rising Documentation

Release 0.2.1+0.gca0cf77.dirty

Justus Schock, Michael Baumgartner

Jul 26, 2021

GETTING STARTED

`rising` is a highly performant, `PyTorch` only, framework for efficient data augmentation with native support for volumetric data

CHAPTER

ONE

WHAT IS RISING?

Rising is a high-performance data loading and augmentation library for 2D *and* 3D data completely written in PyTorch. Our goal is to provide a seamless integration into the PyTorch Ecosystem without sacrificing usability or features. Multiple examples for different use cases can be found in our tutorial docs e.g. 2D Classification on MedNIST, 3D Segmentation of Hippocampus (Medical Decathlon), Example Transformation Output, Integration of External Frameworks

CHAPTER

TWO

INSTALLATION

Pypi Installation

```
pip install rising
```

Editable Installation for development

```
git clone git@github.com:PhoenixDL/rising.git
cd rising
pip install -e .
```

Running tests inside rising directory (top directory not the package directory)

```
python -m unittest
```

Check out our contributing guide for more information or additional help.

CHAPTER

THREE

WHAT CAN I DO WITH RISING?

Rising currently consists out of two main modules:

3.1 rising.loading

The `Dataloader` of rising will be your new best friend because it handles all your transformations and applies them efficiently to the data either on CPU or GPU. On CPU you can easily switch between transformations which can only be performed per sample and transformations which can be applied per batch. In contrast to the native PyTorch datasets you don't need to integrate your augmentation into your dataset. Hence, the only purpose of the dataset is to provide an interface to access individual data samples. Our `DataLoader` is a direct subclass of the PyTorch's dataloader and handles the batch assembly and applies the augmentations/transformations to the data.

3.2 rising.transforms

This module implements many transformations which can be used during training for preprocessing and augmentation. All of them are implemented directly in PyTorch such that gradients can be propagated through the transformations and (optionally) it can be applied on the GPU. Finally, all transforms are implemented for 2D (natural images) and 3D (volumetric) data.

In the future, support for keypoints and other geometric primitives which can be assembled by connected points will be added.

CHAPTER

FOUR

RISING MNIST EXAMPLE WITH CPU AND GPU AUGMENTATION

rising uses the same Dataset structure as PyTorch and thus we can just reuse the MNIST dataset from torchvision.

```
import torchvision
from torchvision.transforms import ToTensor

# define dataset and use to tensor trafo to convert PIL image to tensor
dataset = torchvision.datasets.MNIST('./', train=True, download=True,
                                     transform=ToTensor())
```

In the next step, the transformations/augmentations need to be defined. The first transforms converts the Sequence from the torchvision dataset into a dict for the following rising transform which work on dicts. At the end, the transforms are compose to one callable transform which can be passed to the Dataloader.

```
import rising.transforms as rtr
from rising.loading import DataLoader, default_transform_call
from rising.random import DiscreteParameter, UniformParameter

# define transformations
transforms = [
    rtr.SeqToMap("data", "label"),  # most rising transforms work on dicts
    rtr.NormZeroMeanUnitStd(keys=["data"]),
    rtr.Rot90((0, 1), keys=["data"], p=0.5),
    rtr.Mirror(dims=DiscreteParameter([0, 1]), keys=["data"]),
    rtr.Rotate(UniformParameter(0, 180), degree=True),
]

# by default rising assumes dicts but torchvision outputs tuples
# so we need to modify `transform_call` to support sequences and dicts
composed = rtr.Compose(transforms, transform_call=default_transform_call)
```

The Dataloader from rising automatically applies the specified transformations to the batches inside the multi-processing context of the CPU.

```
dataloader = DataLoader(
    dataset, batch_size=8, num_workers=8, batch_transforms=composed)
```

Alternatively, the augmentations can easily be applied on the GPU as well.

```
dataloader = DataLoader(
    dataset, batch_size=8, num_workers=8, gpu_transforms=composed)
```

If either the GPU or CPU is the bottleneck of the pipeline, the Dataloader can be used to balance the augmentations load between them.

```
transforms_cpu = rtr.Compose(transforms[:2])
transforms_gpu = rtr.Compose(transforms[2:])

dataloader = DataLoader(
    dataset, batch_size=8, num_workers=8,
    batch_transforms=transforms_cpu,
    gpu_transforms=transforms_gpu,
)
```

More details about how and where the augmentations are applied can be found below. You can also check out our example Notebooks for 2D Classification, 3D Segmentation and Transformation Examples.

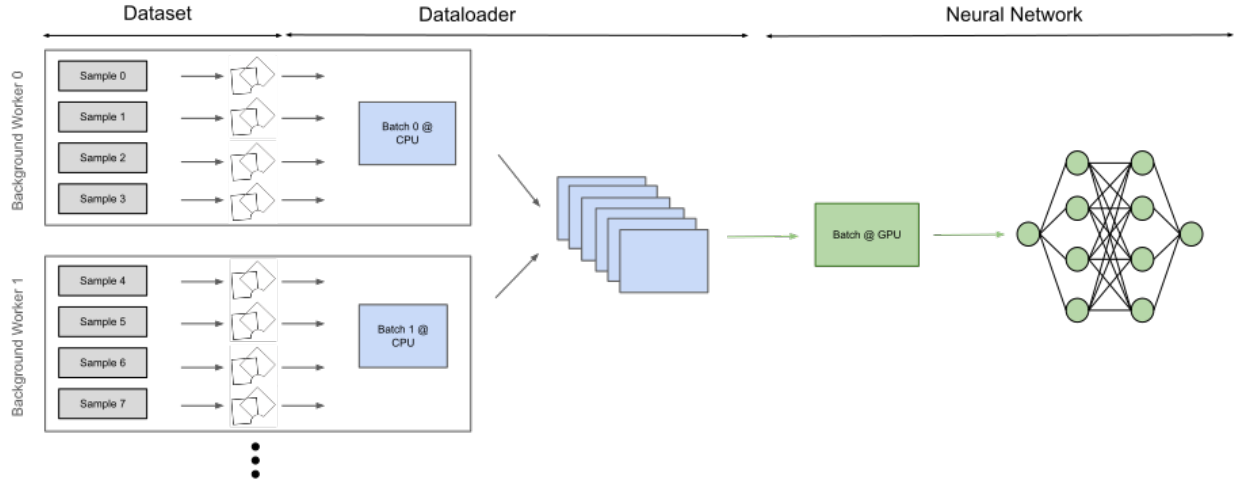
CHAPTER

FIVE

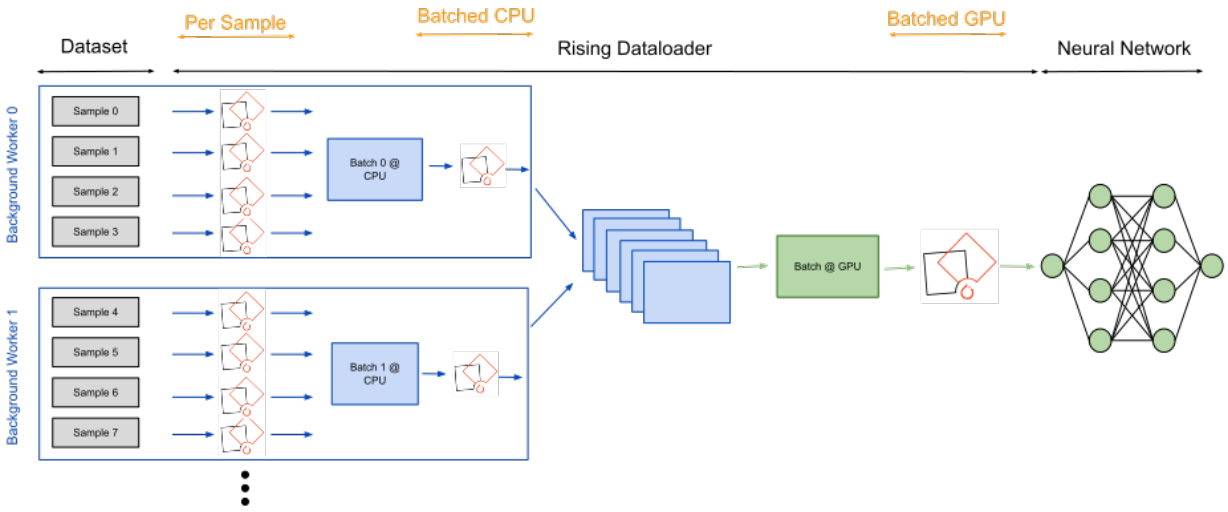
DATALOADING WITH `RISING`

In general you do not need to be familiar with the whole augmentation process which runs in the background but if you are still curious about the detailed pipeline this section will give a very short introduction into the backend of the `Dataloader`. The flow charts below highlight the differences between a conventional augmentation pipeline and the pipeline used in `rising`. CPU operations are visualized in blue while GPU operations are green.

The flow chart below visualizes the default augmentation pipeline of many other frameworks. The transformations are applied to individual samples which are loaded and augmented inside of multiple background workers from the CPU. This approach is already efficient and might only be slightly slower than batched execution of the transformations (if applied on the CPU). GPU augmentations can be used to perform many operations in parallel and profit heavily from vectorization.



`rising` lets the user decide from case to case where augmentations should be applied during this pipeline. This can heavily dependent on the specific tasks and the underlying hardware. Running augmentations on the GPU is only efficient if they can be executed in a batched fashion to maximize the parallelization GPUs can provide. As a consequence, `rising` implements all its transformations in a batched fashion and the `Dataloader` can execute them efficiently on the CPU and GPU. Optionally, the `Dataloader` can still be used to apply transformations on a per sample fashion, e.g. when transforms from other frameworks should be integrated.



Because the `rising` augmentation pipeline is a superset of the currently used methods, external frameworks can be integrated into `rising`.

CHAPTER

SIX

PROJECT ORGANIZATION

`Issues`: If you find any bugs, want some additional features or maybe just have a question don't hesitate to open an issue :)

`General Project Future`: Most of the features and the milestone organisation can be found inside the `projects` tab. Features which are planned for the next release/milestone are listed under `TODO Next Release` while features which are not scheduled yet are under `Todo`.

`Slack`: Join our Slack for the most up to date news or just to have a chat with us :)

CHAPTER

SEVEN

RISING.LOADING

`rising.loading` provides an alternative `DataLoader` that extends `torch.utils.data.DataLoader` by the following:

- Seeding of Numpy in each worker process: The seed is generated by numpy in the main process before starting the workers. For reproducibility numpy must be seeded in the main process.
- Per-Sample Transforms outside the dataset (optional with pseudo batch dimension if the transforms require it). Will be executed within the spawned worker processes before batching.
- Batched Transforms for better performance. Will be executed within the worker processes after batching.
- Batched GPU-Transforms. Will be executed after syncing results back to main process (i.e. as last transforms) to avoid multiple CUDA initializations.

Furthermore it also provides a `Dataset` (based on `torch.utils.data.Dataset`)that can create subsets from itself by given indices and an `AsyncDataset` as well as different options for collation.

7.1 DataLoader

class rising.loading.loader.**DataLoader**(*dataset*, *batch_size=1*, *shuffle=False*, *batch_transforms=None*, *gpu_transforms=None*, *sample_transforms=None*, *pseudo_batch_dim=False*, *device=None*, *sampler=None*, *batch_sampler=None*, *num_workers=0*, *collate_fn=None*, *pin_memory=False*, *drop_last=False*, *timeout=0*, *worker_init_fn=None*, *multiprocessing_context=None*, *auto_convert=True*, *transform_call=<function default_transform_call>*, ***kwargs*)

Bases: `torch.utils.data.DataLoader`

A DataLoader introducing batch-transforms, per-sample-transforms, numpy seeds for worker processes outside the dataset

---

Note: For Reproducibility numpy and pytorch must be seeded in the main process, as these frameworks will be used to generate their own seeds for each worker.

Note: `len(dataloader)` heuristic is based on the length of the sampler used. When `dataset` is an `IterableDataset`, an infinite sampler is used, whose `__len__()` is not implemented, because the actual

length depends on both the iterable as well as multi-process loading configurations. So one should not query this method unless they work with a map-style dataset.

Warning: If the `spawn` start method is used, `worker_init_fn` cannot be an unpicklable object, e.g., a lambda function. See Multiprocessing best practices on more details related to multiprocessing in PyTorch.

Note: The GPU-Transforms for a batch are always executed in the main process after the batch was gathered from subprocesses which apply the CPU-Transformations. The desired workflow is as follows:

Disk -> CPU-Transforms -> GPU-Memory -> GPU-Transforms -> Further GPU Processing (e.g. training a neural network)

Parameters

- **dataset** (Union[Sequence, Dataset]) – dataset from which to load the data
- **batch_size** (int) – how many samples per batch to load (default: 1).
- **shuffle** (bool) – set to True to have the data reshuffled at every epoch (default: False)
- **batch_transforms** (Optional[Callable]) – transforms which can be applied to a whole batch. Usually this accepts either mappings or sequences and returns the same type containing transformed elements
- **gpu_transforms** (Optional[Callable]) – transforms which can be applied to a whole batch (on the GPU). Unlike batch_transforms this is not done in multiple processes, but in the main process on the GPU, because GPUs are capable of non-blocking and asynchronous working. Before executing these transforms all data will be moved to device. This copy is done in a non-blocking way if pin_memory is set to True.
- **sample_transforms** (Optional[Callable]) – transforms applied to each sample (on CPU). These are the first transforms applied to the data, since they are applied on sample retrieval from dataset before batching occurs.
- **pseudo_batch_dim** (bool) – whether the sample_transforms work on batches and thus need a pseudo batch dim of 1 to work correctly.
- **device** (Union[str, device, None]) – the device to move the data to for gpu_transforms. If None: the device will be the current device.
- **sampler** (Optional[Sampler]) – defines the strategy to draw samples from the dataset. If specified, shuffle must be False.
- **batch_sampler** (Optional[Sampler]) – like sampler, but returns a batch of indices at a time. Mutually exclusive with batch_size, shuffle, sampler, and drop_last.
- **num_workers** (int) – how many subprocesses to use for data loading. 0 means that the data will be loaded in the main process. (default: 0)
- **collate_fn** (Optional[Callable]) – merges a list of samples to form a mini-batch of Tensor(s). Used when using batched loading from a map-style dataset.
- **pin_memory** (bool) – If True, the data loader will copy Tensors into CUDA pinned memory before returning them. If your data elements are a custom type, or your collate_fn returns a batch that is a custom type, see the example below.

- **drop_last** (bool) – set to True to drop the last incomplete batch, if the dataset size is not divisible by the batch size. If False and the size of dataset is not divisible by the batch size, then the last batch will be smaller. (default: False)
- **timeout** (Union[int, float]) – if positive, the timeout value for collecting a batch from workers. Should always be non-negative. (default: 0)
- **worker_init_fn** (Optional[Callable]) – If not None, this will be called on each worker subprocess with the worker id (an int in [0, num_workers - 1]) as input, after seeding and before data loading. (default: None)
- **auto_convert** (bool) – if set to True, the batches will always be transformed to torch.Tensors, if possible. (default: True)
- **transform_call** (Callable[[Any, Callable], Any]) – function which determines how transforms are called. By default Mappings and Sequences are unpacked during the transform.

get_batch_transformer()
A getter function for the *BatchTransformer* :returns: the initialized BatchTransformer :rtype: BatchTransformer

get_gpu_batch_transformer()
A getter function for the *BatchTransformer* holding the GPU-Transforms

Returns the initialized BatchTransformer

Return type *BatchTransformer*

get_sample_transformer()
A getter function for the SampleTransformer holding the Per-Sample-Transforms

Returns the initialized SampleTransformer

Return type SampleTransformer

rising.loading.loader.**default_transform_call**(*batch*, *transform*)
Default function to call transforms. Mapping and Sequences are unpacked during the transform call. Other types are passed as a positional argument.

