, *rays_o*, *sdf_vals*, *next_sdf_vals*)

forward(*rays_d*, *rays_o*, *z_vals=None*)

Parameters

- **rays_d** (*torch.Tensor*) – Tensor containing the ray directions. Its shape is `(batch_size, n_rays, 3)` or `(n_rays, 3)`.
- **rays_o** (*torch.Tensor*) – Tensor containing the ray origins. Its shape is `(batch_size, n_rays, 3)` or `(n_rays, 3)`.
- **z_vals** (*torch.Tensor*) – Tensor containing the initial Z values. Its shape is `(batch_size, n_rays)` or `(n_rays,)`.

Returns

Tuple containing the intersection points (as a torch.Tensor with shape `(2, n_rays, 3)`), the Z values along the ray (as a torch.Tensor with shape `(2, n_rays,)`) and a mask specifying which points the algorithm has not converged for (as a torch.Tensor with shape `(2, n_rays,)`). Note that the first channel encodes the intersection information between the ray and the sphere that is processed at each iteration of the algorithm.

Return type tuple

training: bool

2.3 pynif3d.common

exception pynif3d.common.**AttributeNotExistingException**(*variable_name*, *attribute_name*)
Bases: *pynif3d.common.verification.ExceptionBase*

exception pynif3d.common.**DevicesNotMatchingException**(*variable1_name*, *variable2_name*)
Bases: *pynif3d.common.verification.ExceptionBase*

exception pynif3d.common.**ExceptionBase**
Bases: Exception

class pynif3d.common.**Iterable**
Bases: object

exception pynif3d.common.**NoneVariableException**(*variable_name*)
Bases: *pynif3d.common.verification.ExceptionBase*

exception pynif3d.common.**NotBooleanException**(*variable_name*)
Bases: *pynif3d.common.verification.ExceptionBase*

exception pynif3d.common.**NotCallableException**(*fn_name*, *property_name*)
Bases: *pynif3d.common.verification.ExceptionBase*

exception pynif3d.common.**NotEqualException**(*variable1_name*, *variable2_name*)
Bases: *pynif3d.common.verification.ExceptionBase*

exception pynif3d.common.**NotInOptionsException**(*variable_name*, *options*)
Bases: *pynif3d.common.verification.ExceptionBase*

exception pynif3d.common.**NotIterableException**(*variable_name*)
Bases: *pynif3d.common.verification.ExceptionBase*

exception pynif3d.common.**NotPositiveIntegerException**(*variable_name*)
Bases: *pynif3d.common.verification.ExceptionBase*

exception pynif3d.common.**PathNotFoundException**(*path*)
Bases: *pynif3d.common.verification.ExceptionBase*

exception pynif3d.common.**ShapesNotMatchingException**(*variable1_name*, *variable2_name*)
Bases: *pynif3d.common.verification.ExceptionBase*

exception pynif3d.common.**WrongAxisException**(*variable_name*, *axis*)
Bases: *pynif3d.common.verification.ExceptionBase*

exception pynif3d.common.**WrongShapeException**(*variable_name*, *shape*)
Bases: *pynif3d.common.verification.ExceptionBase*

pynif3d.common.**check_axis**(*variable*, *axis*, *variable_name*)

pynif3d.common.**check_bool**(*variable*, *variable_name*)

pynif3d.common.**check_callable**(*fn*, *property_name*, *fn_name*)

pynif3d.common.**check_devices_match**(*variable1*, *variable2*, *variable1_name*, *variable2_name*)

pynif3d.common.**check_equal**(*variable1*, *variable2*, *variable1_name*, *variable2_name*)

pynif3d.common.**check_in_options**(*variable*, *options*, *variable_name*)

pynif3d.common.**check_iterable**(*variable*, *variable_name*)

pynif3d.common.**check_lengths_match**(*variable1*, *variable2*, *variable1_name*, *variable2_name*)

pynif3d.common.**check_not_none**(*variable*, *variable_name*) `pynif3d.common.`**`check_path_exists`**(*path*)

`pynif3d.common.`**`check_pos_int`**(*variable*, *variable_name*)

`pynif3d.common.`**`check_shape`**(*variable*, *shape*, *variable_name*)

`pynif3d.common.`**`check_shapes_match`**(*variable1*, *variable2*, *variable1_name*, *variable2_name*)

`pynif3d.common.`**`check_true`**(*variable*, *variable_name*)

`pynif3d.common.`**`coordinate2index`**(*coordinates*, *resolution*)

The function to convert 2D / 3D coordinates into 1D indices. It supports only the square / cubical resolution.

Parameters

- **`coordinates`** (*`Tensor`*) – 2D or 3D coordinates. axis=2 shall contain the coordinate information.
- **`resolution`** (*`int`*) – The spatial resolution of the coordinates to be parsed.

Returns Tensor containing 1D locations indices of the input coordinates.

`pynif3d.common.`**`decompose_projection`**(*projection*)

`pynif3d.common.`**`init_Conv2d`**(*size_in*, *size_out*, *kernel_size*, ***kwargs*)

`pynif3d.common.`**`init_ConvTranspose2d`**(*size_in*, *size_out*, *kernel_size*, ***kwargs*)

`pynif3d.common.`**`init_Linear`**(*size_in*, *size_out*, ***kwargs*)

`pynif3d.common.`**`is_image_file`**(*file_path*)

`pynif3d.common.`**`normalize_coordinate`**(*point*, *padding=0.1*, *plane='xz'*, *eps=1e-05*)

The function to normalize coordinates to [0, 1], considering input within the limits of *[-0.5 * (1 + padding), +0.5 * (1 + padding)]*. The input limits are not strongly enforced.

Parameters

- **`point`** (*`Tensor`*) – The tensor of the points to be normalized within given interval. It has to have shape (batch_size, n_points, 3)
- **`padding`** (*`float`*) – The ratio of padding to be applied within [lim_min, lim_max]. Default is 0.1.
- **`plane`** (*`str`*) – The plane to apply the normalization. Options are (“xy”, “xz”, “yz”, “grid”). Default is “xz”.
- **`eps`** – The epsilon to prevent zero-division. Default is 1e-5.

Returns Tensor of the points scaled and shifted to [lim_min, lim_max]

2.4 pynif3d.datasets

`class` `pynif3d.datasets.`**`BaseDataset`**(*data_directory*, *mode*)

Bases: `Generic[torch.utils.data.dataset.T_co]`

Base dataset class. All the custom datasets shall inherit this class and implement the required functions by overriding them.

Parameters

- **`data_directory`** (*`str`*) – The dataset root directory.
- **`mode`** (*`str`*) – The dataset usage mode (“train”, “val” or “test”).

`download`(*url*, *save_directory*, *archive_format*, *md5=None*)

`class pynif3d.datasets.`**`Blender`**(*data_directory*, *mode*, *scene*, *half_resolution=False*, *white_background=True*, *download=False*)

Bases: `Generic[torch.utils.data.dataset.T_co]`

Implementation of the synthetic dataset (Blender).

Please refer to the following paper for more information: https://arxiv.org/abs/2003.08934

Note: This implementation is based on the code from: https://github.com/bmild/nerf

Usage:

```
mode = "train"
scene = "chair"
dataset = Blender(data_directory, mode, scene)
```

Parameters

- **`data_directory`** (*str*) – The dataset base directory (see BaseDataset).
- **`mode`** (*str*) – The dataset usage mode (see BaseDataset).
- **`scene`** (*str*) – The scene name (“chair”, “drums”, “ficus”. . .).
- **`half_resolution`** (*bool*) – Boolean indicating whether to load the dataset in half resolution (True) or full resolution (False)
- **`white_background`** (*bool*) – Boolean indicating whether to set the dataset’s background color to white (True) or leave it as it is (False)
- **`download`** (*bool*) – Flag indicating whether to automatically download the dataset (True) or not (False).