Parameters

- **batch** (Any) – current batch which is passed to transforms
- **transform** (Callable) – transform to perform

Returns transformed batch

Return type Any

7.1.1 DataLoader

class rising.loading.loader.**DataLoader**(*dataset*, *batch_size=1*, *shuffle=False*, *batch_transforms=None*, *gpu_transforms=None*, *sample_transforms=None*, *pseudo_batch_dim=False*, *device=None*, *sampler=None*, *batch_sampler=None*, *num_workers=0*, *collate_fn=None*, *pin_memory=False*, *drop_last=False*, *timeout=0*, *worker_init_fn=None*, *multiprocessing_context=None*, *auto_convert=True*, *transform_call=<function default_transform_call>*, ***kwargs*)

Bases: torch.utils.data.DataLoader

A DataLoader introducing batch-transforms, per-sample-transforms, numpy seeds for worker processes outside the dataset

---

Note: For Reproducibility numpy and pytorch must be seeded in the main process, as these frameworks will be used to generate their own seeds for each worker.

Note: `len(dataloader)` heuristic is based on the length of the sampler used. When `dataset` is an `IterableDataset`, an infinite sampler is used, whose `__len__()` is not implemented, because the actual length depends on both the iterable as well as multi-process loading configurations. So one should not query this method unless they work with a map-style dataset.

Warning: If the `spawn` start method is used, `worker_init_fn` cannot be an unpicklable object, e.g., a lambda function. See Multiprocessing best practices on more details related to multiprocessing in PyTorch.

Note: The GPU-Transforms for a batch are always executed in the main process after the batch was gathered from subprocesses which apply the CPU-Transformations. The desired workflow is as follows:

Disk -> CPU-Transforms -> GPU-Memory -> GPU-Transforms -> Further GPU Processing (e.g. training a neural network)

Parameters

- **dataset** (Union[Sequence, Dataset]) – dataset from which to load the data
- **batch_size** (int) – how many samples per batch to load (default: 1).
- **shuffle** (bool) – set to True to have the data reshuffled at every epoch (default: False)
- **batch_transforms** (Optional[Callable]) – transforms which can be applied to a whole batch. Usually this accepts either mappings or sequences and returns the same type containing transformed elements
- **gpu_transforms** (Optional[Callable]) – transforms which can be applied to a whole batch (on the GPU). Unlike `batch_transforms` this is not done in multiple processes, but in the main process on the GPU, because GPUs are capable of non-blocking and asynchronous working. Before executing these transforms all data will be moved to `device`. This copy is done in a non-blocking way if `pin_memory` is set to True.

- **sample_transforms** (Optional[Callable]) – transforms applied to each sample (on CPU). These are the first transforms applied to the data, since they are applied on sample retrieval from dataset before batching occurs.
- **pseudo_batch_dim** (bool) – whether the sample_transforms work on batches and thus need a pseudo batch dim of 1 to work correctly.
- **device** (Union[str, device, None]) – the device to move the data to for gpu_transforms. If None: the device will be the current device.
- **sampler** (Optional[Sampler]) – defines the strategy to draw samples from the dataset. If specified, shuffle must be False.
- **batch_sampler** (Optional[Sampler]) – like sampler, but returns a batch of indices at a time. Mutually exclusive with batch_size, shuffle, sampler, and drop_last.
- **num_workers** (int) – how many subprocesses to use for data loading. 0 means that the data will be loaded in the main process. (default: 0)
- **collate_fn** (Optional[Callable]) – merges a list of samples to form a mini-batch of Tensor(s). Used when using batched loading from a map-style dataset.
- **pin_memory** (bool) – If True, the data loader will copy Tensors into CUDA pinned memory before returning them. If your data elements are a custom type, or your collate_fn returns a batch that is a custom type, see the example below.
- **drop_last** (bool) – set to True to drop the last incomplete batch, if the dataset size is not divisible by the batch size. If False and the size of dataset is not divisible by the batch size, then the last batch will be smaller. (default: False)
- **timeout** (Union[int, float]) – if positive, the timeout value for collecting a batch from workers. Should always be non-negative. (default: 0)
- **worker_init_fn** (Optional[Callable]) – If not None, this will be called on each worker subprocess with the worker id (an int in [0, num_workers - 1]) as input, after seeding and before data loading. (default: None)
- **auto_convert** (bool) – if set to True, the batches will always be transformed to torch.Tensors, if possible. (default: True)
- **transform_call** (Callable[[Any, Callable], Any]) – function which determines how transforms are called. By default Mappings and Sequences are unpacked during the transform.

get_batch_transformer()
A getter function for the *BatchTransformer* :returns: the initialized BatchTransformer :rtype: BatchTransformer

get_gpu_batch_transformer()
A getter function for the *BatchTransformer* holding the GPU-Transforms

Returns the initialized BatchTransformer

Return type *BatchTransformer*

get_sample_transformer()
A getter function for the SampleTransformer holding the Per-Sample-Transforms

Returns the initialized SampleTransformer

Return type SampleTransformer

7.1.2 default_transform_call

rising.loading.loader.**default_transform_call**(*batch*, *transform*)

Default function to call transforms. Mapping and Sequences are unpacked during the transform call. Other types are passed as a positional argument.

Parameters

- **batch** (Any) – current batch which is passed to transforms
- **transform** (Callable) – transform to perform

Returns transformed batch

Return type Any

7.1.3 BatchTransformer

class rising.loading.loader.**BatchTransformer**(*collate_fn*, *transforms=None*, *auto_convert=True*, *transform_call=<function default_transform_call>*)

Bases: object

A callable wrapping the collate_fn to enable transformations on a batch-basis.

Parameters

- **collate_fn** (Callable) – merges a list of samples to form a mini-batch of Tensor(s). Used when using batched loading from a map-style dataset.
- **transforms** (Optional[Callable]) – transforms which can be applied to a whole batch. Usually this accepts either mappings or sequences and returns the same type containing transformed elements
- **auto_convert** (bool) – if set to True, the batches will always be transformed to torch.Tensors, if possible. (default: True)
- **transform_call** (Callable[[Any, Callable], Any]) – function which determines how transforms are called. By default Mappings and Sequences are unpacked during the transform.

__call__(**args*, ***kwargs*)

Apply batch workflow: collate -> augmentation -> default_convert

Parameters

- ***args** – positional batch arguments
- ****kwargs** – keyword batch arguments

Returns batched and augmented data

Return type Any

7.1.4 patch_worker_init_fn

rising.loading.loader.**patch_worker_init_fn**(*loader*, *new_worker_init*)
Patches the loader to temporarily have the correct worker init function.

Parameters

- **loader** (*DataLoader*) – the loader to patch
- **new_worker_init** (Callable) – the new worker init function

Yields the patched loader

Return type Generator

7.1.5 patch_collate_fn

rising.loading.loader.**patch_collate_fn**(*loader*)
Patches the loader to temporarily have the correct collate function

Parameters **loader** (*DataLoader*) – the loader to patch

Yields the patched loader

Return type Generator

7.2 Dataset

class rising.loading.dataset.**Dataset**(**args*, ***kwargs*)
Bases: torch.utils.data.Dataset

Extension of torch.utils.data.Dataset by a get_subset method which returns a sub-dataset.

get_subset(*indices*)
Returns a torch.utils.data.Subset of the current dataset based on given indices

Parameters **indices** (Sequence[int]) – valid indices to extract subset from current dataset

Returns the subset of the current dataset

Return type Subset

class rising.loading.dataset.**AsyncDataset**(*data_path*, *load_fn*, *mode='append'*, *num_workers=0*, *verbose=False*, ***load_kwargs*)
Bases: *rising.loading.dataset.Dataset*

A dataset to preload all the data and cache it for the entire lifetime of this class.

Parameters

- **data_path** (Union[Path, str, list]) – the path(s) containing the actual data samples
- **load_fn** (Callable) – function to load the actual data
- **mode** (str) – whether to append the sample to a list or to extend the list by it. Supported modes are: append and extend. Default: append
- **num_workers** (Optional[int]) – the number of workers to use for preloading. 0 means, all the data will be loaded in the main process, while None means, the number of processes will default to the number of logical cores.

- **verbose** (bool) – whether to show the loading progress.
- ****load_kwargs** – additional keyword arguments. Passed directly to load_fn

Warning: if using multiprocessing to load data, there are some restrictions to which load_fn() are supported, please refer to the dill or pickle documentation

static _add_item(*data*, *item*, *mode*)
Adds items to the given data list. The actual way of adding these items depends on mode

Parameters

- **data** (list) – the list containing the already loaded data
- **item** (Any) – the current item which will be added to the list
- **mode** (str) – the string specifying the mode of how the item should be added.F

Raises **TypeError** – No known mode detected

Return type None

_make_dataset(*path*, *mode*)
Function to build the entire dataset

Parameters

- **path** (Union[Path, str, list]) – the path(s) containing the data samples
- **mode** (str) – whether to append or extend the dataset by the loaded sample

Returns the loaded data

Return type list

load_multi_process(*load_fn*, *path*)
Helper function to load dataset with multiple processes

Parameters

- **load_fn** (Callable) – function to load a single sample
- **path** (Sequence) – a sequence of paths which should be loaded

Returns loaded data

Return type list

load_single_process(*load_fn*, *path*)
Helper function to load dataset with single process

Parameters

- **load_fn** (Callable) – function to load a single sample
- **path** (Sequence) – a sequence of paths which should be loaded

Returns iterator of loaded data

Return type Iterator

7.2.1 Dataset

class rising.loading.dataset.**Dataset**(**args*, ***kwargs*)
Bases: torch.utils.data.Dataset

Extension of torch.utils.data.Dataset by a get_subset method which returns a sub-dataset.

get_subset(*indices*)
Returns a torch.utils.data.Subset of the current dataset based on given indices

Parameters **indices** (Sequence[int]) – valid indices to extract subset from current dataset

Returns the subset of the current dataset

Return type Subset

7.2.2 AsyncDataset

class rising.loading.dataset.**AsyncDataset**(*data_path*, *load_fn*, *mode='append'*, *num_workers=0*, *verbose=False*, ***load_kwargs*)
Bases: *rising.loading.dataset.Dataset*

A dataset to preload all the data and cache it for the entire lifetime of this class.

Parameters

- **data_path** (Union[Path, str, list]) – the path(s) containing the actual data samples
- **load_fn** (Callable) – function to load the actual data
- **mode** (str) – whether to append the sample to a list or to extend the list by it. Supported modes are: append and extend. Default: append
- **num_workers** (Optional[int]) – the number of workers to use for preloading. 0 means, all the data will be loaded in the main process, while None means, the number of processes will default to the number of logical cores.
- **verbose** (bool) – whether to show the loading progress.
- ****load_kwargs** – additional keyword arguments. Passed directly to load_fn

Warning: if using multiprocessing to load data, there are some restrictions to which load_fn() are supported, please refer to the dill or pickle documentation

static _add_item(*data*, *item*, *mode*)
Adds items to the given data list. The actual way of adding these items depends on mode

Parameters

- **data** (list) – the list containing the already loaded data
- **item** (Any) – the current item which will be added to the list
- **mode** (str) – the string specifying the mode of how the item should be added.F

Raises **TypeError** – No known mode detected

Return type None

_make_dataset(*path*, *mode*)
Function to build the entire dataset

Parameters

- **path** (Union[Path, str, list]) – the path(s) containing the data samples
- **mode** (str) – whether to append or extend the dataset by the loaded sample

Returns the loaded data

Return type list

load_multi_process(*load_fn*, *path*)
Helper function to load dataset with multiple processes

Parameters

- **load_fn** (Callable) – function to load a single sample
- **path** (Sequence) – a sequence of paths which should be loaded

Returns loaded data

Return type list

load_single_process(*load_fn*, *path*)
Helper function to load dataset with single process

Parameters

- **load_fn** (Callable) – function to load a single sample
- **path** (Sequence) – a sequence of paths which should be loaded

Returns iterator of loaded data

Return type Iterator

7.2.3 dill_helper

rising.loading.dataset.**dill_helper**(*payload*)
Load single sample from data serialized by dill :type payload: Any :param payload: data which is loaded with dill

Returns loaded data

Return type Any

7.2.4 load_async

rising.loading.dataset.**load_async**(*pool*, *fn*, **args*, *callback=None*, ***kwargs*)
Load data asynchronously and serialize data via dill

Parameters

- **pool** (Pool) – multiprocessing pool to use for apply_async()
- **fn** (Callable) – function to load a single sample
- ***args** – positional arguments to dump with dill
- **callback** (Optional[Callable]) – optional callback. defaults to None.
- ****kwargs** – keyword arguments to dump with dill

Returns reference to obtain data with get()

Return type Any

7.3 Collation

rising.loading.collate.**numpy_collate**(*batch*)
function to collate the samples to a whole batch of numpy arrays. PyTorch Tensors, scalar values and sequences will be casted to arrays automatically.

Parameters **batch** (Any) – a batch of samples. In most cases either sequence, mapping or mixture of them

Returns

collated batch with optionally converted type (to numpy.ndarray)

Return type Any

Raises **TypeError** – When batch could not be collated automatically

rising.loading.collate.**do_nothing_collate**(*batch*)
Returns the batch as is (with out any collation :type batch: Any :param batch: input batch (typically a sequence, mapping or mixture of those).

Returns the batch as given to this function

Return type Any

7.3.1 numpy_collate

rising.loading.collate.**numpy_collate**(*batch*)
function to collate the samples to a whole batch of numpy arrays. PyTorch Tensors, scalar values and sequences will be casted to arrays automatically.

Parameters **batch** (Any) – a batch of samples. In most cases either sequence, mapping or mixture of them

Returns

collated batch with optionally converted type (to numpy.ndarray)

Return type Any

Raises **TypeError** – When batch could not be collated automatically

7.3.2 do_nothing_collate

rising.loading.collate.**do_nothing_collate**(*batch*)
Returns the batch as is (with out any collation :type batch: Any :param batch: input batch (typically a sequence, mapping or mixture of those).

Returns the batch as given to this function

Return type Any

CHAPTER

EIGHT

RISING.OPS

Provides Operators working on single tensors.

8.1 On Tensors

8.1.1 torch_one_hot

rising.ops.tensor.**torch_one_hot**(*target*, *num_classes=None*)

Compute one hot encoding of input tensor

Parameters

- **target** (Tensor) – tensor to be converted
- **num_classes** (Optional[int]) – number of classes. If num_classes is None, the maximum of target is used

Returns one hot encoded tensor

Return type torch.Tensor

8.1.2 np_one_hot

rising.ops.tensor.**np_one_hot**(*target*, *num_classes=None*)

Compute one hot encoding of input array

Parameters

- **target** (ndarray) – array to be converted
- **num_classes** (Optional[int]) – number of classes

Returns one hot encoded array

Return type numpy.ndarray

CHAPTER

NINE

RISING.RANDOM

9.1 Random Parameter Injection Base Classes

class rising.random.abstract.**AbstractParameter**(**args*, ***kwargs*)

Bases: torch.nn.Module

Abstract Parameter class to inject randomness to transforms

static _get_n_samples(*size=(1,)*)

Calculates the number of elements in the given size

Parameters **size** (Union[Sequence, Size]) – Sequence or torch.Size

Returns the number of elements

Return type int

forward(*size=None*, *device=None*, *dtype=None*, *tensor_like=None*)

Forward function (will also be called if the module is called). Calculates the number of samples from the given shape, performs the sampling and converts it back to the correct shape.

Parameters

- **size** (Union[Sequence, Size, None]) – the size of the sampled values. If None, it samples one value without reshaping
- **device** (Union[device, str, None]) – the device the result value should be set to, if it is a tensor
- **dtype** (Union[dtype, str, None]) – the dtype, the result value should be casted to, if it is a tensor
- **tensor_like** (Optional[Tensor]) – the tensor, having the correct dtype and device. The result will be pushed onto this device and casted to this dtype if this is specified.

Returns the sampled values

Return type list or torch.Tensor

Notes

if the parameter `tensor_like` is given, it overwrites the parameters `dtype` and `device`

abstract sample(*n_samples*)
Abstract sampling function

Parameters **n_samples** (`int`) – the number of samples to return

Returns the sampled values

Return type torch.Tensor or list

9.1.1 AbstractParameter

class rising.random.abstract.**AbstractParameter**(**args*, ***kwargs*)
Bases: `torch.nn.Module`

Abstract Parameter class to inject randomness to transforms

static _get_n_samples(*size=(1,)*)
Calculates the number of elements in the given size

Parameters **size** (`Union[Sequence, Size]`) – Sequence or torch.Size

Returns the number of elements

Return type int

forward(*size=None*, *device=None*, *dtype=None*, *tensor_like=None*)
Forward function (will also be called if the module is called). Calculates the number of samples from the given shape, performs the sampling and converts it back to the correct shape.

Parameters

- **size** (`Union[Sequence, Size, None]`) – the size of the sampled values. If None, it samples one value without reshaping
- **device** (`Union[device, str, None]`) – the device the result value should be set to, if it is a tensor
- **dtype** (`Union[dtype, str, None]`) – the dtype, the result value should be casted to, if it is a tensor
- **tensor_like** (`Optional[Tensor]`) – the tensor, having the correct dtype and device. The result will be pushed onto this device and casted to this dtype if this is specified.

Returns the sampled values

Return type list or torch.Tensor

Notes

if the parameter `tensor_like` is given, it overwrites the parameters `dtype` and `device`

abstract sample(*n_samples*)
Abstract sampling function

Parameters **n_samples** (`int`) – the number of samples to return

Returns the sampled values

Return type torch.Tensor or list

9.2 Continuous Random Parameters

class rising.random.continuous.**ContinuousParameter**(*distribution*)
Bases: *rising.random.abstract.AbstractParameter*

Class to perform parameter sampling from torch distributions

Parameters **distribution** (`Distribution`) – the distribution to sample from

sample(*n_samples*)
Samples from the internal distribution

Parameters **n_samples** (`int`) – the number of elements to sample

Returns torch.Tensor: samples

Return type `Tensor`

class rising.random.continuous.**NormalParameter**(*mu*, *sigma*)
Bases: *rising.random.continuous.ContinuousParameter*

Samples Parameters from a normal distribution. For details have a look at `torch.distributions.Normal`

Parameters

- **mu** (`Union[float, Tensor]`) – the distributions mean
- **sigma** (`Union[float, Tensor]`) – the distributions standard deviation

class rising.random.continuous.**UniformParameter**(*low*, *high*)
Bases: *rising.random.continuous.ContinuousParameter*

Samples Parameters from a uniform distribution. For details have a look at `torch.distributions.Uniform`

Parameters

- **low** (`Union[float, Tensor]`) – the lower range (inclusive)
- **high** (`Union[float, Tensor]`) – the higher range (exclusive)

9.2.1 ContinuousParameter

class rising.random.continuous.**ContinuousParameter**(*distribution*)
Bases: *rising.random.abstract.AbstractParameter*

Class to perform parameter sampling from torch distributions

Parameters **distribution** (Distribution) – the distribution to sample from

sample(*n_samples*)
Samples from the internal distribution

Parameters **n_samples** (int) – the number of elements to sample

Returns torch.Tensor: samples

Return type Tensor

9.2.2 NormalParameter

class rising.random.continuous.**NormalParameter**(*mu*, *sigma*)
Bases: *rising.random.continuous.ContinuousParameter*

Samples Parameters from a normal distribution. For details have a look at torch.distributions.Normal

Parameters

- **mu** (Union[float, Tensor]) – the distributions mean
- **sigma** (Union[float, Tensor]) – the distributions standard deviation

9.2.3 UniformParameter

class rising.random.continuous.**UniformParameter**(*low*, *high*)
Bases: *rising.random.continuous.ContinuousParameter*

Samples Parameters from a uniform distribution. For details have a look at torch.distributions.Uniform

Parameters

- **low** (Union[float, Tensor]) – the lower range (inclusive)
- **high** (Union[float, Tensor]) – the higher range (exclusive)

9.3 Discrete Random Parameters

class rising.random.discrete.**DiscreteParameter**(*population*, *replacement=False*, *weights=None*, *cum_weights=None*)
Bases: *rising.random.abstract.AbstractParameter*

Samples parameters from a discrete population with or without replacement

Parameters

- **population** (Sequence) – the parameter population to sample from
- **replacement** (bool) – whether or not to sample with replacement

- **weights** (Optional[Sequence]) – relative sampling weights
- **cum_weights** (Optional[Sequence]) – cumulative sampling weights

sample(*n_samples*)
Samples from the discrete internal population

Parameters **n_samples** (int) – the number of elements to sample

Returns the sampled values

Return type list

class rising.random.discrete.**DiscreteCombinationsParameter**(*population*, *replacement=False*)
Bases: *rising.random.discrete.DiscreteParameter*

Sample parameters from an extended population which consists of all possible combinations of the given population

Parameters

- **population** (Sequence) – population to build combination of
- **replacement** (bool) – whether or not to sample with replacement

9.3.1 DiscreteParameter

class rising.random.discrete.**DiscreteParameter**(*population*, *replacement=False*, *weights=None*, *cum_weights=None*)
Bases: *rising.random.abstract.AbstractParameter*

Samples parameters from a discrete population with or without replacement

Parameters

- **population** (Sequence) – the parameter population to sample from
- **replacement** (bool) – whether or not to sample with replacement
- **weights** (Optional[Sequence]) – relative sampling weights
- **cum_weights** (Optional[Sequence]) – cumulative sampling weights

sample(*n_samples*)
Samples from the discrete internal population

Parameters **n_samples** (int) – the number of elements to sample

Returns the sampled values

Return type list

9.3.2 DiscreteCombinationsParameter

class rising.random.discrete.**DiscreteCombinationsParameter**(*population*, *replacement=False*)

Bases: *rising.random.discrete.DiscreteParameter*

Sample parameters from an extended population which consists of all possible combinations of the given population

Parameters

- **population** (Sequence) – population to build combination of
- **replacement** (bool) – whether or not to sample with replacement

9.3.3 combinations_all

rising.random.discrete.**combinations_all**(*data*)

Return all combinations of all length for given sequence

Parameters **data** (Sequence) – sequence to get combinations of

Returns all combinations

Return type List

CHAPTER

TEN

RISING.TRANSFORMS

Provides the Augmentations and Transforms used by the `rising.loading.DataLoader`.

Implementations include:

- Transformation Base Classes
- Composed Transforms
- Affine Transforms
- Channel Transforms
- Cropping Transforms
- Device Transforms
- Format Transforms
- Intensity Transforms
- Kernel Transforms
- Spatial Transforms
- Tensor Transforms
- Utility Transforms

10.1 Transformation Base Classes

class rising.transforms.abstract.**AbstractTransform**(*grad=False*, ***kwargs*)

Bases: `torch.nn.Module`

Base class for all transforms

Parameters **grad** (`bool`) – enable gradient computation inside transformation

__call__(**args*, ***kwargs*)

Call super class with correct torch context

Parameters

- ***args** – forwarded positional arguments
- ****kwargs** – forwarded keyword arguments

Returns transformed data

Return type Any

forward(***data*)
Implement transform functionality here

Parameters ****data** – dict with data

Returns dict with transformed data

Return type dict

register_sampler(*name*, *sampler*, **args*, ***kwargs*)
Registers a parameter sampler to the transform. Internally a property is created to forward calls to the attribute to calls of the sampler.

Parameters

- **name** (str) – the property name
- **sampler** (Union[Sequence, *AbstractParameter*]) – the sampler. Will be wrapped to a sampler always returning the same element if not already a sampler
- ***args** – additional positional arguments (will be forwarded to sampler call)
- ****kwargs** – additional keyword arguments (will be forwarded to sampler call)

class rising.transforms.abstract.**BaseTransform**(*augment_fn*, **args*, *keys=('data',)*, *grad=False*, *property_names=()*, ***kwargs*)
Bases: *rising.transforms.abstract.AbstractTransform*

Transform to apply a functional interface to given keys

Warning: This transform should not be used with functions which have randomness build in because it will result in different augmentations per key.

Parameters

- **augment_fn** (Callable[[Tensor], Any]) – function for augmentation
- ***args** – positional arguments passed to augment_fn
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **property_names** (Sequence[str]) – a tuple containing all the properties to call during forward pass
- ****kwargs** – keyword arguments passed to augment_fn

forward(***data*)
Apply transformation

Parameters **data** – dict with tensors

Returns dict with augmented data

Return type dict

class rising.transforms.abstract.**PerSampleTransform**(*augment_fn*, **args*, *keys=('data',)*, *grad=False*, *property_names=()*, ***kwargs*)
Bases: *rising.transforms.abstract.BaseTransform*

Apply transformation to each sample in batch individually `augment_fn` must be callable with option `out` where results are saved in.

Warning: This transform should not be used with functions which have randomness build in because it will result in different augmentations per sample and key.

Parameters

- **augment_fn** (Callable[[Tensor], Any]) – function for augmentation
- ***args** – positional arguments passed to augment_fn
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **property_names** (Sequence[str]) – a tuple containing all the properties to call during forward pass
- ****kwargs** – keyword arguments passed to augment_fn

forward(***data*)

Parameters **data** – dict with tensors

Returns dict with augmented data

Return type dict

class rising.transforms.abstract.**PerChannelTransform**(*augment_fn*, *per_channel=False*, *keys=('data',)*, *grad=False*, *property_names=()*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Apply transformation per channel (but still to whole batch)

Warning: This transform should not be used with functions which have randomness build in because it will result in different augmentations per channel and key.

Parameters

- **augment_fn** (Callable[[Tensor], Any]) – function for augmentation
- **per_channel** (bool) – enable transformation per channel
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **kwargs** – keyword arguments passed to augment_fn

forward(***data*)

Apply transformation

Parameters **data** – dict with tensors

Returns dict with augmented data

Return type dict

class rising.transforms.abstract.**BaseTransformSeeded**(*augment_fn*, **args*, *keys=('data',)*, *grad=False*, *property_names=()*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Transform to apply a functional interface to given keys and use the same pytorch(!) seed for every key.

Parameters

- **augment_fn** (Callable[[Tensor], Any]) – function for augmentation
- ***args** – positional arguments passed to augment_fn
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **property_names** (Sequence[str]) – a tuple containing all the properties to call during forward pass
- ****kwargs** – keyword arguments passed to augment_fn

forward(***data*)

Apply transformation and use same seed for every key

Parameters **data** – dict with tensors

Returns dict with augmented data

Return type dict

10.1.1 AbstractTransform

class rising.transforms.abstract.**AbstractTransform**(*grad=False*, ***kwargs*)

Bases: torch.nn.Module

Base class for all transforms

Parameters **grad** (bool) – enable gradient computation inside transformation

__call__(**args*, ***kwargs*)

Call super class with correct torch context

Parameters

- ***args** – forwarded positional arguments
- ****kwargs** – forwarded keyword arguments

Returns transformed data

Return type Any

forward(***data*)

Implement transform functionality here

Parameters ****data** – dict with data

Returns dict with transformed data

Return type dict

register_sampler(*name*, *sampler*, **args*, ***kwargs*)

Registers a parameter sampler to the transform. Internally a property is created to forward calls to the attribute to calls of the sampler.

Parameters

- **name** (str) – the property name
- **sampler** (Union[Sequence, *AbstractParameter*]) – the sampler. Will be wrapped to a sampler always returning the same element if not already a sampler
- ***args** – additional positional arguments (will be forwarded to sampler call)
- ****kwargs** – additional keyword arguments (will be forwarded to sampler call)

10.1.2 BaseTransform

class rising.transforms.abstract.**BaseTransform**(*augment_fn*, **args*, *keys=('data',)*, *grad=False*, *property_names=()*, ***kwargs*)

Bases: *rising.transforms.abstract.AbstractTransform*

Transform to apply a functional interface to given keys

Warning: This transform should not be used with functions which have randomness build in because it will result in different augmentations per key.

Parameters

- **augment_fn** (Callable[[Tensor], Any]) – function for augmentation
- ***args** – positional arguments passed to augment_fn
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **property_names** (Sequence[str]) – a tuple containing all the properties to call during forward pass
- ****kwargs** – keyword arguments passed to augment_fn

forward(***data*)

Apply transformation

Parameters **data** – dict with tensors

Returns dict with augmented data

Return type dict

10.1.3 BaseTransformSeeded

class rising.transforms.abstract.**BaseTransformSeeded**(*augment_fn*, **args*, *keys=('data',)*, *grad=False*, *property_names=()*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Transform to apply a functional interface to given keys and use the same pytorch(!) seed for every key.

Parameters

- **augment_fn** (Callable[[Tensor], Any]) – function for augmentation

- ***args** – positional arguments passed to augment_fn
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **property_names** (Sequence[str]) – a tuple containing all the properties to call during forward pass
- ****kwargs** – keyword arguments passed to augment_fn

forward(***data*)

Apply transformation and use same seed for every key

Parameters **data** – dict with tensors

Returns dict with augmented data

Return type dict

10.1.4 PerSampleTransform

class rising.transforms.abstract.**PerSampleTransform**(*augment_fn*, **args*, *keys=('data',)*, *grad=False*, *property_names=()*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Apply transformation to each sample in batch individually `augment_fn` must be callable with option `out` where results are saved in.

Warning: This transform should not be used with functions which have randomness build in because it will result in different augmentations per sample and key.