`dataset_md5 = 'ac0cfb13b1e4ff748b132abc8e8c26b6'`

`dataset_url = 'https://drive.google.com/u/0/uc?id=18JxhpWD-4ZmuFKLzKlAw-w5PpzZxXOcG'`

`class pynif3d.datasets.`**`DTUMVSIDR`**(*data_directory*, *mode*, *scan_id*, *download=False*, ***kwargs*)

Bases: `Generic[torch.utils.data.dataset.T_co]`

Implementation of the DTU MVS dataset, as used in the IDR paper:

Multiview Neural Surface Reconstruction by Disentangling Geometry and Appearance Yariv et al., NeurIPS, 2020

Please refer to the following paper for more information: https://arxiv.org/abs/2003.09852

Usage:

```
mode = "train"
scan_id = 110
dataset = DTUMVSIDR(data_directory, mode, scan_id)
```

Parameters

- **`data_directory`** (*str*) – The dataset base directory (see BaseDataset).
- **`mode`** (*str*) – The dataset usage mode (see BaseDataset).

- **scan_id** (*int*) – ID of the scan.
- **download** (*bool*) – Flag indicating whether to automatically download the dataset (True) or not (False).
- **kwargs** (*dict*) –
  - **calibration_file** (str): The name of the calibration file. Default is "cameras_linear_init.npz".

`dataset_md5 = 'b1ad1eff5c4a4f99ae4d3503e976dafb'`

`dataset_url = 'https://www.dropbox.com/s/ujmakiaiekdl6sh/DTU.zip?dl=1'`

`class pynif3d.datasets.DTUMVSPixelNeRF`(*data_directory*, *mode*, *scan_ids_file*, *download=False*)

Bases: `Generic[torch.utils.data.dataset.T_co]`

Implementation of the DTU MVS dataset, as used in the pixelNeRF paper:

pixelNeRF: Neural Radiance Fields from One or Few Images Yu et al., CVPR, 2021

Please refer to the following paper for more information: https://arxiv.org/abs/2012.02190

Parameters

- **data_directory** (*str*) – The dataset base directory (see BaseDataset).
- **mode** (*str*) – The dataset usage mode (see BaseDataset).
- **scan_ids_file** (*str*) – The path to the file that contains the IDs of the scans that need to be processed.
- **download** (*bool*) – Flag indicating whether to automatically download the dataset (True) or not (False).

`dataset_md5 = '02af85c542238d9832e348caee2a6bba'`

`dataset_url = 'https://drive.google.com/uc?id=1aTSmJa8Oo2qCc2Ce2kT90MHEA6UTSBKj'`

`class pynif3d.datasets.DeepVoxels`(*data_directory*, *mode*, *scene*, *download=False*)

Bases: `Generic[torch.utils.data.dataset.T_co]`

Loads DeepVoxels data from a given directory into a *Dataset* object.

Please refer to the following paper for more information: https://arxiv.org/abs/1812.01024

Project page: https://vsitzmann.github.io/deepvoxels

Note: This implementation is based on the code from: https://github.com/bmild/nerf

Usage:

```
mode = "train"
scene = "bus"
dataset = DeepVoxels(data_directory, mode, scene)
```

Parameters

- **data_directory** (*str*) – The dataset base directory (see BaseDataset).
- **mode** (*str*) – The dataset usage mode (see BaseDataset).
- **scene** (*str*) – The scene name ("armchair", "bus", "cube"...).

- **download** (*bool*) – Flag indicating whether to automatically download the dataset (True) or not (False).

dataset_md5 = 'd715b810f1a6c2a71187e3235b2c5c56'

dataset_url = 'https://drive.google.com/u/0/uc?id=1lUvJWB6oFtT8EQ_NzBrXnmi25BufxRfl'

class pynif3d.datasets.**LLFF**(*data_directory*, *mode*, *scene*, *factor=8*, *recenter=True*, *bd_factor=0.75*, *spherify=False*, *path_z_flat=False*, *download=False*)

Bases: `Generic[torch.utils.data.dataset.T_co]`

Loads LLFF data from a given directory into a *Dataset* object.

Please refer to the following paper for more information: https://arxiv.org/abs/1905.00889

Note: This implementation is based on the code from: https://github.com/bmild/nerf

Usage:

```
mode = "train"
scan_id = "bus"
dataset = LLFF(data_directory, mode, scan_id)
```

Parameters

- **data_directory** (*str*) – The dataset base directory (see BaseDataset).
- **mode** (*str*) – The dataset usage mode (see BaseDataset).
- **scene** (*str*) – The scene name ("armchair", "bus", "cube"…).
- **factor** (*float*) – The factor to reduce image size by. Default is 8.
- **recenter** (*bool*) – Boolean flag indicating whether to re-center poses (True) or not (False). Default is True.
- **bd_factor** (*float*) – The factor to rescale poses by. Default is 0.75.
- **spherify** (*bool*) – Boolean flag indicating whether the poses should be converted to spherical coordinates (True) or not (False). Default is False.
- **path_z_flat** (*bool*) – (TODO: Add explanation). Defaults to False.
- **download** (*bool*) – Flag indicating whether to automatically download the dataset (True) or not (False).

dataset_md5 = '74cc8bd336e9a19fce3c03f4a1614c2d'

dataset_url = 'https://drive.google.com/u/0/uc?id=16VnMcF1KJYxN9QId6TClMsZRahHNMW5g'

class pynif3d.datasets.**Shapes3dDataset**(*data_directory*, *mode*, *download=False*, ***kwargs*)

Bases: `Generic[torch.utils.data.dataset.T_co]`

Loads ShapeNet and Synthetic Indoor Scene data from a given directory into a *Dataset* object.

Please refer to the Convolutional Occupancy Networks (CON) paper for more information: https://arxiv.org/abs/2003.04618

Note: This implementation is based on the original one, which can be found here: https://github.com/autonomousvision/convolutional_occupancy_networks

Usage:

```
mode = "train"
dataset = Shapes3dDataset(data_directory, mode)
```

Parameters

- **data_directory** (*str*) – The parent dictionary of the dataset.
- **mode** (*str*) – The subset of the dataset. Has to be one of ("train", "val", test").
- **download** (*bool*) – Flag indicating whether to automatically download the dataset (True) or not (False).
- **kwargs** (*dict*) –
  - **categories** (list): List of strings defining the object categories. Default is None.
  - **points_filename** (str): The name for the points file. Default is "points.npz".
  - **pointcloud_filename** (str): The name for the pointcloud file. Default is "pointcloud.npz".
  - **unpackbits** (bool): Boolean flag which defines if bit unpacking is needed during point cloud and occupancy loading. Default is True.
  - **gt_point_sample_count** (uint): The number of the point samples used as ground truth. Default is 2048.
  - **in_points_sample_count** (uint): The number of the point samples used as input to the network. Default is 3000.
  - **in_points_noise_stddev** (float): The stddev for noise to add to input points. Setting it to 0 will cancel noise addition. Default is 0.005.

2.5 pynif3d.encoding

class pynif3d.encoding.**FourierEncoding**(*input_dimensions=3*, *output_dimensions=256*, *scale=10*)

Bases: `torch.nn.modules.module.Module`

The implementation of the paper "Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains". This class encodes input N-dimensional coordinates into M-dimensional gaussian distribution and applies trigonometric encoding.

For more details, please check https://arxiv.org/abs/2006.10739.

Usage:

```
encoder = FourierEncoding()
encoded_points = encoder(points)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Parameters **x** (*torch.Tensor*) – Input tensor. Its shape is `(number_of_samples, input_dimensions)`.

Returns

Tensor with shape `(number_of_samples, 2 * output_dimensions)`.

Return type torch.Tensor

`get_dimensions()`

`training: bool`

`class pynif3d.encoding.PositionalEncoding(`*is_include_input: bool = True*, *input_dimensions: int = 3*, *max_frequency: int = 9*, *num_frequency: int = 10*, *frequency_factor: float = 1.0*, *is_log_sampling: bool = True*, *periodic_functions: list = None*`)`

Bases: `torch.nn.modules.module.Module`

The positional encoding class. It defines several frequency bands and applies several frequency responses to the input signal.

Usage:

```
encoder = PositionalEncoding()
encoded_points = encoder(points)
```

Parameters

- **is_include_input** (*bool*) – Boolean flags indicating whether to include the input signal in the output (True) or not (False). Default is True.
- **input_dimensions** (*int*) – The dimension of the input signal. Default is 3.
- **max_frequency** (*int*) – The upper limit of the frequency band. The upper limit will be set to *2^max_frequency*. Default is 9.
- **num_frequency** (*int*) – The number of frequency samples within the spectrum. Default is 10.
- **frequency_factor** (*float*) – The factor to multiply the frequency samples by. Default value is 1.0.
- **is_log_sampling** (*bool*) – Boolean flag indicating whether sampling shall be done in log spectrum (True) or not (False). Default is True.
- **periodic_functions** (*list*) – The periodic functions to be applied per frequency. Default is `[sin, cos]`.

`forward(`*x*`)`

Parameters **x** (*torch.Tensor*) – Input tensor. Its shape is `(number_of_samples, sample_dimension)`.

Returns Tensor with shape `(number_of_samples, output_dimensions)`. *output_dimensions* can be obtained by calling the *get_dimensions* method.

Return type torch.Tensor

`get_dimensions()`

`training: bool`

2.6 pynif3d.io

2.7 pynif3d.log

2.8 pynif3d.loss

`pynif3d.loss.`**`eikonal_loss`**`(`*x*`)`

Computes the eikonal loss for a given set of points.

Parameters **`x`** (*`torch.Tensor`*) – Tensor containing the point coordinates. Its shape is `(batch_size, n_samples, 3)` or `(n_samples, 3)`.

Returns Tensor containing the eikonal loss. Its shape is `(1,)`.

Return type torch.Tensor

`pynif3d.loss.`**`mse_to_psnr`**`(`*mse_loss*, *max_intensity=1.0*`)`

Converts a mean-squared error (MSE) loss to peak signal-to-noise ratio (PSNR).

Parameters

- **`mse_loss`** (*`torch.Tensor`*) – MSE loss. Its shape is `(1,)`.
- **`max_intensity`** (*`float`*) – The maximum pixel intensity. Default value is 1.0.

Returns Tensor with the corresponding PSNR loss.

Return type torch.Tensor

2.9 pynif3d.models

`class` `pynif3d.models.`**`ConvolutionalOccupancyNetworksModel`**`(`*input_channels=3*, *output_channels=1*, *block_depth=5*, *block_channels=256*, *linear_channels=128*, *is_linear_active=True*, *encoding_fn=None*`)`

Bases: `torch.nn.modules.module.Module`

The model for Convolutional Occupancy Networks (CON) as described in: https://arxiv.org/abs/2003.04618

Note: This implementation is based on the original one, which can be found at: https://github.com/autonomousvision/convolutional_occupancy_networks

This class is the neural implicit function (NIF) model of CON. It takes the query points as input, along with optional plane or grid features at query locations and outputs the occupancy probability of the input point. If *encoding_fn* is provided, the input points will be processed with *encoding_fn* before being supplied to model.

Usage:

```
model = ConvolutionalOccupancyNetworksModel()
occupancies = model(query_points, query_features)
```

Parameters

- **input_channels** (*int*) – The input layer's channel size. If *encoding_fn* is provided, this value will be overridden by the *encoding_fn.get_dimensions()* function. Default is 3.
- **output_channels** (*int*) – The output channel size. Default is 1.
- **block_depth** (*int*) – The number of resnet blocks connected sequentially. Default is 5.
- **block_channels** (*int*) – The channel size of each Fully-Connected ResNet block. Default is 256.
- **linear_channels** (*int*) – The channel size for the linear layers that bind plane features to Fully-Connected ResNet blocks. This value shall be equal to the input feature's channel dimensions. Default is 128.
- **is_linear_active** (*bool*) – Boolean flag indicating whether linear layers are enabled for the plane features. If True, *query_features* shall be provided during inference. Default is True.
- **encoding_fn** – The function instance that is called in order to apply encoding to input point coordinates. It has to contain callable *get_dimensions* property which returns the resulting dimensions. Default is None.

forward(*query_points*, *query_features=None*)

Parameters

- **query_points** (*torch.Tensor*) – The points to provide as input to the network. Its shape is `(batch_size, n_points, input_channels)`.
- **query_features** (*torch.Tensor*) – The plane or grid features related to *query_points* locations. Its shape is `(batch_size, n_points, linear_channels)`. Optional.

Returns Tensor which holds the occupancy probabilities of query locations. Its shape is `(batch_size, n_points)`.

Return type torch.Tensor

training: bool

class pynif3d.models.**IDRNIFModel**(*input_channels=3*, *output_channels=257*, *base_network_depth=8*, *base_network_channels=512*, *skip_layers=None*, *encoding_fn=None*, *is_encoding_active=True*, *normalize_weights=True*, *geometric_init=True*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

The multi-layer MLP model for NIF representation. If provided, it applies positional encoding to the inputs and overrides the input channel information accordingly.

Note: Please check the paper for more information: https://arxiv.org/abs/2003.09852

Usage:

```
model = IDRNIFModel()
pred_dict = model(points)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*points*)

Parameters **points** (*torch.Tensor*) – Tensor containing the points that are processed. Its shape is `(batch_size, n_rays, 3)` or `(n_rays, 3)`.

Returns Dictionary containing the computed SDF values (as torch.Tensor of shape `(*points.shape[:-1])`) and feature vectors (as torch.Tensor of shape `(*points.shape[:-1], output_channels - 1)`).

Return type dict

init_geometric(*size_in*, *size_out*, *layer_index*, *n_layers*)

training: bool

class pynif3d.models.**IDRRenderingModel**(*input_channel_points=3*, *input_channel_view_dirs=3*, *input_channel_normals=3*, *input_channel_features=256*, *output_channels=3*, *base_network_depth=4*, *base_network_channels=512*, *is_use_view_directions=True*, *is_use_normals=True*, *encoding_viewdir_fn=None*, *is_input_encoding_active=True*)

Bases: *pynif3d.models.nerf_model.NeRFModel*

The multi-layer MLP model for IDR rendering. If provided, it applies positional encoding to the view directions and overrides the input channel information accordingly.