Parameters

- **augment_fn** (Callable[[Tensor], Any]) – function for augmentation
- ***args** – positional arguments passed to augment_fn
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **property_names** (Sequence[str]) – a tuple containing all the properties to call during forward pass
- ****kwargs** – keyword arguments passed to augment_fn

forward(***data*)

Parameters **data** – dict with tensors

Returns dict with augmented data

Return type dict

10.1.5 PerChannelTransform

class rising.transforms.abstract.**PerChannelTransform**(*augment_fn*, *per_channel=False*, *keys=('data',)*, *grad=False*, *property_names=()*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Apply transformation per channel (but still to whole batch)

Warning: This transform should not be used with functions which have randomness build in because it will result in different augmentations per channel and key.

Parameters

- **augment_fn** (Callable[[Tensor], Any]) – function for augmentation
- **per_channel** (bool) – enable transformation per channel
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **kwargs** – keyword arguments passed to augment_fn

forward(***data*)

Apply transformation

Parameters **data** – dict with tensors

Returns dict with augmented data

Return type dict

10.2 Compose Transforms

class rising.transforms.compose.**Compose**(**transforms*, *shuffle=False*, *transform_call=<function dict_call>*)

Bases: *rising.transforms.abstract.AbstractTransform*

Compose multiple transforms

Parameters

- **transforms** (Union[*AbstractTransform*, Sequence[*AbstractTransform*]]) – one or multiple transformations which are applied in consecutive order
- **shuffle** (bool) – apply transforms in random order
- **transform_call** (Callable[[Any, Callable], Any]) – function which determines how transforms are called. By default Mappings and Sequences are unpacked during the transform.

forward(**seq_like*, ***map_like*)

Apply transforms in a consecutive order. Can either handle Sequence like or Mapping like data.

Parameters

- ***seq_like** – data which is unpacked like a Sequence

- ****map_like** – data which is unpacked like a dict

Returns transformed data

Return type Union[Sequence, Mapping]

property shuffle
Getter for attribute shuffle

Returns True if shuffle is enabled, False otherwise

Return type bool

property transforms
Transforms getter

Returns transforms to compose

Return type torch.nn.ModuleList

class rising.transforms.compose.**DropoutCompose**(**transforms*, *dropout=0.5*, *shuffle=False*, *random_sampler=None*, *transform_call=<function dict_call>*, ***kwargs*)

Bases: *rising.transforms.compose.Compose*

Compose multiple transforms to one and randomly apply them

Parameters

- ***transforms** – one or multiple transformations which are applied in consecutive order
- **dropout** (Union[float, Sequence[float]]) – if provided as float, each transform is skipped with the given probability if dropout is a sequence, it needs to specify the dropout probability for each given transform
- **shuffle** (bool) – apply transforms in random order
- **random_sampler** (Optional[*ContinuousParameter*]) – a continuous parameter sampler. Samples a random value for each of the transforms.
- **transform_call** (Callable[[Any, Callable], Any]) – function which determines how transforms are called. By default Mappings and Sequences are unpacked during the transform.

Raises **ValueError** – if dropout is a sequence it must have the same length as transforms

forward(**seq_like*, ***map_like*)
Apply transforms in a consecutive order. Can either handle Sequence like or Mapping like data.

Parameters

- ***seq_like** – data which is unpacked like a Sequence
- ****map_like** – data which is unpacked like a dict

Returns dict with transformed data

Return type Union[Sequence, Mapping]

class rising.transforms.compose.**OneOf**(**transforms*, *weights=None*, *p=1.0*, *transform_call=<function dict_call>*)

Bases: *rising.transforms.abstract.AbstractTransform*

Apply one of the given transforms.

Parameters

- ***transforms** – transforms to choose from
- **weights** (Optional[Sequence[float]]) – additional weights for transforms
- **p** (float) – probability that one transform i applied
- **transform_call** (Callable[[Any, Callable], Any]) – function which determines how transforms are called. By default Mappings and Sequences are unpacked during the transform.

forward(***data*)
Implement transform functionality here

Parameters ****data** – dict with data

Returns dict with transformed data

Return type dict

10.2.1 Compose

class rising.transforms.compose.**Compose**(**transforms*, *shuffle=False*, *transform_call=<function dict_call>*)
Bases: *rising.transforms.abstract.AbstractTransform*

Compose multiple transforms

Parameters

- **transforms** (Union[*AbstractTransform*, Sequence[*AbstractTransform*]]) – one or multiple transformations which are applied in consecutive order
- **shuffle** (bool) – apply transforms in random order
- **transform_call** (Callable[[Any, Callable], Any]) – function which determines how transforms are called. By default Mappings and Sequences are unpacked during the transform.

forward(**seq_like*, ***map_like*)
Apply transforms in a consecutive order. Can either handle Sequence like or Mapping like data.

Parameters

- ***seq_like** – data which is unpacked like a Sequence
- ****map_like** – data which is unpacked like a dict

Returns transformed data

Return type Union[Sequence, Mapping]

property shuffle
Getter for attribute shuffle

Returns True if shuffle is enabled, False otherwise

Return type bool

property transforms
Transforms getter

Returns transforms to compose

Return type torch.nn.ModuleList

10.2.2 DropoutCompose

class rising.transforms.compose.**DropoutCompose**(**transforms*, *dropout=0.5*, *shuffle=False*, *random_sampler=None*, *transform_call=<function dict_call>*, ***kwargs*)

Bases: *rising.transforms.compose.Compose*

Compose multiple transforms to one and randomly apply them

Parameters

- ***transforms** – one or multiple transformations which are applied in consecutive order
- **dropout** (Union[float, Sequence[float]]) – if provided as float, each transform is skipped with the given probability if `dropout` is a sequence, it needs to specify the dropout probability for each given transform
- **shuffle** (bool) – apply transforms in random order
- **random_sampler** (Optional[*ContinuousParameter*]) – a continuous parameter sampler. Samples a random value for each of the transforms.
- **transform_call** (Callable[[Any, Callable], Any]) – function which determines how transforms are called. By default Mappings and Sequences are unpacked during the transform.

Raises **ValueError** – if dropout is a sequence it must have the same length as transforms

forward(**seq_like*, ***map_like*)

Apply transforms in a consecutive order. Can either handle Sequence like or Mapping like data.

Parameters

- ***seq_like** – data which is unpacked like a Sequence
- ****map_like** – data which is unpacked like a dict

Returns dict with transformed data

Return type Union[Sequence, Mapping]

10.2.3 OneOf

class rising.transforms.compose.**OneOf**(**transforms*, *weights=None*, *p=1.0*, *transform_call=<function dict_call>*)

Bases: *rising.transforms.abstract.AbstractTransform*

Apply one of the given transforms.

Parameters

- ***transforms** – transforms to choose from
- **weights** (Optional[Sequence[float]]) – additional weights for transforms
- **p** (float) – probability that one transform i applied
- **transform_call** (Callable[[Any, Callable], Any]) – function which determines how transforms are called. By default Mappings and Sequences are unpacked during the transform.

forward(***data*)

Implement transform functionality here

Parameters ****data** – dict with data

Returns dict with transformed data

Return type dict

10.2.4 dict_call

rising.transforms.compose.**dict_call**(*batch*, *transform*)

Unpacks the dict for every transformation

Parameters

- **batch** (dict) – current batch which is passed to transform
- **transform** (Callable) – transform to perform

Returns transformed batch

Return type Any

10.3 Affine Transforms

class rising.transforms.affine.**Affine**(*matrix=None*, *keys=('data',)*, *grad=False*, *output_size=None*, *adjust_size=False*, *interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*, *reverse_order=False*, *per_sample=True*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Class Performing an Affine Transformation on a given sample dict. The transformation will be applied to all the dict-entries specified in `keys`.

Parameters

- **matrix** (Union[Tensor, Sequence[Sequence[float]], None]) – if given, overwrites the parameters for `scale`, :attr:rotation` and `translation`. Should be a matrix of shape [(BATCHSIZE,) NDIM, NDIM(+1)] This matrix represents the whole transformation matrix
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **output_size** (Optional[tuple]) – if given, this will be the resulting image size. Defaults to `None`
- **adjust_size** (bool) – if True, the resulting image size will be calculated dynamically to ensure that the whole image fits.
- **interpolation_mode** (str) – interpolation mode to calculate output values `'bilinear'` | `'nearest'`. Default: `'bilinear'`
- **padding_mode** (str) – padding mode for outside grid values `'zeros'` | `'border'` | `'reflection'`. Default: `'zeros'`
- **align_corners** (bool) – Geometrically, we consider the pixels of the input as squares rather than points. If set to True, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to False, they are instead considered

as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic.

- **reverse_order** (bool) – reverses the coordinate order of the transformation to conform to the pytorch convention: transformation params order [W,H(,D)] and batch order [(D,)H,W]
- **per_sample** (bool) – sample different values for each element in the batch. The transform is still applied in a batched wise fashion.
- ****kwargs** – additional keyword arguments passed to the affine transform

assemble_matrix(***data*)

Assembles the matrix (and takes care of batching and having it on the right device and in the correct dtype and dimensionality).

Parameters ****data** – the data to be transformed. Will be used to determine batchsize, dimensionality, dtype and device

Returns the (batched) transformation matrix

Return type torch.Tensor

forward(***data*)

Assembles the matrix and applies it to the specified sample-entities.

Parameters ****data** – the data to transform

Returns dictionary containing the transformed data

Return type dict

class rising.transforms.affine.**BaseAffine**(*scale=None*, *rotation=None*, *translation=None*, *degree=False*, *image_transform=True*, *keys=('data',)*, *grad=False*, *output_size=None*, *adjust_size=False*, *interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*, *reverse_order=False*, *per_sample=True*, ***kwargs*)

Bases: *rising.transforms.affine.Affine*

Class performing a basic Affine Transformation on a given sample dict. The transformation will be applied to all the dict-entries specified in `keys`.

Parameters

- **scale** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*], None]) – the scale factor(s). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per dimension, which will be replicated for all batch samples * None will be treated as a scaling factor of 1
- **rotation** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*], None]) – the rotation factor(s). The rotation is performed in consecutive order axis0 -> axis1 (-> axis 2). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per dimension, which will be replicated for all batch samples * None will be treated as a rotation angle of 0
- **translation** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*], None]) – torch.Tensor, int, float the translation offset(s) relative to image (should be in the range [0, 1]). Supported are: * a single parameter (as float or int), which will be replicated for all

dimensions and batch samples * a parameter per dimension, which will be replicated for all batch samples * None will be treated as a translation offset of 0

- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **degree** (bool) – whether the given rotation(s) are in degrees. Only valid for rotation parameters, which aren't passed as full transformation matrix.
- **output_size** (Optional[tuple]) – if given, this will be the resulting image size. Defaults to None
- **adjust_size** (bool) – if True, the resulting image size will be calculated dynamically to ensure that the whole image fits.
- **interpolation_mode** (str) – interpolation mode to calculate output values 'bilinear'|'nearest'. Default: 'bilinear'
- **padding_mode** (str) – padding mode for outside grid values 'zeros'|'border'|'reflection'. Default: 'zeros'
- **align_corners** (bool) – Geometrically, we consider the pixels of the input as squares rather than points. If set to True, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to False, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic.
- **reverse_order** (bool) – reverses the coordinate order of the transformation to conform to the pytorch convention: transformation params order [W,H(,D)] and batch order [(D,)H,W]
- **per_sample** (bool) – sample different values for each element in the batch. The transform is still applied in a batched wise fashion.
- ****kwargs** – additional keyword arguments passed to the affine transform

assemble_matrix(***data*)

Assembles the matrix (and takes care of batching and having it on the right device and in the correct dtype and dimensionality).

Parameters ****data** – the data to be transformed. Will be used to determine batchsize, dimensionality, dtype and device

Returns the (batched) transformation matrix

Return type torch.Tensor

sample_for_batch(*name*, *batchsize*)

Sample elements for batch

Parameters

- **name** (str) – name of parameter
- **batchsize** (int) – batch size

Returns sampled elements

Return type Optional[Union[Any, Sequence[Any]]]

class rising.transforms.affine.**StackedAffine**(**transforms*, *keys=('data',)*, *grad=False*, *output_size=None*, *adjust_size=False*, *interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*, *reverse_order=False*, ***kwargs*)

Bases: *rising.transforms.affine.Affine*

Class to stack multiple affines with dynamic ensembling by matrix multiplication to avoid multiple interpolations.

Parameters

- **transforms** (Union[*Affine*, Sequence[Union[Sequence[*Affine*], *Affine*]]]) – the transforms to stack. Each transform must have a function called `assemble_matrix`, which is called to dynamically assemble stacked matrices. Afterwards these transformations are stacked by matrix-multiplication to only perform a single interpolation
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **output_size** (Optional[tuple]) – if given, this will be the resulting image size. Defaults to `None`
- **adjust_size** (bool) – if True, the resulting image size will be calculated dynamically to ensure that the whole image fits.
- **interpolation_mode** (str) – interpolation mode to calculate output values `'bilinear'|'nearest'`. Default: `'bilinear'`
- **padding_mode** (str) – padding mode for outside grid values `'zeros'|'border'|'reflection'`. Default: `'zeros'`
- **align_corners** (bool) – Geometrically, we consider the pixels of the input as squares rather than points. If set to True, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to False, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic.
- **reverse_order** (bool) – reverses the coordinate order of the transformation to conform to the pytorch convention: transformation params order [W,H(,D)] and batch order [(D,)H,W]
- ****kwargs** – additional keyword arguments passed to the affine transform

assemble_matrix(***data*)

Handles the matrix assembly and stacking

Parameters ****data** – the data to be transformed. Will be used to determine batchsize, dimensionality, dtype and device

Returns the (batched) transformation matrix

Return type torch.Tensor

class rising.transforms.affine.**Rotate**(*rotation*, *keys=('data',)*, *grad=False*, *degree=False*, *output_size=None*, *adjust_size=False*, *interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*, *reverse_order=False*, ***kwargs*)

Bases: *rising.transforms.affine.BaseAffine*

Class Performing a Rotation-OnlyAffine Transformation on a given sample dict. The rotation is applied in consecutive order: rot axis 0 -> rot axis 1 -> rot axis 2 The transformation will be applied to all the dict-entries specified in `keys`.

Parameters

- **rotation** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – the rotation factor(s). The rotation is performed in consecutive order axis0 -> axis1 (-> axis 2). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per dimension, which will be replicated for all batch samples * `None` will be treated as a rotation angle of 0
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **degree** (bool) – whether the given rotation(s) are in degrees. Only valid for rotation parameters, which aren't passed as full transformation matrix.
- **output_size** (Optional[tuple]) – if given, this will be the resulting image size. Defaults to `None`
- **adjust_size** (bool) – if True, the resulting image size will be calculated dynamically to ensure that the whole image fits.
- **interpolation_mode** (str) – interpolation mode to calculate output values `'bilinear'|'nearest'`. Default: `'bilinear'`
- **padding_mode** (str) – padding mode for outside grid values `'zeros'|'border'|'reflection'`. Default: `'zeros'`
- **align_corners** (bool) – Geometrically, we consider the pixels of the input as squares rather than points. If set to True, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to False, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic.
- **reverse_order** (bool) – reverses the coordinate order of the transformation to conform to the pytorch convention: transformation params order [W,H(,D)] and batch order [(D,)H,W]
- ****kwargs** – additional keyword arguments passed to the affine transform

class rising.transforms.affine.**Scale**(*scale*, *keys=('data',)*, *grad=False*, *output_size=None*, *adjust_size=False*, *interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*, *reverse_order=False*, ***kwargs*)

Bases: *rising.transforms.affine.BaseAffine*

Class Performing a Scale-Only Affine Transformation on a given sample dict. The transformation will be applied to all the dict-entries specified in `keys`.