Note: Please check the paper for more information: https://arxiv.org/abs/2003.09852

Usage:

```
model = IDRRenderingModel()
rgb_values = model(points, features, normals, view_dirs)
```

Parameters

- **input_channels** (*int*) – The input channel dimension to the model. If positional encoding is used, this value will be overridden. Default is 3 (XYZ).
- **input_channel_view_dirs** (*int*) – The input channel dimension for viewing directions. If positional encoding is used, this value will be overridden. Default is 3 (XYZ).
- **input_channel_normals** (*int*) – The input channel dimension for surface normals. Default is 3 (XYZ).
- **input_channel_features** (*int*) – The input channel dimension for the extracted features. Default value is 256.
- **output_channels** (*int*) – The output channel dimension. Default is 4 (RGBA).
- **base_network_depth** (*int*) – The depth of the network MLP layers. One linear layer will be added to the base network for each increment. Default is 4.
- **base_network_channels** (*int*) – The output dimension of each inner linear layers of the MLP model. A positive integer value is expected. Default is 512.
- **is_use_view_directions** (*bool*) – Boolean flag indicating whether to use view direction (True) or not (False). If True, the view direction block will be added on top of the base MLP layers. Default is True.
- **is_use_normals** (*bool*) – Boolean flag indicating whether to use surface normals (True) or not (False). Default is True.

- **encoding_viewdir_fn** – The function that is called in order to apply encoding to the view directions input. Default is *PositionalEncoding()*.
- **is_input_encoding_active** (*bool*) – Boolean flag indicating whether encoding shall be applied to both the base network input and view directions. Default is True.

forward(*points*, *features*, *normals=None*, *view_dirs=None*)

Parameters

- **points** (*torch.Tensor*) – Tensor containing the points that are processed. Its shape is `(batch_size, n_rays, input_channel_points)` or `(n_rays, input_channel_points)`.
- **features** (*torch.Tensor*) – Tensor containing the features that are processed. Its shape is `(batch_size, n_rays, input_channel_features)` or `(n_rays, input_channel_features)`.
- **normals** (*torch.Tensor*) – (Optional) Tensor containing the normals that are processed. Its shape is `(batch_size, n_rays, input_channel_normals)` or `(n_rays, input_channel_normals)`.
- **view_dirs** (*torch.Tensor*) – (Optional) Tensor containing the view directions that are processed. Its shape is `(batch_size, n_rays, input_channel_view_dirs)` or `(n_rays, input_channel_view_dirs)`.

Returns Tensor containing the rendered RGB values. Its shape is `(*points.shape[:-1], 3)`.

Return type torch.Tensor

training: bool

class pynif3d.models.**NeRFModel**(*input_channels=3*, *input_channel_view_dirs=3*, *output_channels=4*, *base_network_depth=8*, *base_network_channels=256*, *skip_layers=None*, *is_use_view_directions=True*, *view_dir_network_depth=1*, *view_dir_network_channels=256*, *encoding_fn=None*, *encoding_viewdir_fn=None*, *is_input_encoding_active=True*, *init_kwargs=None*, *normalize_weights=False*)

Bases: `torch.nn.modules.module.Module`

The multi-layer MLP model for NeRF rendering. If provided, it applies positional encoding to the inputs overrides the input channel information accordingly. It can also integrate view direction information into the network.

Usage:

```
model = NeRF()
prediction = model(points, view_dirs)
```

Parameters

- **input_channels** (*int*) – The input channel dimension to the model. If positional encoding is used, this value will be overridden. Default is 3 (XYZ).
- **input_channel_view_dirs** (*int*) – The input channel dimension for viewing directions. If positional encoding is used, this value will be overridden. Default is 3 (XYZ).
- **output_channels** (*int*) – The output channel dimension. Default is 4 (RGBA).
- **base_network_depth** (*int*) – The depth of the network MLP layers. One linear layer will be added to the base network for each increment. Default is 8.

- **base_network_channels** (*int*) – The output dimension of each inner linear layers of the MLP model. A positive integer value is expected. Default is 256.
- **skip_layers** (`Iterable`) – The layers to add skip connection. It shall be an iterable of positive integers. Values larger than *network_depth* will be discarded. Default is [4,].
- **is_use_view_directions** (*bool*) – Boolean flag indicating whether to use view direction (True) or not (False). If True, the view direction block will be added on top of the base MLP layers. Default is True.
- **view_dir_network_depth** (*int*) – The depth of the network that processes view directions. One linear layer for processing view direction will be added to the network for each increment. Default value is 1.
- **view_dir_network_channels** (*int*) – The output dimension of each inner linear layers of the MLP model which processes view directions. A positive integer is expected. Default value is 256.
- **encoding_fn** (*torch.nn.Module*) – The function that is called in order to apply encoding to the NIF model input. Default is *PositionalEncoding()*.
- **encoding_viewdir_fn** (*torch.nn.Module*) – The function that is called in order to apply encoding to the view directions input. Default is *PositionalEncoding()*.
- **is_input_encoding_active** (*bool*) – Boolean flag indicating whether encoding shall be applied to both the base network input and view directions. Default is True.
- **init_kwargs** (*dict*) – Dictionary containing the initialization parameters for the linear layers.
- **normalize_weights** (*bool*) – Boolean flag indicating whether to normalize the linear layer's weights (True) or not (False).

forward(*query_points*, *view_dirs=None*)

Parameters

- **query_points** (*torch.Tensor*) – Tensor containing the points to that are queried. Its shape is `(number_of_rays, number_of_points_per_ray, point_dims)`.
- **view_dirs** (*torch.Tensor*) – (Optional) Tensor containing the view directions. Its shape is `(number_of_rays, number_of_points_per_ray, point_dims)`.

Returns Tensor containing the prediction result of the model. Its shape is `(n_samples, n_rays, output_channel)`.

Return type torch.Tensor

training: bool

class pynif3d.models.**PixelNeRFNIFModel**(*input_channel_points: int = 3*, *input_channel_view_dirs: int = 3*, *output_channels: int = 4*, *hidden_channels: int = 128*, *is_use_view_directions: bool = True*, *is_point_encoding_active: bool = True*, *is_view_encoding_active: bool = False*, *encoding_fn: torch.nn.modules.module.Module = None*, *encoding_viewdir_fn: torch.nn.modules.module.Module = None*, *n_resnet_blocks: int = 5*, *reduce_block_index: int = 3*, *activation_fn: torch.nn.modules.module.Module = None*, *init_kwargs: dict = None*)

Bases: `torch.nn.modules.module.Module`

The multi-layer MLP model for PixelNeRF rendering. If provided, it applies positional encoding to the inputs overrides the input channel information accordingly. It can also integrate view direction information into the network.

Parameters

- **input_channel_points** (*int*) – The input channel dimension to the model. If positional encoding is used, this value will be overridden. Default is 3 (XYZ).
- **input_channel_view_dirs** (*int*) – The input channel dimension for viewing directions. If positional encoding is used, this value will be overridden. Default is 3 (XYZ).
- **output_channels** (*int*) – The output channel dimension. Default is 4 (RGBA).
- **hidden_channels** (*int*) – The number of hidden channels contained within each ResNetBlockFC block. Default is 128.
- **is_use_view_directions** (*bool*) – Boolean flag indicating whether to use view direction (True) or not (False). If True, the view direction block will be added on top of the base MLP layers. Default is True.
- **is_point_encoding_active** (*bool*) – Boolean flag indicating whether encoding shall be applied to the input points. Default is True.
- **is_view_encoding_active** (*bool*) – Boolean flag indicating whether encoding shall be applied to the viewing directions. Default is False.
- **encoding_fn** (*torch.nn.Module*) – The function that is called in order to apply encoding to the NIF model input. Default is *PositionalEncoding()*.
- **encoding_viewdir_fn** (*torch.nn.Module*) – The function that is called in order to apply encoding to the view directions input. Default is *PositionalEncoding()*.
- **n_resnet_blocks** (*int*) – The number of ResNetBlockFC blocks that are contained within the base network. Default value is 5.
- **reduce_block_index** (*int*) – The index of the ResNetBlockFC block at which the reduce operation is going to be applied (along the dimension that is related to the number of objects). Default value is 3.
- **activation_fn** (*torch.nn.Module*) – The activation function. Default is ReLU.
- **init_kwargs** (*dict*) – Dictionary containing the initialization parameters for the linear layers.