Parameters

- **scale** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – torch.Tensor, int, float, optional the scale factor(s). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per dimension, which will be replicated for all batch samples * None will be treated as a scaling factor of 1
- **keys** (Sequence) – Sequence keys which should be augmented

- **grad** (bool) – bool enable gradient computation inside transformation
- **degree** – bool whether the given rotation(s) are in degrees. Only valid for rotation parameters, which aren't passed as full transformation matrix.
- **output_size** (Optional[tuple]) – Iterable if given, this will be the resulting image size. Defaults to None
- **adjust_size** (bool) – bool if True, the resulting image size will be calculated dynamically to ensure that the whole image fits.
- **interpolation_mode** (str) – str interpolation mode to calculate output values 'bilinear' | 'nearest'. Default: 'bilinear'
- **padding_mode** (str) – padding mode for outside grid values 'zeros' | 'border' | 'reflection'. Default: 'zeros'
- **align_corners** (bool) – bool Geometrically, we consider the pixels of the input as squares rather than points. If set to True, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to False, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic.
- **reverse_order** (bool) – bool reverses the coordinate order of the transformation to conform to the pytorch convention: transformation params order [W,H(,D)] and batch order [(D,)H,W]
- ****kwargs** – additional keyword arguments passed to the affine transform

class rising.transforms.affine.**Translate**(*translation*, *keys=('data',)*, *grad=False*, *output_size=None*, *adjust_size=False*, *interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*, *unit='pixel'*, *reverse_order=False*, ***kwargs*)

Bases: *rising.transforms.affine.BaseAffine*

Class Performing an Translation-Only Affine Transformation on a given sample dict. The transformation will be applied to all the dict-entries specified in keys.

Parameters

- **translation** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – torch.Tensor, int, float the translation offset(s) relative to image (should be in the range [0, 1]). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per dimension, which will be replicated for all batch samples * None will be treated as a translation offset of 0
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **output_size** (Optional[tuple]) – if given, this will be the resulting image size. Defaults to None
- **adjust_size** (bool) – if True, the resulting image size will be calculated dynamically to ensure that the whole image fits.
- **interpolation_mode** (str) – interpolation mode to calculate output values 'bilinear' | 'nearest'. Default: 'bilinear'
- **padding_mode** (str) – padding mode for outside grid values 'zeros' | 'border' | 'reflection'. Default: 'zeros'

- **align_corners** (bool) – Geometrically, we consider the pixels of the input as squares rather than points. If set to True, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to False, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic.
- **unit** (str) – defines the unit of the translation. Either `relative' to the image size or in `pixel'
- **reverse_order** (bool) – reverses the coordinate order of the transformation to conform to the pytorch convention: transformation params order [W,H(,D)] and batch order [(D,)H,W]
- ****kwargs** – additional keyword arguments passed to the affine transform

assemble_matrix(***data*)

Assembles the matrix (and takes care of batching and having it on the right device and in the correct dtype and dimensionality).

Parameters ****data** – the data to be transformed. Will be used to determine batchsize, dimensionality, dtype and device

Returns the (batched) transformation matrix [N, NDIM, NDIM]

Return type torch.Tensor

class rising.transforms.affine.**Resize**(*size*, *keys=('data',)*, *grad=False*, *interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*, *reverse_order=False*, ***kwargs*)

Bases: *rising.transforms.affine.Scale*

Class Performing a Resizing Affine Transformation on a given sample dict. The transformation will be applied to all the dict-entries specified in `keys`.

Parameters

- **size** (Union[int, Tuple[int]]) – the target size. If int, this will be repeated for all the dimensions
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **interpolation_mode** (str) – nterpolation mode to calculate output values 'bilinear' | 'nearest'. Default: 'bilinear'
- **padding_mode** (str) – padding mode for outside grid values 'zeros' | 'border' | 'reflection'. Default: 'zeros'
- **align_corners** (bool) – Geometrically, we consider the pixels of the input as squares rather than points. If set to True, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to False, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic.
- **reverse_order** (bool) – reverses the coordinate order of the transformation to conform to the pytorch convention: transformation params order [W,H(,D)] and batch order [(D,)H,W]
- ****kwargs** – additional keyword arguments passed to the affine transform

Notes

The offsets for shifting back and to origin are calculated on the entry matching the first item iin `keys` for each batch

assemble_matrix(***data*)
Handles the matrix assembly and calculates the scale factors for resizing

Parameters ****data** – the data to be transformed. Will be used to determine batchsize, dimensionality, dtype and device

Returns the (batched) transformation matrix

Return type torch.Tensor

10.3.1 Affine

class rising.transforms.affine.**Affine**(*matrix=None*, *keys=('data',)*, *grad=False*, *output_size=None*, *adjust_size=False*, *interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*, *reverse_order=False*, *per_sample=True*, ***kwargs*)
Bases: *rising.transforms.abstract.BaseTransform*

Class Performing an Affine Transformation on a given sample dict. The transformation will be applied to all the dict-entries specified in `keys`.

Parameters

- **matrix** (Union[Tensor, Sequence[Sequence[float]], None]) – if given, overwrites the parameters for `scale`, :attr:rotation` and `translation`. Should be a matrix of shape [(BATCHSIZE,) NDIM, NDIM(+1)] This matrix represents the whole transformation matrix
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **output_size** (Optional[tuple]) – if given, this will be the resulting image size. Defaults to None
- **adjust_size** (bool) – if True, the resulting image size will be calculated dynamically to ensure that the whole image fits.
- **interpolation_mode** (str) – interpolation mode to calculate output values `'bilinear'` | `'nearest'`. Default: `'bilinear'`
- **padding_mode** (str) – padding mode for outside grid values `'zeros'` | `'border'` | `'reflection'`. Default: `'zeros'`
- **align_corners** (bool) – Geometrically, we consider the pixels of the input as squares rather than points. If set to True, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to False, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic.
- **reverse_order** (bool) – reverses the coordinate order of the transformation to conform to the pytorch convention: transformation params order [W,H(,D)] and batch order [(D,)H,W]

- **per_sample** (bool) – sample different values for each element in the batch. The transform is still applied in a batched wise fashion.
- ****kwargs** – additional keyword arguments passed to the affine transform

assemble_matrix(***data*)

Assembles the matrix (and takes care of batching and having it on the right device and in the correct dtype and dimensionality).

Parameters ****data** – the data to be transformed. Will be used to determine batchsize, dimensionality, dtype and device

Returns the (batched) transformation matrix

Return type torch.Tensor

forward(***data*)

Assembles the matrix and applies it to the specified sample-entities.

Parameters ****data** – the data to transform

Returns dictionary containing the transformed data

Return type dict

10.3.2 StackedAffine

class rising.transforms.affine.**StackedAffine**(**transforms*, *keys=('data',)*, *grad=False*, *output_size=None*, *adjust_size=False*, *interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*, *reverse_order=False*, ***kwargs*)

Bases: *rising.transforms.affine.Affine*

Class to stack multiple affines with dynamic ensembling by matrix multiplication to avoid multiple interpolations.

Parameters

- **transforms** (Union[*Affine*, Sequence[Union[Sequence[*Affine*], *Affine*]]]) – the transforms to stack. Each transform must have a function called `assemble_matrix`, which is called to dynamically assemble stacked matrices. Afterwards these transformations are stacked by matrix-multiplication to only perform a single interpolation
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **output_size** (Optional[tuple]) – if given, this will be the resulting image size. Defaults to `None`
- **adjust_size** (bool) – if True, the resulting image size will be calculated dynamically to ensure that the whole image fits.
- **interpolation_mode** (str) – interpolation mode to calculate output values `'bilinear'|'nearest'`. Default: `'bilinear'`
- **padding_mode** (str) – padding mode for outside grid values `'zeros'|'border'|'reflection'`. Default: `'zeros'`

- **align_corners** (bool) – Geometrically, we consider the pixels of the input as squares rather than points. If set to True, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to False, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic.
- **reverse_order** (bool) – reverses the coordinate order of the transformation to conform to the pytorch convention: transformation params order [W,H(,D)] and batch order [(D,)H,W]
- ****kwargs** – additional keyword arguments passed to the affine transform

assemble_matrix(***data*)

Handles the matrix assembly and stacking

Parameters ****data** – the data to be transformed. Will be used to determine batchsize, dimensionality, dtype and device

Returns the (batched) transformation matrix

Return type torch.Tensor

10.3.3 BaseAffine

class rising.transforms.affine.**BaseAffine**(*scale=None*, *rotation=None*, *translation=None*, *degree=False*, *image_transform=True*, *keys=('data',)*, *grad=False*, *output_size=None*, *adjust_size=False*, *interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*, *reverse_order=False*, *per_sample=True*, ***kwargs*)

Bases: *rising.transforms.affine.Affine*

Class performing a basic Affine Transformation on a given sample dict. The transformation will be applied to all the dict-entries specified in `keys`.

Parameters

- **scale** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*], None]) – the scale factor(s). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per dimension, which will be replicated for all batch samples * None will be treated as a scaling factor of 1
- **rotation** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*], None]) – the rotation factor(s). The rotation is performed in consecutive order axis0 -> axis1 (-> axis 2). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per dimension, which will be replicated for all batch samples * None will be treated as a rotation angle of 0
- **translation** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*], None]) – torch.Tensor, int, float the translation offset(s) relative to image (should be in the range [0, 1]). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per dimension, which will be replicated for all batch samples * None will be treated as a translation offset of 0
- **keys** (Sequence) – keys which should be augmented

- **grad** (bool) – enable gradient computation inside transformation
- **degree** (bool) – whether the given rotation(s) are in degrees. Only valid for rotation parameters, which aren't passed as full transformation matrix.
- **output_size** (Optional[tuple]) – if given, this will be the resulting image size. Defaults to None
- **adjust_size** (bool) – if True, the resulting image size will be calculated dynamically to ensure that the whole image fits.
- **interpolation_mode** (str) – interpolation mode to calculate output values 'bilinear'|'nearest'. Default: 'bilinear'
- **padding_mode** (str) – padding mode for outside grid values 'zeros'|'border'|'reflection'. Default: 'zeros'
- **align_corners** (bool) – Geometrically, we consider the pixels of the input as squares rather than points. If set to True, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to False, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic.
- **reverse_order** (bool) – reverses the coordinate order of the transformation to conform to the pytorch convention: transformation params order [W,H(,D)] and batch order [(D,)H,W]
- **per_sample** (bool) – sample different values for each element in the batch. The transform is still applied in a batched wise fashion.
- ****kwargs** – additional keyword arguments passed to the affine transform

assemble_matrix(***data*)

Assembles the matrix (and takes care of batching and having it on the right device and in the correct dtype and dimensionality).

Parameters ****data** – the data to be transformed. Will be used to determine batchsize, dimensionality, dtype and device

Returns the (batched) transformation matrix

Return type torch.Tensor

sample_for_batch(*name*, *batchsize*)

Sample elements for batch

Parameters

- **name** (str) – name of parameter
- **batchsize** (int) – batch size

Returns sampled elements

Return type Optional[Union[Any, Sequence[Any]]]

10.3.4 Rotate

class rising.transforms.affine.**Rotate**(*rotation*, *keys=('data',)*, *grad=False*, *degree=False*, *output_size=None*, *adjust_size=False*, *interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*, *reverse_order=False*, ***kwargs*)

Bases: *rising.transforms.affine.BaseAffine*

Class Performing a Rotation-OnlyAffine Transformation on a given sample dict. The rotation is applied in consecutive order: rot axis 0 -> rot axis 1 -> rot axis 2 The transformation will be applied to all the dict-entries specified in `keys`.

Parameters

- **rotation** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – the rotation factor(s). The rotation is performed in consecutive order axis0 -> axis1 (-> axis 2). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per dimension, which will be replicated for all batch samples * `None` will be treated as a rotation angle of 0
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **degree** (bool) – whether the given rotation(s) are in degrees. Only valid for rotation parameters, which aren't passed as full transformation matrix.
- **output_size** (Optional[tuple]) – if given, this will be the resulting image size. Defaults to `None`
- **adjust_size** (bool) – if True, the resulting image size will be calculated dynamically to ensure that the whole image fits.
- **interpolation_mode** (str) – interpolation mode to calculate output values `'bilinear'` | `'nearest'`. Default: `'bilinear'`
- **padding_mode** (str) – padding mode for outside grid values `'zeros'` | `'border'` | `'reflection'`. Default: `'zeros'`
- **align_corners** (bool) – Geometrically, we consider the pixels of the input as squares rather than points. If set to True, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to False, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic.
- **reverse_order** (bool) – reverses the coordinate order of the transformation to conform to the pytorch convention: transformation params order [W,H(,D)] and batch order [(D,)H,W]
- ****kwargs** – additional keyword arguments passed to the affine transform

10.3.5 Translate

class rising.transforms.affine.**Translate**(*translation*, *keys=('data',)*, *grad=False*, *output_size=None*, *adjust_size=False*, *interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*, *unit='pixel'*, *reverse_order=False*, ***kwargs*)

Bases: *rising.transforms.affine.BaseAffine*

Class Performing an Translation-Only Affine Transformation on a given sample dict. The transformation will be applied to all the dict-entries specified in `keys`.

Parameters

- **translation** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – torch.Tensor, int, float the translation offset(s) relative to image (should be in the range [0, 1]). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per dimension, which will be replicated for all batch samples * None will be treated as a translation offset of 0
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **output_size** (Optional[tuple]) – if given, this will be the resulting image size. Defaults to `None`
- **adjust_size** (bool) – if True, the resulting image size will be calculated dynamically to ensure that the whole image fits.
- **interpolation_mode** (str) – interpolation mode to calculate output values `'bilinear'|'nearest'`. Default: `'bilinear'`
- **padding_mode** (str) – padding mode for outside grid values `'zeros'|'border'|'reflection'`. Default: `'zeros'`
- **align_corners** (bool) – Geometrically, we consider the pixels of the input as squares rather than points. If set to True, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to False, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic.
- **unit** (str) – defines the unit of the translation. Either `` `relative' `` to the image size or in `` `pixel' ``
- **reverse_order** (bool) – reverses the coordinate order of the transformation to conform to the pytorch convention: transformation params order [W,H(,D)] and batch order [(D,)H,W]
- ****kwargs** – additional keyword arguments passed to the affine transform

assemble_matrix(***data*)

Assembles the matrix (and takes care of batching and having it on the right device and in the correct dtype and dimensionality).

Parameters ****data** – the data to be transformed. Will be used to determine batchsize, dimensionality, dtype and device

Returns the (batched) transformation matrix [N, NDIM, NDIM]

Return type torch.Tensor

10.3.6 Scale

class rising.transforms.affine.**Scale**(*scale*, *keys=('data',)*, *grad=False*, *output_size=None*, *adjust_size=False*, *interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*, *reverse_order=False*, ***kwargs*)

Bases: *rising.transforms.affine.BaseAffine*

Class Performing a Scale-Only Affine Transformation on a given sample dict. The transformation will be applied to all the dict-entries specified in `keys`.