`forward`(*ray_points: torch.Tensor*, *camera_poses: torch.Tensor*, *features: torch.Tensor*, *view_dirs: Optional[torch.Tensor] = None*, ***kwargs: dict*) → torch.Tensor

Parameters

- **ray_points** (*torch.Tensor*) – Tensor containing the ray points to that are going to be processed. Its shape is `(n_ray_samples, input_channel_points)`.
- **camera_poses** (*torch.Tensor*) – Tensor containing the camera poses. Its shape is `(n_objects, 3, 4)`.
- **features** (*torch.Tensor*) – (Optional) Tensor containing the feature vectors. Its shape is `(n_views, n_ray_samples, feature_size)`.
- **view_dirs** (*torch.Tensor*) – (Optional) Tensor containing the view directions. Its shape is `(n_ray_samples, input_channel_points)`.
- **kwargs** (*dict*) –

- **reduce** (str): The reduce operation that is applied to the ResNetBlockFC block at *reduce_block_index*. Currently supported options are "average" and "max".

Returns

Tensor containing the prediction result of the model. Its shape is `(n_ray_samples, output_channel)`.

Return type torch.Tensor

training: bool

class pynif3d.models.**PointNet_LocalPool**(*input_channels=3*, *point_network_depth=5*, *point_feature_channels=128*, *scatter_type='max'*, *feature_grids=None*, *feature_processing_fn=None*, *feature_grid_resolution=32*, *feature_grid_channels=128*, *padding=0.1*, *encoding_fn=None*)

Bases: `torch.nn.modules.module.Module`

The point encoder model for Convolutional Occupancy Networks (CON) as described in: https://arxiv.org/abs/2003.04618

Note: This implementation is based on the original one, which can be found at: https://github.com/autonomousvision/convolutional_occupancy_networks

PointNet-based encoder network with ResNet blocks for each point. It takes the input points, applies a variation of PointNet and projects each point to defined plane(s). It returns the plane features. The number of input points is fixed.

Usage:

```
plane = "xz"
plane_resolution = 256
feature_channels = 32

model = PointNet_LocalPool(
    feature_grids=[plane],
    feature_grid_resolution=plane_resolution,
    feature_grid_channels=feature_channels,
)

features = model(points)
```

Parameters

- **input_channels** (*int*) – The input layer's channel size. If *encoding_fn* is provided, this value will be overridden by *encoding_fn.get_dimensions()*. Default is 3.
- **point_network_depth** (*int*) – The number of resnet blocks that are connected sequentially. Default is 5.
- **point_feature_channels** (*int*) – The channel size of each Fully-Connected ResNet blocks. Default is 128.
- **scatter_type** (*str*) – The type of the scattering operation. Options are ("mean", "max"). Default is "max".

- **feature_grids** (*iterable*) – The iterable object to define the planes of the points to be projected. The options are ("xy", "yz", "xz", "grid"). "grid" cannot be used in combination with other options. Default is ["xz"].
- **feature_processing_fn** (*instance*) – (Optional) The model that processes point features projected to 2D planes. The instance of the pre-initialized model has to be provided. If not provided, the 2D plane processing step will be skipped.
- **feature_grid_resolution** (*int*) – The resolution of the 2D planes that the points are projected to. It has to be a positive integer. Only square planes are supported for now. Default is 32.
- **feature_grid_channels** (*int*) – The channel size of the 2D plane features. It has to be same as the input channel dimensions of the model provided through *plane_processing_fn*. Default is 128.
- **padding** (*float*) – Padding variable used during the normalization operations. Assign to 0 to cancel any padding. Default is 0.1.
- **encoding_fn** (*instance*) – The function instance that applies encoding to input point coordinates. It has to contain the callable *get_dimensions* property which returns the resulting dimensions. Default is None.

forward(*input_points*)

Parameters **input_points** (*torch.Tensor*) – Tensor containing the points that are provided as input to the network. Its shape is `(batch_size, n_points, input_channels)`.

Returns Dictionary containing the tensors for all the features of the planes.

Return type dict

generate_coordinate_features(*p*, *c*, *feature_grid='xz'*)

Scatters the given features (c) based on given coordinates (p) by using the grid resolution. This is the orthographic point-to-plane projection function.

Parameters

- **p** (*torch.Tensor*) – Tensor containing the locations of the points. Its shape is `(batch_size, number_of_points, 3)`.
- **c** (*torch.Tensor*) – Tensor containing the point features. Its shape is `(batch_size, number_of_points, feature_dimensions)`.

Returns Tensor containing the scattered features of the points. Its shape is `(batch_size, feature_dimensions, grid_resolution, grid_resolution)`.

Return type torch.Tensor

pool_local(*keys*, *indices*, *features*)

Applies the max pooling operation to the point features, based on the grid resolution. After pooling, the points within the same pooling region are to the same features.

Parameters

- **keys** (*list*) – List containing the plane IDs. It is expected that such indices exist in `indices`.
- **indices** (*dict*) – Dictionary containing *(plane_id, point_indices)* pairs mapping each point's 1D index to a 2D plane. *point_indices* is a *torch.Tensor* with shape `(batch_size, 1, point_count)`.

- **features** (*torch.Tensor*) – Tensor containing the point features. Its shape is `(batch_size, point_count, feature_size)`.

Returns Tensor with shape `(batch_size, point_count, feature_size)`.

Return type torch.Tensor

training: bool

class pynif3d.models.**ResnetBlockFC**(*size_in: int*, *size_out: int*, *size_inner: int*, *activation_fn: torch.nn.modules.module.Module = None*, *init_fc_0_kwargs: dict = None*, *init_fc_1_kwargs: dict = None*, *init_fc_s_kwargs: dict = None*)

Bases: `torch.nn.modules.module.Module`

Implementation of the fully-connected ResNet block for Convolutional Occupancy Networks (CON), as described in: https://arxiv.org/abs/2003.04618

Note: This implementation is based on the original one, which can be found at: https://github.com/autonomousvision/convolutional_occupancy_networks

It replaces convolutional layers of vanilla ResNet blocks with linear layers.

Usage:

```
input_channels = 32
hidden_channels = 32
output_channels = 32

model = ResnetBlockFC(input_channels, output_channels, hidden_channels)
features = model(x)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Parameters **x** (*torch.Tensor*) – Tensor with shape `(batch_size, n_points, size_in)`.

Returns Tensor with shape `(batch_size, n_points, size_out)`.

Return type torch.Tensor

training: bool

class pynif3d.models.**SpatialEncoder**(*backbone_fn: torch.nn.modules.module.Module = None*, *backbone_fn_kwargs: dict = None*, *n_layers: int = 4*, *pretrained: bool = True*)

Bases: `torch.nn.modules.module.Module`

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*images*, ***kwargs*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class pynif3d.models.**UNet**(*output_channels*, *input_channels=3*, *network_depth=5*, *first_layer_channels=64*, *upconv_mode='transpose'*, *merge_mode='concat'*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

UNet class for Convolutional Occupancy Networks (CON), as described in: https://arxiv.org/abs/2003.04618

Note: This implementation is based on the original one, which can be found at: https://github.com/autonomousvision/convolutional_occupancy_networks

The U-Net is a convolutional encoder-decoder neural network. Contextual spatial information (from the decoding, expansive pathway) related to the input tensor is merged with information representing the localization of details (from the encoding, compressive pathway).

Usage:

```
input_channels = 3
output_channels = 1

model = UNet(output_channels, input_channels)
h = model(h)
```

Parameters

- **output_channels** (*int*) – The number of channels for the output tensor.
- **input_channels** (*int*) – The number of channels in the input tensor. Default is 3.
- **network_depth** (*int*) – The number of convolution blocks. Default is 5.
- **first_layer_channels** (*int*) – The number of convolutional filters for the first convolution. For each depth level, the channel size is multiplied by 2.
- **up_mode** (*str*) – The type of upconvolution ("transpose", "upsample").
- **merge_mode** (*str*) – The type of the merge operation ("concat", "add").
- **kwargs** (*dict*) –
  - **w_init_fn**: Callback function for parameter initialization. Default is *xavier_normal*.
  - **w_init_fn_args**: The arguments to pass to the *w_init_fn* function. Optional.
  - **b_init_fn**: Callback function for bias initialization. Default is constant 0.
  - **b_init_fn_args**: The arguments to pass the *b_init_fn* function. Optional.

forward(*h*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class pynif3d.models.**UNet3D**(*output_channels*, *input_channels*, *final_sigmoid=True*, *feature_maps=64*, *num_groups=8*, *num_levels=4*, *encoder_pool_type='max'*, *is_segmentation=False*, *testing=False*)

Bases: `torch.nn.modules.module.Module`

3D Unet class. It applies encoder and decoder as 3D U-Net.

Usage:

```
input_channels = 16
output_channels = 32

model = UNet3D(output_channels, input_channels)
features = model(x)
```

Parameters

- **output_channels** (*int*) – number of output segmentation masks; note that the value of out_channels might correspond to either different semantic classes or to different binary segmentation mask. It's up to the user of the class to interpret the out_channels and use the proper loss criterion during training (i.e. CrossEntropyLoss (multi-class) or BCEWithLogitsLoss (two-class) respectively).
- **input_channels** (*int*) – The number of input channels
- **final_sigmoid** (*bool*) – if True apply element-wise nn. Sigmoid after the final 1x1 convolution, otherwise apply nn.Softmax. MUST be True if nn.BCELoss (two-class) is used to train the model. MUST be False if nn.CrossEntropyLoss (multi-class) is used to train the model.
- **feature_maps** (*int, tuple*) – if int: number of feature maps in the first conv layer of the encoder (default: 64); if tuple: number of feature maps at each level
- **num_groups** (*int*) – number of groups for the GroupNorm
- **num_levels** (*int*) – number of levels in the encoder/decoder path (applied only if f_maps is an int)
- **is_segmentation** (*bool*) – if True (semantic segmentation problem) Sigmoid/Softmax normalization is applied after the final convolution; if False (regression problem) the normalization layer is skipped at the end
- **testing** (*bool*) – if True (testing mode) the *final_activation* (if present, i.e. *is_segmentation=true*) will be applied as the last operation during the forward pass; if False the model is in training mode and the *final_activation* (even if present) won't be applied; default: False

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

2.10 pynif3d.pipeline

class `pynif3d.pipeline.`**`BasePipeline`**

Bases: `torch.nn.modules.module.Module`

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`load_pretrained_model`(*yaml_file*, *model_name*, *cache_directory='.'*)

`training: bool`

class `pynif3d.pipeline.`**`ConvolutionalOccupancyNetworks`**(*encoder_fn=None*, *feature_sampler_fn=None*, *nif_model=None*, *rendering_fn=None*, *pretrained=None*)

Bases: *`pynif3d.pipeline.base_pipeline.BasePipeline`*

This is the main pipeline function for the Convolutional Occupancy Networks: https://arxiv.org/abs/2003.04618

This class takes the noisy point cloud, applies an encoding function (i.e. PointNet) to extract features from inputs, projects the points to 2D plane(s) or 3D grid, optionally applies an auto-encoder network in 2D or 3D planes to generate features. For each input query point, bilinear/trilinear sampling on feature plane(s) or grid is applied in order to extract the features query point features. By applying a shallow neural implicit function model, the occupancy probability of each input query point is predicted.

This class takes an encoder, feature sampler, NIF model and rendering functions during initialization as input, in order to define the pipeline.

Usage:

```
model = ConvolutionalOccupancyNetworks()
occupancies = model(input_points, query_points)
```

Parameters

- **`encoder_fn`** (*`instance`*) – The function instance that is called in order to encode the input points. Default is *PointNet_LocalPool*.
- **`feature_sampler_fn`** (*`instance`*) – The function instance that is called in order to sample the features on a plane or on a grid. The sampler has to match the 2D/3D operation mode. Default is *PlaneFeatureSampler*.
- **`nif_model`** (*`instance`*) – The model instance that outputs occupancy information given some query points and sampled features. Default is *ConvolutionalOccupancyNetworksModel*.
- **`rendering_fn`** (*`instance`*) – The function instance that is called in order to render the query points obtained using the *nif_model*. Default is *PointRenderer*.
- **`pretrained`** (*`str`*) – (Optional) The pretrained configuration to load model weights from. Default is None.

`forward`(*input_points*, *query_points*)

Parameters

- **`input_points`** (*`torch.Tensor`*) – Tensor that holds noisy input points. Its shape is `(batch_size, n_points, point_dimension)`.
- **`query_points`** (*`torch.Tensor`*) – Tensor containing the queried occupancy locations. Its shape is `(batch_size, n_points, point_dimension)`.

Returns Tensor containing occupancy probabilities. Its shape is `(batch_size, n_points)`.

Return type torch.Tensor

training: bool

class `pynif3d.pipeline.IDR`(*image_size*, *n_rays_per_image=2048*, *input_sampler_training=None*, *input_sampler_inference=None*, *nif_model=None*, *rendering_fn=None*)

Bases: `torch.nn.modules.module.Module`

This is the main pipeline function for the Implicit Differentiable Renderer (IDR) algorithm: https://arxiv.org/abs/2003.09852

It takes an image, object mask, intrinsic parameters and camera pose as input and returns the reconstructed 3D points, the rendered pixel values and the predicted mask, given the input pose. During training it also returns the predicted Z values of the sampled points, along with the value of *gradient_theta*, used in the computation of the eikonal loss.

Usage: .. code-block:: python

image_size = (image_height, image_width) model = IDR(image_size) pred_dict = model(image, object_mask, intrinsics, camera_poses)

Parameters

- **image_size** (*tuple*) – Tuple containing the image size, expressed as `(image_height, image_width)`.
- **n_rays_per_image** (*int*) – The number of rays to be sampled for each image. Default value is 2048.
- **input_sampler_training** (*torch.nn.Module*) – The ray sampler to be used during training. If set to None, it will default to RandomPixelSampler. Default value is None.
- **input_sampler_inference** (*torch.nn.Module*) – The ray sampler to be used during inference. If set to None, it will default to AllPixelSampler. Default value is None.
- **nif_model** (*torch.nn.Module*) – NIF model for outputting the prediction. If set to None, it will default to IDRNIFModel. Default value is None.
- **rendering_fn** (*torch.nn.Module*) – The rendering function to be used during both training and inference. If set to None, it will default to IDRRenderingModel. Default value is None.

compute_gradient(*points*)

compute_rgb_values(*points*, *view_dirs*)

forward(*image*, *object_mask*, *intrinsics*, *camera_poses*, ***kwargs*)

Parameters

- **image** (*torch.Tensor*) – Tensor containing the input images. Its shape is `(batch_size, 3, image_height, image_width)`.
- **object_mask** (*torch.Tensor*) – Tensor containing the object masks. Its shape is `(batch_size, 1, image_height, image_width)`.
- **intrinsics** (*torch.Tensor*) – Tensor containing the camera intrinsics. Its shape is `(batch_size, 4, 4)`.
- **camera_poses** (*torch.Tensor*) – Tensor containing the camera poses. Its shape is `(batch_size, 4, 4)`.

- **kwargs** (*dict*) –
  - **chunk_size** (int): The chunk size of the tensor that is passed for NIF prediction.

Returns Dictionary containing the prediction outputs: the 3D coordinates of the intersection points + corresponding RGB values + the ray-to-surface intersection mask (used in training and inference) and Z values + gradient theta + sampled 3D coordinates (used in training only).

Return type dict

`training: bool`

`class pynif3d.pipeline.NeRF`(*image_size*, *focal_length*, *n_rays_per_image=1024*, *n_points_per_chunk=1024*, *input_sampler_training=None*, *input_sampler_inference=None*, *background_color=None*, *ray_generator=None*, *ray_samplers=None*, *n_points_per_ray=None*, *level_of_sampling=2*, *near=2*, *far=6*, *nif_models=None*, *rendering_fn=None*, *aggregation_fn=None*, *pretrained=None*)

Bases: `pynif3d.pipeline.base_pipeline.BasePipeline`

This is the main pipeline function for the Neural Radiance Fields (NeRF) algorithm: https://arxiv.org/abs/2003.08934

It takes a camera pose as input and returns the rendered pixel values given the input pose.

Usage:

```
image_size = (image_height, image_width)
focal_length = (focal_x, focal_y)

model = NeRF(image_size, focal_length)
pred_dict = model(camera_pose)
```

Parameters

- **image_size** (*list, tuple*) – List or tuple containing the spatial image size (`height, width`). Its shape is `(2,)`.
- **focal_length** (*list, tuple*) – List or tuple containing the camera's focal length (`focal_x, focal_y)`). Its shape is `(2,)`.
- **n_rays_per_image** (*int*) – The number of ray samples that are extracted from an image and processed. Default is *1024*. Optional.
- **n_points_per_chunk** – The number of sampled points passed to the NIF model at once.
- **input_sampler_training** (*instance*) – (Optional) The pixel sampling function used during training. Default is *RandomPixelSampler*.
- **input_sampler_inference** (*instance*) – (Optional) The pixel sampling function used during inference. Default is *AllPixelSampler*.
- **ray_generator** (*instance*) – (Optional) The function that is called in order to generate rays with respect to a given camera pose. Default is *CameraRayGenerator*.
- **ray_samplers** (*list, tuple*) – (Optional) List or tuple of the same length as *level_of_sampling* containing the function(s) that define the sampling logic for each ray. Default is *UniformRaySampler* for the first level and *WeightedRaySampler* for the second level.

- **n_points_per_ray** (*list, tuple*) – (Optional) List or tuple with a length equal to *level_of_sampling* containing the number of points that are sampled across each ray. Default is 64 for each level.
- **level_of_sampling** (*list, tuple*) – (Optional) List or tuple containing the levels of fine samples. Default value is 2 to follow coarse/fine pattern in the original NeRF paper.
- **near** (*float*) – (Optional) The boundary value for each sampled ray. Each ray will be sampled between [*near*, *far*]. Default is 2.
- **far** (*float*) – (Optional) The boundary value for each sampled ray. Each ray will be sampled between [*near*, *far*]. Default is 6.
- **nif_models** (*list, tuple*) – (Optional) List or tuple with the length equal to *level_of_sampling* containing the models that define the neural implicit representation of the 3D scene. Default is *NeRFModel* for each level.
- **rendering_fn** (*instance*) – (Optional) The function that defines the NIF model execution logic, in order to obtain the resulting pixel values. Default is *PointRenderer*.
- **aggregation_fn** (*instance*) – (Optional) The function that defines the aggregation logic for the predicted 3D point values, in order to obtain the final pixel values. Default is *NeRFAggregator*.
- **pretrained** (*str*) – (Optional) The pretrained configuration to load model weights from. Default is None.

forward(*pose*)

Parameters **pose** (*torch.Tensor*) – Tensor containing the camera pose information, used for querying. Its shape is `(3, 4)`.

Returns Dictionary containing the rendering result (RGB, depth, disparity and transparency values for each pixel that is sampled by *input_sampler_inference* or *input_sampler_training*).

Return type dict

training: bool

2.11 pynif3d.renderer

class pynif3d.renderer.**PointRenderer**(*chunk_size=None*)

Bases: `torch.nn.modules.module.Module`

The function that is used for rendering each sampled point using the NIF model.

Usage:

```
# Assume that a NIF model (torch.nn.Module) is given.
renderer = PointRenderer(chunk_size=256)
prediction = renderer(nif_model)
```

Parameters **chunk_size** (*int*) – The chunk size of the tensor that is passed for NIF prediction.

forward(*nif_model*, **args*)

Parameters

- **nif_model** (*torch.nn.Module*) – NIF model for outputting the prediction.
- **args** (*list, tuple*) – Tuple or list containing the tensors that are passed through the NIF model.

Returns Tensor containing the concatenated predictions.

Return type torch.Tensor

training: bool

2.12 pynif3d.sampling

class pynif3d.sampling.**AllPixelSampler**(*height*, *width*)

Bases: `torch.nn.modules.module.Module`

The function that is used to sample all the elements of the given input 2D array. This function simply flattens a 2D array based on the [y, x] coordinates and returns each pixel as result.

Usage:

```
sampler = AllPixelSampler(image_height, image_width)
sampled_data = sampler(rays_directions=rays_d, rays_origins=rays_o)

rays_d = sampled_data["ray_directions"]
rays_o = sampled_data["ray_origins"]
```

Parameters

- **height** (*int*) – The height of the 2D array to be sampled. Positive integer.
- **width** (*int*) – The width of the 2D array to be sampled. Positive integer.

forward(*image=None*, ***kwargs*)

Parameters **image** (*torch.Tensor*) – (Optional) The input 2D array to be sampled. Its shape is `(1, 3, image_height, image_width)`.

Returns Dictionary containing the sampled colors (RGB values) and pixel coordinates. The sampled colors are represented as *torch.Tensor* with shape `(n_pixels, 3)`, while the sampled coordinates are represented as a *torch.Tensor* with shape `(n_pixels, 2)`.

Return type dict

training: bool

class pynif3d.sampling.**FeatureSampler2D**(*sample_mode='bilinear'*, *padding=0.1*)

Bases: `torch.nn.modules.module.Module`

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*points*, *plane_features*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class pynif3d.sampling.**FeatureSampler3D**(*sample_mode='bilinear'*, *padding=0.1*)
Bases: `torch.nn.modules.module.Module`

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*points*, *plane_features*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class pynif3d.sampling.**RandomPixelSampler**(*height*, *width*)
Bases: `torch.nn.modules.module.Module`

Randomly samples N elements from a given 2D array as input.

Parameters

- **height** (*int*) – Positive integer defining the height of the 2D array to be sampled.
- **width** (*int*) – Positive integer defining the width of the 2D array to be sampled.

Usage:

```
sampler = RandomPixelSampler(image_height, image_width)
sampled_data = sampler(rays_directions=rays_d, rays_origins=rays_o)

rays_d = sampled_data["ray_directions"]
rays_o = sampled_data["ray_origins"]
```

forward(*n_sample*, ***kwargs*)

Parameters

- **n_sample** (*int*) – Positive integer defining the number of samples to be queried.
- **kwargs** (*dict*) – The (key, value) pairs for multiple values to be sampled at once.

Returns Dictionary containing the same (key, value) pairs as ***kwargs*. It also contains the random sampling locations.

Return type dict

training: bool

class pynif3d.sampling.**SecantRaySampler**(*sdf_model*)
Bases: `torch.nn.modules.module.Module`

Samples the ray in a given range, computes the SDF values for the sampled points and runs the secant method for the rays which have sign transition. Returns the resulting points and their corresponding SDF values.

:: note:

> For more information about the secant method, please check the following page: https://en.wikipedia.org/wiki/Secant_method

Usage:

```
# Assume an SDF model (torch.nn.Module) is given.
sampler = SecantRaySampler(sdf_model)
points, z_vals, mask = sampler(
    ray_directions, ray_origins, ray_mask, zs_min, zs_max
)
```

Parameters `sdf_model` (*instance*) – Instance of an SDF model. When calling the *forward* method with some input points, it needs to return a dictionary containing the SDF values corresponding to those points, as an "sdf_vals" key/value pair.

`forward`(*rays_d*, *rays_o*, *rays_m*, *zs_min*, *zs_max*, ***kwargs*)

Parameters

- **rays_d** (*torch.Tensor*) – Tensor containing the ray directions. Its shape is `(n_rays, 3)`.
- **rays_o** (*torch.Tensor*) – Tensor containing the ray origins. Its shape is `(n_rays, 3)`.
- **rays_m** (*torch.Tensor*) – Boolean tensor containing the object mask for the given rays. If rays_d[i] intersects the object, rays_m[i] is marked as True, otherwise as False. Its shape is `(n_rays,)`.
- **zs_min** (*torch.Tensor*) – Tensor containing the minimum Z values of the points that are sampled along the ray. Its shape is `(n_rays,)`.
- **zs_max** (*torch.Tensor*) – Tensor containing the maximum Z values of the points that are sampled along the ray. Its shape is `(n_rays,)`.
- **kwargs** (*dict*) –
  - **n_samples** (int): The number of points that are sampled along the rays. Default value is 100.
  - **chunk_size** (int): The size of the chunk of points that is passed to the SDF model. Default value is 10000.

Returns Tuple containing the secant points (as a torch.Tensor with shape `(n_rays, 3)`), corresponding Z values (as a torch.Tensor with shape `(n_rays,)`) and a mask (as a torch.Tensor with shape `(n_rays,)`) specifying which points were successfully found by the secant method to be roots of the optimization function.

Return type tuple

`training: bool`

class pynif3d.sampling.**UniformRaySampler**(*near*, *far*, *n_samples*, *is_perturb=True*)

Bases: `torch.nn.modules.module.Module`

Randomly samples a ray. Takes as input a ray origin and direction, *near*, *far*, the number of points to sample and generates point coordinates across each given ray.

Usage:

```
near = 0.1
far = 5.0
n_samples = 1000

sampler = UniformRaySampler(near, far, n_samples)
points, z_vals = sampler(ray_directions, ray_origins)
```

Parameters

- **near** (*float*) – Minimum depth value corresponding to the sampled points.
- **far** (*float*) – Maximum depth value corresponding to the sampled points.
- **n_samples** (*int*) – Number of sampled points along the ray.
- **is_perturb** (*bool*) – Boolean flag indicating whether to perturb the sampled points (True) or not (False). Default value is True.

forward(*rays_d*, *rays_o*)

Parameters

- **rays_d** (*torch.Tensor*) – Tensor containing the ray directions. Its shape is `(n_ray_samples, 3)` or `(batch_size, n_ray_samples, 3)`.
- **rays_o** (*torch.Tensor*) – Tensor containing the ray origins. Its shape is `(n_ray_samples, 3)` or `(batch_size, n_ray_samples, 3)`.

Returns Tuple containing the sampled points and the corresponding Z values.

Return type tuple

training: bool

class pynif3d.sampling.**WeightedRaySampler**(*near*, *far*, *n_sample*, *eps=1e-05*)

Bases: `torch.nn.modules.module.Module`

Neural implicit model-based importance sampling function for rays which is used in the NeRF paper. For details, please check https://arxiv.org/abs/2003.08934.

Usage:

```
near = 0.1
far = 5.0
n_samples = 1000

sampler = WeightedRaySampler(near, far, n_samples)
points, z_vals = sampler(ray_directions, ray_origins, z_vals, weights)
```

Parameters

- **near** (*float*) – Minimum depth value corresponding to the sampled points.
- **far** (*float*) – Maximum depth value corresponding to the sampled points.
- **n_sample** (*int*) – Number of sampled points along the ray.
- **eps** (*float*) – Epsilon that is added to the weights, in order to avoid zero values. Default value is 1e-5.

`forward`(*rays_d*, *rays_o*, *z_vals*, *weights*, ***kwargs*)

Parameters

- **rays_d** (*torch.Tensor*) – Tensor containing ray directions. Its shape is `(batch_size, n_rays, 3)`.
- **rays_o** (*torch.Tensor*) – Tensor containing ray origins. Its shape is `(batch_size, n_rays, 3)`.
- **z_vals** (*torch.Tensor*) – Tensor containing Z values. Its shape is `(n_rays, n_samples_per_ray,)`.
- **weights** (*torch.Tensor*) – Tensor containing the sampling weights. Its shape is `(batch_size, n_rays, n_samples_per_ray)`.
- **kwargs** (*dict*) –
  - **is_deterministic** (bool): (Optional) Boolean flag indicating whether to sample the rays in a deterministic manner (True) or not (False). Default is False.

Returns Tuple containing the sampled points and Z values.

Return type tuple

`sample_pdf`(*bins*, *weights*, *is_deterministic=False*)

`training: bool`

2.13 pynif3d.utils

`pynif3d.utils.check_in_options`(*variable*, *options*, *variable_name*)

`pynif3d.utils.normalize`(*x*)

Normalizes an input vector.

Parameters **x** (*np.array*) – Array containing the vector's coordinates.

Returns Array containing the normalized coordinates.

Return type np.array

`pynif3d.utils.radians`(*theta_deg*)

Converts an angle from degrees to radians.

Parameters **theta_deg** (*float*) – Angle value in degrees.

Returns Angle value in radians.

Return type float

`pynif3d.utils.ray_sphere_intersection`(*rays_d*, *spheres_o*, *radius=1.0*)

Computes the intersection points between a given set of spheres placed at origins *spheres_o* and a given set of ray directions *rays_d*.

Parameters

- **rays_d** (*torch.Tensor*) – Tensor containing the ray directions. Its shape is `(batch_size, n_rays, 3)`.
- **spheres_o** (*torch.Tensor*) – Tensor containing the sphere origins. Its shape is `(batch_size, n_rays, 3)`.

- `radius` (*float*) – The radius of the spheres.

Returns

Tuple containing the two intersection velocities per ray (as a `(batch_size, n_rays, 2)` tensor) and a mask (as a `(batch_size, n_rays)` tensor). The rays which intersect the sphere are marked as True, while the ones that do not are marked as False.

Return type tuple

`pynif3d.utils.`**`rotation_mat`**`(`*angle*, *axis*`)`

Creates a rotation matrix given an angle and a coordinate axis.

Parameters

- **`angle`** (*float*) – Rotation angle (in radians).
- **`axis`** (*str*) – Rotation axis (“x”, “y” or “z”).

Returns Rotation matrix of shape `(4, 4)`.

Return type np.array

`pynif3d.utils.`**`translation_mat`**`(`*t*, *axis='z'*`)`

Generates a translation matrix given an input translation vector.

Parameters

- **`t`** (*float*) – Array containing the translation vector’s coordinates.
- **`axis`** (*str*) – Rotation axis (“x”, “y” or “z”). Default is “z”.

Returns Translation matrix of shape `(4, 4)`.

Return type np.array

2.14 pynif3d.vis

PYTHON MODULE INDEX

INDEX