Parameters

- **scale** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – torch.Tensor, int, float, optional the scale factor(s). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per dimension, which will be replicated for all batch samples * None will be treated as a scaling factor of 1
- **keys** (Sequence) – Sequence keys which should be augmented
- **grad** (bool) – bool enable gradient computation inside transformation
- **degree** – bool whether the given rotation(s) are in degrees. Only valid for rotation parameters, which aren't passed as full transformation matrix.
- **output_size** (Optional[tuple]) – Iterable if given, this will be the resulting image size. Defaults to `None`
- **adjust_size** (bool) – bool if True, the resulting image size will be calculated dynamically to ensure that the whole image fits.
- **interpolation_mode** (str) – str interpolation mode to calculate output values 'bilinear' | 'nearest'. Default: 'bilinear'
- **padding_mode** (str) – padding mode for outside grid values 'zeros' | 'border' | 'reflection'. Default: 'zeros'
- **align_corners** (bool) – bool Geometrically, we consider the pixels of the input as squares rather than points. If set to True, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to False, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic.
- **reverse_order** (bool) – bool reverses the coordinate order of the transformation to conform to the pytorch convention: transformation params order [W,H(,D)] and batch order [(D,)H,W]
- ****kwargs** – additional keyword arguments passed to the affine transform

10.3.7 Resize

class rising.transforms.affine.**Resize**(*size*, *keys=('data',)*, *grad=False*, *interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*, *reverse_order=False*, ***kwargs*)

Bases: *rising.transforms.affine.Scale*

Class Performing a Resizing Affine Transformation on a given sample dict. The transformation will be applied to all the dict-entries specified in `keys`.

Parameters

- **size** (Union[int, Tuple[int]]) – the target size. If int, this will be repeated for all the dimensions
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **interpolation_mode** (str) – nterpolation mode to calculate output values 'bilinear' | 'nearest'. Default: 'bilinear'
- **padding_mode** (str) – padding mode for outside grid values 'zeros' | 'border' | 'reflection'. Default: 'zeros'
- **align_corners** (bool) – Geometrically, we consider the pixels of the input as squares rather than points. If set to True, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to False, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic.
- **reverse_order** (bool) – reverses the coordinate order of the transformation to conform to the pytorch convention: transformation params order [W,H(,D)] and batch order [(D,)H,W]
- ****kwargs** – additional keyword arguments passed to the affine transform

Notes

The offsets for shifting back and to origin are calculated on the entry matching the first item iin `keys` for each batch

assemble_matrix(***data*)

Handles the matrix assembly and calculates the scale factors for resizing

Parameters ****data** – the data to be transformed. Will be used to determine batchsize, dimensionality, dtype and device

Returns the (batched) transformation matrix

Return type torch.Tensor

10.4 Channel Transforms

class rising.transforms.channel.**OneHot**(*num_classes*, *keys=('seg',)*, *dtype=None*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Convert to one hot encoding. One hot encoding is applied in first dimension which results in shape N x Num-Classes x [same as input] while input is expected to have shape N x 1 x [arbitrary additional dimensions]

Parameters

- **num_classes** (int) – number of classes. If num_classes is None, the number of classes is automatically determined from the current batch (by using the max of the current batch and assuming a consecutive order from zero)
- **dtype** (Optional[dtype]) – optionally changes the dtype of the onehot encoding
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to one_hot_batch()

Warning: Input tensor needs to be of type torch.long. This could be achieved by applying *TenorOp("long", keys=("seg",))*.

class rising.transforms.channel.**ArgMax**(*dim*, *keepdim=True*, *keys=('seg',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Compute argmax along given dimension. Can be used to revert OneHot encoding.

Parameters

- **dim** (int) – dimension to apply argmax
- **keepdim** (bool) – whether the output tensor has dim retained or not
- **dtype** – optionally changes the dtype of the onehot encoding
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to one_hot_batch()

Warnings The output of the argmax function is always a tensor of dtype long.

10.4.1 OneHot

class rising.transforms.channel.**OneHot**(*num_classes*, *keys=('seg',)*, *dtype=None*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Convert to one hot encoding. One hot encoding is applied in first dimension which results in shape N x Num-Classes x [same as input] while input is expected to have shape N x 1 x [arbitrary additional dimensions]

Parameters

- **num_classes** (int) – number of classes. If num_classes is None, the number of classes is automatically determined from the current batch (by using the max of the current batch and assuming a consecutive order from zero)
- **dtype** (Optional[dtype]) – optionally changes the dtype of the onehot encoding
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to one_hot_batch()

Warning: Input tensor needs to be of type torch.long. This could be achieved by applying *TenorOp("long", keys=("seg",))*.

10.4.2 ArgMax

class rising.transforms.channel.**ArgMax**(*dim*, *keepdim=True*, *keys=('seg',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Compute argmax along given dimension. Can be used to revert OneHot encoding.

Parameters

- **dim** (int) – dimension to apply argmax
- **keepdim** (bool) – whether the output tensor has dim retained or not
- **dtype** – optionally changes the dtype of the onehot encoding
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to one_hot_batch()

Warnings The output of the argmax function is always a tensor of dtype long.

10.5 Cropping Transforms

class rising.transforms.crop.**CenterCrop**(*size*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Parameters

- **size** (Union[int, Sequence, *AbstractParameter*]) – size of crop
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

class rising.transforms.crop.**RandomCrop**(*size*, *dist=0*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransformSeeded*

Parameters

- **size** (Union[int, Sequence, *AbstractParameter*]) – size of crop
- **dist** (Union[int, Sequence, *AbstractParameter*]) – minimum distance to border. By default zero
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

10.5.1 CenterCrop

class rising.transforms.crop.**CenterCrop**(*size*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Parameters

- **size** (Union[int, Sequence, *AbstractParameter*]) – size of crop
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

10.5.2 RandomCrop

class rising.transforms.crop.**RandomCrop**(*size*, *dist=0*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransformSeeded*

Parameters

- **size** (Union[int, Sequence, *AbstractParameter*]) – size of crop
- **dist** (Union[int, Sequence, *AbstractParameter*]) – minimum distance to border. By default zero
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

10.6 Format Transforms

class rising.transforms.format.**MapToSeq**(**keys*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.AbstractTransform*

Convert dict to sequence

Parameters

- **keys** – keys which are mapped into sequence.
- **grad** (bool) – enable gradient computation inside transformation
- **kwargs** (****) – additional keyword arguments passed to superclass

forward(***data*)
Convert input

Parameters **data** – input dict

Returns mapped data

Return type tuple

class rising.transforms.format.**SeqToMap**(**keys*, *grad=False*, ***kwargs*)
Bases: *rising.transforms.abstract.AbstractTransform*

Convert sequence to dict

Parameters

- **keys** – keys which are mapped into dict.
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – additional keyword arguments passed to superclass

forward(**data*, ***kwargs*)
Convert input

Parameters **data** – input tuple

Returns mapped data

Return type dict

class rising.transforms.format.**PopKeys**(*keys*, *return_popped=False*)
Bases: *rising.transforms.abstract.AbstractTransform*

Pops keys from a given data dict

Parameters

- **keys** (Union[Callable, Sequence]) – if callable it must return a boolean for each key indicating whether it should be popped from the dict. if sequence of strings, the strings shall be the keys to be popped
- **return_popped** (bool) – whether to also return the popped values (default: False)

forward(***data*)
Implement transform functionality here

Parameters ****data** – dict with data

Returns dict with transformed data

Return type dict

class rising.transforms.format.**FilterKeys**(*keys*, *return_popped=False*)
Bases: *rising.transforms.abstract.AbstractTransform*

Filters keys from a given data dict

Parameters

- **keys** (Union[Callable, Sequence]) – if callable it must return a boolean for each key indicating whether it should be retained in the dict. if sequence of strings, the strings shall be the keys to be retained
- **return_popped** (bool) – whether to also return the popped values (default: False)

forward(***data*)
Implement transform functionality here

Parameters ****data** – dict with data

Returns dict with transformed data

Return type dict

class rising.transforms.format.**RenameKeys**(*keys*)
Bases: *rising.transforms.abstract.AbstractTransform*

Rename keys inside batch

Parameters **keys** (Mapping[Hashable, Hashable]) – keys of mapping define current name and items define the new names

forward(***data*)
Implement transform functionality here

Parameters ****data** – dict with data

Returns dict with transformed data

Return type dict

10.6.1 MapToSeq

class rising.transforms.format.**MapToSeq**(**keys*, *grad=False*, ***kwargs*)
Bases: *rising.transforms.abstract.AbstractTransform*

Convert dict to sequence

Parameters

- **keys** – keys which are mapped into sequence.
- **grad** (bool) – enable gradient computation inside transformation
- **kwargs** (*) – additional keyword arguments passed to superclass

forward(***data*)
Convert input

Parameters **data** – input dict

Returns mapped data

Return type tuple

10.6.2 SeqToMap

class rising.transforms.format.**SeqToMap**(**keys*, *grad=False*, ***kwargs*)
Bases: *rising.transforms.abstract.AbstractTransform*

Convert sequence to dict

Parameters

- **keys** – keys which are mapped into dict.
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – additional keyword arguments passed to superclass

forward(**data*, ***kwargs*)
Convert input

Parameters **data** – input tuple

Returns mapped data

Return type dict

10.6.3 PopKeys

class rising.transforms.format.**PopKeys**(*keys*, *return_popped=False*)

Bases: *rising.transforms.abstract.AbstractTransform*

Pops keys from a given data dict

Parameters

- **keys** (Union[Callable, Sequence]) – if callable it must return a boolean for each key indicating whether it should be popped from the dict. if sequence of strings, the strings shall be the keys to be popped
- **return_popped** (bool) – whether to also return the popped values (default: False)

forward(***data*)

Implement transform functionality here

Parameters ****data** – dict with data

Returns dict with transformed data

Return type dict

10.6.4 FilterKeys

class rising.transforms.format.**FilterKeys**(*keys*, *return_popped=False*)

Bases: *rising.transforms.abstract.AbstractTransform*

Filters keys from a given data dict

Parameters

- **keys** (Union[Callable, Sequence]) – if callable it must return a boolean for each key indicating whether it should be retained in the dict. if sequence of strings, the strings shall be the keys to be retained
- **return_popped** (bool) – whether to also return the popped values (default: False)

forward(***data*)

Implement transform functionality here

Parameters ****data** – dict with data

Returns dict with transformed data

Return type dict

10.6.5 RenameKeys

class rising.transforms.format.**RenameKeys**(*keys*)
Bases: *rising.transforms.abstract.AbstractTransform*

Rename keys inside batch

Parameters **keys** (Mapping[Hashable, Hashable]) – keys of mapping define current name and items define the new names

forward(***data*)
Implement transform functionality here

Parameters ****data** – dict with data

Returns dict with transformed data

Return type dict

10.7 Intensity Transforms

class rising.transforms.intensity.**Clamp**(*min*, *max*, *keys=('data',)*, *grad=False*, ***kwargs*)
Bases: *rising.transforms.abstract.BaseTransform*

Apply augment_fn to keys

Parameters

- **min** (Union[float, *AbstractParameter*]) – minimal value
- **max** (Union[float, *AbstractParameter*]) – maximal value
- **keys** (Sequence) – the keys corresponding to the values to clamp
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

class rising.transforms.intensity.**NormRange**(*min*, *max*, *keys=('data',)*, *per_channel=True*, *grad=False*, ***kwargs*)
Bases: *rising.transforms.abstract.PerSampleTransform*

Parameters

- **min** (Union[float, *AbstractParameter*]) – minimal value
- **max** (Union[float, *AbstractParameter*]) – maximal value
- **keys** (Sequence) – keys to normalize
- **per_channel** (bool) – normalize per channel
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to normalization function

class rising.transforms.intensity.**NormMinMax**(*keys=('data',)*, *per_channel=True*, *grad=False*, *eps=1e-08*, ***kwargs*)
Bases: *rising.transforms.abstract.PerSampleTransform*

Norm to [0, 1]

Parameters

- **keys** (Sequence) – keys to normalize

- **per_channel** (bool) – normalize per channel
- **grad** (bool) – enable gradient computation inside transformation
- **eps** (Optional[float]) – small constant for numerical stability. If None, no factor constant will be added
- ****kwargs** – keyword arguments passed to normalization function

class rising.transforms.intensity.**NormZeroMeanUnitStd**(*keys=('data',)*, *per_channel=True*, *grad=False*, *eps=1e-08*, ***kwargs*)

Bases: *rising.transforms.abstract.PerSampleTransform*

Normalize mean to zero and std to one

Parameters

- **keys** (Sequence) – keys to normalize
- **per_channel** (bool) – normalize per channel
- **grad** (bool) – enable gradient computation inside transformation
- **eps** (Optional[float]) – small constant for numerical stability. If None, no factor constant will be added
- ****kwargs** – keyword arguments passed to normalization function

class rising.transforms.intensity.**NormMeanStd**(*mean*, *std*, *keys=('data',)*, *per_channel=True*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.PerSampleTransform*

Normalize mean and std with provided values

Parameters

- **mean** (Union[float, Sequence[float]]) – used for mean normalization
- **std** (Union[float, Sequence[float]]) – used for std normalization
- **keys** (Sequence[str]) – keys to normalize
- **per_channel** (bool) – normalize per channel
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to normalization function

class rising.transforms.intensity.**Noise**(*noise_type*, *per_channel=False*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.PerChannelTransform*

Add noise to data

Warning: This transform will apply different noise patterns to different keys.

Parameters

- **noise_type** (str) – supports all inplace functions of a torch.Tensor
- **per_channel** (bool) – enable transformation per channel
- **keys** (Sequence) – keys to normalize

- **grad** (bool) – enable gradient computation inside transformation
- **kwargs** – keyword arguments passed to noise function

See also:

`torch.Tensor.normal_()`, `torch.Tensor.exponential_()`

class rising.transforms.intensity.**GaussianNoise**(*mean*, *std*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.intensity.Noise*

Add gaussian noise to data

Warning: This transform will apply different noise patterns to different keys.

Parameters

- **mean** (float) – mean of normal distribution
- **std** (float) – std of normal distribution
- **keys** (Sequence) – keys to normalize
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to noise function

class rising.transforms.intensity.**ExponentialNoise**(*lambd*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.intensity.Noise*

Add exponential noise to data

Warning: This transform will apply different noise patterns to different keys.

Parameters

- **lambd** (float) – lambda of exponential distribution
- **keys** (Sequence) – keys to normalize
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to noise function

class rising.transforms.intensity.**GammaCorrection**(*gamma*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Apply Gamma correction

Parameters

- **gamma** (Union[float, *AbstractParameter*]) – define gamma
- **keys** (Sequence) – keys to normalize
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to superclass

class rising.transforms.intensity.**RandomValuePerChannel**(*augment_fn*, *random_sampler*, *per_channel=False*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.PerChannelTransform*

Apply augmentations which take random values as input by keyword `value`

Warning: This transform will apply different values to different keys.

Parameters

- **augment_fn** (callable) – augmentation function
- **random_mode** – specifies distribution which should be used to sample additive value. All function from python's random module are supported
- **random_args** – positional arguments passed for random function
- **per_channel** (bool) – enable transformation per channel
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

forward(***data*)

Perform Augmentation.

Parameters **data** – dict with data

Returns augmented data

Return type dict

class rising.transforms.intensity.**RandomAddValue**(*random_sampler*, *per_channel=False*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.intensity.RandomValuePerChannel*

Increase values additively

Warning: This transform will apply different values to different keys.

Parameters

- **random_sampler** (*AbstractParameter*) – specify values to add
- **per_channel** (bool) – enable transformation per channel
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

class rising.transforms.intensity.**RandomScaleValue**(*random_sampler*, *per_channel=False*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.intensity.RandomValuePerChannel*

Scale Values

Warning: This transform will apply different values to different keys.

Parameters

- **random_sampler** (*AbstractParameter*) – specify values to add
- **per_channel** (bool) – enable transformation per channel
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

10.7.1 Clamp

class rising.transforms.intensity.**Clamp**(*min*, *max*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Apply augment_fn to keys

Parameters

- **min** (Union[float, *AbstractParameter*]) – minimal value
- **max** (Union[float, *AbstractParameter*]) – maximal value
- **keys** (Sequence) – the keys corresponding to the values to clamp
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

10.7.2 NormRange

class rising.transforms.intensity.**NormRange**(*min*, *max*, *keys=('data',)*, *per_channel=True*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.PerSampleTransform*

Parameters

- **min** (Union[float, *AbstractParameter*]) – minimal value
- **max** (Union[float, *AbstractParameter*]) – maximal value
- **keys** (Sequence) – keys to normalize
- **per_channel** (bool) – normalize per channel
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to normalization function

10.7.3 NormMinMax

class rising.transforms.intensity.**NormMinMax**(*keys=('data',)*, *per_channel=True*, *grad=False*, *eps=1e-08*, ***kwargs*)

Bases: *rising.transforms.abstract.PerSampleTransform*

Norm to [0, 1]

Parameters

- **keys** (Sequence) – keys to normalize
- **per_channel** (bool) – normalize per channel
- **grad** (bool) – enable gradient computation inside transformation
- **eps** (Optional[float]) – small constant for numerical stability. If None, no factor constant will be added
- ****kwargs** – keyword arguments passed to normalization function

10.7.4 NormZeroMeanUnitStd

class rising.transforms.intensity.**NormZeroMeanUnitStd**(*keys=('data',)*, *per_channel=True*, *grad=False*, *eps=1e-08*, ***kwargs*)

Bases: *rising.transforms.abstract.PerSampleTransform*

Normalize mean to zero and std to one

Parameters

- **keys** (Sequence) – keys to normalize
- **per_channel** (bool) – normalize per channel
- **grad** (bool) – enable gradient computation inside transformation
- **eps** (Optional[float]) – small constant for numerical stability. If None, no factor constant will be added
- ****kwargs** – keyword arguments passed to normalization function

10.7.5 NormMeanStd

class rising.transforms.intensity.**NormMeanStd**(*mean*, *std*, *keys=('data',)*, *per_channel=True*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.PerSampleTransform*

Normalize mean and std with provided values

Parameters

- **mean** (Union[float, Sequence[float]]) – used for mean normalization
- **std** (Union[float, Sequence[float]]) – used for std normalization
- **keys** (Sequence[str]) – keys to normalize
- **per_channel** (bool) – normalize per channel

- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to normalization function

10.7.6 Noise

class rising.transforms.intensity.**Noise**(*noise_type*, *per_channel=False*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.PerChannelTransform*

Add noise to data

Warning: This transform will apply different noise patterns to different keys.

Parameters

- **noise_type** (str) – supports all inplace functions of a torch.Tensor
- **per_channel** (bool) – enable transformation per channel
- **keys** (Sequence) – keys to normalize
- **grad** (bool) – enable gradient computation inside transformation
- **kwargs** – keyword arguments passed to noise function

See also:

torch.Tensor.normal_(),torch.Tensor.exponential_()

10.7.7 GaussianNoise

class rising.transforms.intensity.**GaussianNoise**(*mean*, *std*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.intensity.Noise*

Add gaussian noise to data

Warning: This transform will apply different noise patterns to different keys.

Parameters

- **mean** (float) – mean of normal distribution
- **std** (float) – std of normal distribution
- **keys** (Sequence) – keys to normalize
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to noise function

10.7.8 ExponentialNoise

class rising.transforms.intensity.**ExponentialNoise**(*lambd*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.intensity.Noise*

Add exponential noise to data

Warning: This transform will apply different noise patterns to different keys.

Parameters

- **lambd** (float) – lambda of exponential distribution
- **keys** (Sequence) – keys to normalize
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to noise function

10.7.9 GammaCorrection

class rising.transforms.intensity.**GammaCorrection**(*gamma*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Apply Gamma correction

Parameters

- **gamma** (Union[float, *AbstractParameter*]) – define gamma
- **keys** (Sequence) – keys to normalize
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to superclass

10.7.10 RandomValuePerChannel

class rising.transforms.intensity.**RandomValuePerChannel**(*augment_fn*, *random_sampler*, *per_channel=False*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.PerChannelTransform*

Apply augmentations which take random values as input by keyword `value`

Warning: This transform will apply different values to different keys.

Parameters

- **augment_fn** (callable) – augmentation function
- **random_mode** – specifies distribution which should be used to sample additive value. All function from python's random module are supported

- **random_args** – positional arguments passed for random function
- **per_channel** (bool) – enable transformation per channel
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

forward(***data*)

Perform Augmentation.

Parameters **data** – dict with data

Returns augmented data

Return type dict

10.7.11 RandomAddValue

class rising.transforms.intensity.**RandomAddValue**(*random_sampler*, *per_channel=False*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.intensity.RandomValuePerChannel*

Increase values additively

> **Warning:** This transform will apply different values to different keys.

Parameters

- **random_sampler** (*AbstractParameter*) – specify values to add
- **per_channel** (bool) – enable transformation per channel
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

10.7.12 RandomScaleValue

class rising.transforms.intensity.**RandomScaleValue**(*random_sampler*, *per_channel=False*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.intensity.RandomValuePerChannel*

Scale Values

> **Warning:** This transform will apply different values to different keys.

Parameters

- **random_sampler** (*AbstractParameter*) – specify values to add
- **per_channel** (bool) – enable transformation per channel

- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

10.8 Kernel Transforms

class rising.transforms.kernel.**KernelTransform**(*in_channels*, *kernel_size*, *dim=2*, *stride=1*, *padding=0*, *padding_mode='zero'*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.AbstractTransform*

Baseclass for kernel based transformations (kernel is applied to each channel individually)

Parameters

- **in_channels** (int) – number of input channels
- **kernel_size** (Union[int, Sequence]) – size of kernel
- **dim** (int) – number of spatial dimensions
- **stride** (Union[int, Sequence]) – stride of convolution
- **padding** (Union[int, Sequence]) – padding size for input
- **padding_mode** (str) – padding mode for input. Supports all modes from `torch.functional.pad()` except `circular`
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **kwargs** – keyword arguments passed to superclass

See also:

`torch.functional.pad()`

create_kernel()
Create kernel for convolution

Return type Tensor

forward(***data*)
Apply kernel to selected keys

Parameters **data** – input data

Returns dict with transformed data

Return type dict

static get_conv(*dim*)
Select convolution with regard to dimension

Parameters **dim** – spatial dimension of data

Returns the suitable convolutional function

Return type Callable

class rising.transforms.kernel.**GaussianSmoothing**(*in_channels*, *kernel_size*, *std*, *dim=2*, *stride=1*, *padding=0*, *padding_mode='reflect'*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.kernel.KernelTransform*

Perform Gaussian Smoothing. Filtering is performed seperately for each channel in the input using a depthwise convolution. This code is adapted from: 'https://discuss.pytorch.org/t/is-there-anyway-to-do-' 'gaussian-filtering-for-an-image-2d-3d-in-pytorch/12351/10'

Parameters

- **in_channels** (int) – number of input channels
- **kernel_size** (Union[int, Sequence]) – size of kernel
- **std** (Union[int, Sequence]) – standard deviation of gaussian
- **dim** (int) – number of spatial dimensions
- **stride** (Union[int, Sequence]) – stride of convolution
- **padding** (Union[int, Sequence]) – padding size for input
- **padding_mode** (str) – padding mode for input. Supports all modes from `torch.functional.pad()` except `circular`
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to superclass

See also:

`torch.functional.pad()`

create_kernel()

Create gaussian blur kernel

Return type Tensor

10.8.1 KernelTransform

class rising.transforms.kernel.**KernelTransform**(*in_channels*, *kernel_size*, *dim=2*, *stride=1*, *padding=0*, *padding_mode='zero'*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.AbstractTransform*

Baseclass for kernel based transformations (kernel is applied to each channel individually)

Parameters

- **in_channels** (int) – number of input channels
- **kernel_size** (Union[int, Sequence]) – size of kernel
- **dim** (int) – number of spatial dimensions
- **stride** (Union[int, Sequence]) – stride of convolution
- **padding** (Union[int, Sequence]) – padding size for input

- **padding_mode** (str) – padding mode for input. Supports all modes from torch.functional.pad() except circular
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **kwargs** – keyword arguments passed to superclass

See also:

torch.functional.pad()

create_kernel()
Create kernel for convolution

Return type Tensor

forward(***data*)
Apply kernel to selected keys

Parameters **data** – input data

Returns dict with transformed data

Return type dict

static get_conv(*dim*)
Select convolution with regard to dimension

Parameters **dim** – spatial dimension of data

Returns the suitable convolutional function

Return type Callable

10.8.2 GaussianSmoothing

class rising.transforms.kernel.**GaussianSmoothing**(*in_channels*, *kernel_size*, *std*, *dim=2*, *stride=1*, *padding=0*, *padding_mode='reflect'*, *keys=('data',)*, *grad=False*, ***kwargs*)
Bases: *rising.transforms.kernel.KernelTransform*

Perform Gaussian Smoothing. Filtering is performed seperately for each channel in the input using a depthwise convolution. This code is adapted from: 'https://discuss.pytorch.org/t/is-there-anyway-to-do-' 'gaussian-filtering-for-an-image-2d-3d-in-pytorch/12351/10'

Parameters

- **in_channels** (int) – number of input channels
- **kernel_size** (Union[int, Sequence]) – size of kernel
- **std** (Union[int, Sequence]) – standard deviation of gaussian
- **dim** (int) – number of spatial dimensions
- **stride** (Union[int, Sequence]) – stride of convolution
- **padding** (Union[int, Sequence]) – padding size for input
- **padding_mode** (str) – padding mode for input. Supports all modes from torch.functional.pad() except circular

- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to superclass

See also:

`torch.functional.pad()`

create_kernel()
Create gaussian blur kernel

Return type Tensor

10.9 Spatial Transforms

class rising.transforms.spatial.**Mirror**(*dims*, *keys=('data',)*, *grad=False*, ***kwargs*)
Bases: *rising.transforms.abstract.BaseTransform*

Random mirror transform

Parameters

- **dims** (Union[int, *DiscreteParameter*, Sequence[Union[int, *DiscreteParameter*]]]) – axes which should be mirrored
- **keys** (Sequence[str]) – keys which should be mirrored
- **prob** – probability for mirror. If float value is provided, it is used for all dims
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to superclass

Examples

```
>>> # Use mirror transform for augmentations
>>> from rising.random import DiscreteCombinationsParameter
>>> # We sample from all possible mirror combination for
>>> # volumetric data
>>> trafo = Mirror(DiscreteCombinationsParameter((0, 1, 2)))
```

class rising.transforms.spatial.**Rot90**(*dims*, *keys=('data',)*, *num_rots=(0, 1, 2, 3)*, *prob=0.5*, *grad=False*, ***kwargs*)
Bases: *rising.transforms.abstract.AbstractTransform*

Rotate 90 degree around dims

Parameters

- **dims** (Union[Sequence[int], *DiscreteParameter*]) – dims/axis ro rotate. If more than two dims are provided, 2 dimensions are randomly chosen at each call
- **keys** (Sequence[str]) – keys which should be rotated
- **num_rots** (Sequence[int]) – possible values for number of rotations
- **prob** (float) – probability for rotation
- **grad** (bool) – enable gradient computation inside transformation

- **kwargs** – keyword arguments passed to superclass

See also:

`torch.Tensor.rot90()`

forward(***data*)
Apply transformation

Parameters **data** – dict with tensors

Returns dict with augmented data

Return type dict

class rising.transforms.spatial.**ResizeNative**(*size*, *mode='nearest'*, *align_corners=None*, *preserve_range=False*, *keys=('data',)*, *grad=False*, ***kwargs*)
Bases: *rising.transforms.abstract.BaseTransform*

Resize data to given size

Parameters

- **size** (Union[int, Sequence[int]]) – spatial output size (excluding batch size and number of channels)
- **mode** (str) – one of `nearest`, `linear`, `bilinear`, `bicubic`, `trilinear`, `area` (for more inforamtion see `torch.nn.functional.interpolate()`)
- **align_corners** (Optional[bool]) – input and output tensors are aligned by the center points of their corners pixels, preserving the values at the corner pixels.
- **preserve_range** (bool) – output tensor has same range as input tensor
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

class rising.transforms.spatial.**Zoom**(*scale_factor=(0.75, 1.25)*, *mode='nearest'*, *align_corners=None*, *preserve_range=False*, *keys=('data',)*, *grad=False*, ***kwargs*)
Bases: *rising.transforms.abstract.BaseTransform*

Apply augment_fn to keys. By default the scaling factor is sampled from a uniform distribution with the range specified by `random_args`

Parameters

- **scale_factor** (Union[Sequence, *AbstractParameter*]) – positional arguments passed for random function. If Sequence[Sequence] is provided, a random value for each item in the outer Sequence is generated. This can be used to set different ranges for different axis.
- **mode** (str) – one of *nearest*, *linear*, *bilinear*, *bicubic*, *trilinear*, *area* (for more inforamtion see `torch.nn.functional.interpolate()`)
- **align_corners** (Optional[bool]) – input and output tensors are aligned by the center points of their corners pixels, preserving the values at the corner pixels.
- **preserve_range** (bool) – output tensor has same range as input tensor
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation

- ****kwargs** – keyword arguments passed to augment_fn

See also:

`random.uniform()`, `torch.nn.functional.interpolate()`

class rising.transforms.spatial.**ProgressiveResize**(*scheduler*, *mode='nearest'*, *align_corners=None*, *preserve_range=False*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.spatial.ResizeNative*

Resize data to sizes specified by scheduler

Parameters

- **scheduler** (Callable[[int], Union[int, Sequence[int]]]) – scheduler which determined the current size. The scheduler is called with the current iteration of the transform
- **mode** (str) – one of nearest, linear, bilinear, bicubic, trilinear, area (for more inforamtion see `torch.nn.functional.interpolate()`)
- **align_corners** (Optional[bool]) – input and output tensors are aligned by the center points of their corners pixels, preserving the values at the corner pixels.
- **preserve_range** (bool) – output tensor has same range as input tensor
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

Warning: When this transformations is used in combination with multiprocessing, the step counter is not perfectly synchronized between multiple processes. As a result the step count my jump between values in a range of the number of processes used.

forward(***data*)

Resize data

Parameters ****data** – input batch

Returns augmented batch

Return type dict

increment()

Increment step by 1

Returns returns self to allow chaining

Return type *ResizeNative*

reset_step()

Reset step to 0

Returns returns self to allow chaining

Return type *ResizeNative*

property step

Current step

Returns number of steps

Return type int

class rising.transforms.spatial.**SizeStepScheduler**(*milestones*, *sizes*)
Bases: object

Scheduler return size when milestone is reached

Parameters

- **milestones** (Sequence[int]) – contains number of iterations where size should be changed
- **sizes** (Union[Sequence[int], Sequence[Sequence[int]]]) – sizes corresponding to milestones

__call__(*step*)
Return size with regard to milestones

Parameters **step** – current step

Returns current size

Return type Union[int, Sequence[int], Sequence[Sequence[int]]]

10.9.1 Mirror

class rising.transforms.spatial.**Mirror**(*dims*, *keys=('data',)*, *grad=False*, ***kwargs*)
Bases: *rising.transforms.abstract.BaseTransform*

Random mirror transform

Parameters

- **dims** (Union[int, *DiscreteParameter*, Sequence[Union[int, *DiscreteParameter*]]]) – axes which should be mirrored
- **keys** (Sequence[str]) – keys which should be mirrored
- **prob** – probability for mirror. If float value is provided, it is used for all dims
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to superclass

Examples

```
>>> # Use mirror transform for augmentations
>>> from rising.random import DiscreteCombinationsParameter
>>> # We sample from all possible mirror combination for
>>> # volumetric data
>>> trafo = Mirror(DiscreteCombinationsParameter((0, 1, 2)))
```

10.9.2 Rot90

class rising.transforms.spatial.**Rot90**(*dims*, *keys=('data',)*, *num_rots=(0, 1, 2, 3)*, *prob=0.5*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.AbstractTransform*

Rotate 90 degree around dims

Parameters

- **dims** (Union[Sequence[int], *DiscreteParameter*]) – dims/axis ro rotate. If more than two dims are provided, 2 dimensions are randomly chosen at each call
- **keys** (Sequence[str]) – keys which should be rotated
- **num_rots** (Sequence[int]) – possible values for number of rotations
- **prob** (float) – probability for rotation
- **grad** (bool) – enable gradient computation inside transformation
- **kwargs** – keyword arguments passed to superclass

See also:

`torch.Tensor.rot90()`

forward(***data*)

Apply transformation

Parameters **data** – dict with tensors

Returns dict with augmented data

Return type dict

10.9.3 ResizeNative

class rising.transforms.spatial.**ResizeNative**(*size*, *mode='nearest'*, *align_corners=None*, *preserve_range=False*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Resize data to given size

Parameters

- **size** (Union[int, Sequence[int]]) – spatial output size (excluding batch size and number of channels)
- **mode** (str) – one of `nearest`, `linear`, `bilinear`, `bicubic`, `trilinear`, `area` (for more inforamtion see `torch.nn.functional.interpolate()`)
- **align_corners** (Optional[bool]) – input and output tensors are aligned by the center points of their corners pixels, preserving the values at the corner pixels.
- **preserve_range** (bool) – output tensor has same range as input tensor
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

10.9.4 Zoom

class rising.transforms.spatial.**Zoom**(*scale_factor=(0.75, 1.25)*, *mode='nearest'*, *align_corners=None*, *preserve_range=False*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Apply augment_fn to keys. By default the scaling factor is sampled from a uniform distribution with the range specified by `random_args`

Parameters

- **scale_factor** (Union[Sequence, *AbstractParameter*]) – positional arguments passed for random function. If Sequence[Sequence] is provided, a random value for each item in the outer Sequence is generated. This can be used to set different ranges for different axis.
- **mode** (str) – one of *nearest*, *linear*, *bilinear*, *bicubic*, *trilinear*, *area* (for more inforamtion see `torch.nn.functional.interpolate()`)
- **align_corners** (Optional[bool]) – input and output tensors are aligned by the center points of their corners pixels, preserving the values at the corner pixels.
- **preserve_range** (bool) – output tensor has same range as input tensor
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

See also:

`random.uniform()`, `torch.nn.functional.interpolate()`

10.9.5 ProgressiveResize

class rising.transforms.spatial.**ProgressiveResize**(*scheduler*, *mode='nearest'*, *align_corners=None*, *preserve_range=False*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.spatial.ResizeNative*

Resize data to sizes specified by scheduler

Parameters

- **scheduler** (Callable[[int], Union[int, Sequence[int]]]) – scheduler which determined the current size. The scheduler is called with the current iteration of the transform
- **mode** (str) – one of `nearest`, `linear`, `bilinear`, `bicubic`, `trilinear`, `area` (for more inforamtion see `torch.nn.functional.interpolate()`)
- **align_corners** (Optional[bool]) – input and output tensors are aligned by the center points of their corners pixels, preserving the values at the corner pixels.
- **preserve_range** (bool) – output tensor has same range as input tensor
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation

- ****kwargs** – keyword arguments passed to augment_fn

Warning: When this transformations is used in combination with multiprocessing, the step counter is not perfectly synchronized between multiple processes. As a result the step count my jump between values in a range of the number of processes used.

forward(***data*)
Resize data

Parameters ****data** – input batch

Returns augmented batch

Return type dict

increment()
Increment step by 1

Returns returns self to allow chaining

Return type *ResizeNative*

reset_step()
Reset step to 0

Returns returns self to allow chaining

Return type *ResizeNative*

property step
Current step

Returns number of steps

Return type int

10.9.6 SizeStepScheduler

class rising.transforms.spatial.**SizeStepScheduler**(*milestones*, *sizes*)
Bases: object

Scheduler return size when milestone is reached

Parameters

- **milestones** (Sequence[int]) – contains number of iterations where size should be changed
- **sizes** (Union[Sequence[int], Sequence[Sequence[int]]]) – sizes corresponding to milestones

__call__(*step*)
Return size with regard to milestones

Parameters **step** – current step

Returns current size

Return type Union[int, Sequence[int], Sequence[Sequence[int]]]

10.10 Tensor Transforms

class rising.transforms.tensor.**ToTensor**(*keys=('data',)*, *grad=False*, ***kwargs*)
Bases: *rising.transforms.abstract.BaseTransform*

Transform Input Collection to Collection of torch.Tensor

Parameters

- **keys** (Sequence) – keys which should be transformed
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

class rising.transforms.tensor.**ToDeviceDtype**(*device=None*, *dtype=None*, *non_blocking=False*, *copy=False*, *keys=('data',)*, *grad=False*, ***kwargs*)
Bases: *rising.transforms.abstract.BaseTransform*

Push data to device and convert to tdype

Parameters

- **device** (Union[device, str, None]) – target device
- **dtype** (Optional[dtype]) – target dtype
- **non_blocking** (bool) – if True and this copy is between CPU and GPU, the copy may occur asynchronously with respect to the host. For other cases, this argument has no effect.
- **copy** (bool) – create copy of data
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to function

class rising.transforms.tensor.**ToDevice**(*device*, *non_blocking=False*, *copy=False*, *keys=('data',)*, *grad=False*, ***kwargs*)
Bases: *rising.transforms.tensor.ToDeviceDtype*

Push data to device

Parameters

- **device** (Union[device, str, None]) – target device
- **non_blocking** (bool) – if True and this copy is between CPU and GPU, the copy may occur asynchronously with respect to the host. For other cases, this argument has no effect.
- **copy** (bool) – create copy of data
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to function

class rising.transforms.tensor.**ToDtype**(*dtype*, *keys=('data',)*, *grad=False*, ***kwargs*)
Bases: *rising.transforms.tensor.ToDeviceDtype*

Convert data to dtype

Parameters

- **dtype** (dtype) – target dtype

- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- **kwargs** – keyword arguments passed to function

class rising.transforms.tensor.**TensorOp**(*op_name*, **args*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Apply function which are supported by the *torch.Tensor* class

Parameters

- **op_name** (str) – name of tensor operation
- ***args** – positional arguments passed to function
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to function

class rising.transforms.tensor.**Permute**(*dims*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Permute dimensions of tensor

Parameters

- **dims** (Dict[str, Sequence[int]]) – defines permutation sequence for respective key
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to permute function

forward(***data*)

Forward input

Args: data: batch dict

Returns augmented data

Return type dict

10.10.1 ToTensor

class rising.transforms.tensor.**ToTensor**(*keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Transform Input Collection to Collection of torch.Tensor

Parameters

- **keys** (Sequence) – keys which should be transformed
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to augment_fn

10.10.2 ToDeviceDtype

class rising.transforms.tensor.**ToDeviceDtype**(*device=None*, *dtype=None*, *non_blocking=False*, *copy=False*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Push data to device and convert to tdype

Parameters

- **device** (Union[device, str, None]) – target device
- **dtype** (Optional[dtype]) – target dtype
- **non_blocking** (bool) – if True and this copy is between CPU and GPU, the copy may occur asynchronously with respect to the host. For other cases, this argument has no effect.
- **copy** (bool) – create copy of data
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to function

10.10.3 ToDevice

class rising.transforms.tensor.**ToDevice**(*device*, *non_blocking=False*, *copy=False*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.tensor.ToDeviceDtype*

Push data to device

Parameters

- **device** (Union[device, str, None]) – target device
- **non_blocking** (bool) – if True and this copy is between CPU and GPU, the copy may occur asynchronously with respect to the host. For other cases, this argument has no effect.
- **copy** (bool) – create copy of data
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to function

10.10.4 ToDtype

class rising.transforms.tensor.**ToDtype**(*dtype*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.tensor.ToDeviceDtype*

Convert data to dtype

Parameters

- **dtype** (dtype) – target dtype
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation

- **kwargs** – keyword arguments passed to function

10.10.5 TensorOp

class rising.transforms.tensor.**TensorOp**(*op_name*, **args*, *keys=('data',)*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Apply function which are supported by the *torch.Tensor* class

Parameters

- **op_name** (str) – name of tensor operation
- ***args** – positional arguments passed to function
- **keys** (Sequence) – keys which should be augmented
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to function

10.10.6 Permute

class rising.transforms.tensor.**Permute**(*dims*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.BaseTransform*

Permute dimensions of tensor

Parameters

- **dims** (Dict[str, Sequence[int]]) – defines permutation sequence for respective key
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to permute function

forward(***data*)

Forward input

Args: data: batch dict

Returns augmented data

Return type dict

10.11 Utility Transforms

class rising.transforms.utility.**DoNothing**(*grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.AbstractTransform*

Transform that returns the input as is

Parameters

- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to superclass

forward(***data*)

Forward input

Parameters **data** – input dict

Return type dict

Returns input dict

class rising.transforms.utility.**SegToBox**(*keys*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.AbstractTransform*

Convert instance segmentation to bounding boxes

Parameters

- **keys** (Mapping[Hashable, Hashable]) – the key specifies which item to use as segmentation and the item specifies where the save the bounding boxes
- **grad** (bool) – enable gradient computation inside transformation

forward(***data*)

Parameters ****data** – input data

Returns transformed data

Return type dict

class rising.transforms.utility.**BoxToSeg**(*keys*, *shape*, *dtype*, *device*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.AbstractTransform*

Convert bounding boxes to instance segmentation

Parameters

- **keys** (Mapping[Hashable, Hashable]) – the key specifies which item to use as the bounding boxes and the item specifies where the save the bounding boxes
- **shape** (Sequence[int]) – spatial shape of output tensor (batchsize is derived from bounding boxes and has one channel)
- **dtype** (dtype) – dtype of segmentation
- **device** (Union[device, str]) – device of segmentation
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – Additional keyword arguments forwarded to the Base Class

forward(***data*)

Forward input

Parameters ****data** – input data

Returns transformed data

Return type dict

class rising.transforms.utility.**InstanceToSemantic**(*keys*, *cls_key*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.AbstractTransform*

Convert an instance segmentation to a semantic segmentation

Parameters

- **keys** (Mapping[str, str]) – the key specifies which item to use as instance segmentation and the item specifies where the save the semantic segmentation

- **cls_key** (Hashable) – key where the class mapping is saved. Mapping needs to be a Sequence{Sequence[int]].
- **grad** (bool) – enable gradient computation inside transformation

forward(***data*)
Forward input

Parameters ****data** – input data

Returns transformed data

Return type dict

10.11.1 DoNothing

class rising.transforms.utility.**DoNothing**(*grad=False*, ***kwargs*)
Bases: *rising.transforms.abstract.AbstractTransform*

Transform that returns the input as is

Parameters

- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – keyword arguments passed to superclass

forward(***data*)
Forward input

Parameters **data** – input dict

Return type dict

Returns input dict

10.11.2 SegToBox

class rising.transforms.utility.**SegToBox**(*keys*, *grad=False*, ***kwargs*)
Bases: *rising.transforms.abstract.AbstractTransform*

Convert instance segmentation to bounding boxes

Parameters

- **keys** (Mapping[Hashable, Hashable]) – the key specifies which item to use as segmentation and the item specifies where the save the bounding boxes
- **grad** (bool) – enable gradient computation inside transformation

forward(***data*)

Parameters ****data** – input data

Returns transformed data

Return type dict

10.11.3 BoxToSeg

class rising.transforms.utility.**BoxToSeg**(*keys*, *shape*, *dtype*, *device*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.AbstractTransform*

Convert bounding boxes to instance segmentation

Parameters

- **keys** (Mapping[Hashable, Hashable]) – the key specifies which item to use as the bounding boxes and the item specifies where the save the bounding boxes
- **shape** (Sequence[int]) – spatial shape of output tensor (batchsize is derived from bounding boxes and has one channel)
- **dtype** (dtype) – dtype of segmentation
- **device** (Union[device, str]) – device of segmentation
- **grad** (bool) – enable gradient computation inside transformation
- ****kwargs** – Additional keyword arguments forwarded to the Base Class

forward(***data*)

Forward input

Parameters ****data** – input data

Returns transformed data

Return type dict

10.11.4 InstanceToSemantic

class rising.transforms.utility.**InstanceToSemantic**(*keys*, *cls_key*, *grad=False*, ***kwargs*)

Bases: *rising.transforms.abstract.AbstractTransform*

Convert an instance segmentation to a semantic segmentation

Parameters

- **keys** (Mapping[str, str]) – the key specifies which item to use as instance segmentation and the item specifies where the save the semantic segmentation
- **cls_key** (Hashable) – key where the class mapping is saved. Mapping needs to be a Sequence{Sequence[int]].
- **grad** (bool) – enable gradient computation inside transformation

forward(***data*)

Forward input

Parameters ****data** – input data

Returns transformed data

Return type dict

CHAPTER

ELEVEN

RISING.TRANSFORMS.FUNCTIONAL

Provides a functional interface for transforms (usually working on single tensors rather then collections thereof). All transformations are implemented to work on batched tensors. Implementations include:

- Affine Transforms
- Channel Transforms
- Cropping Transforms
- Device Transforms
- Intensity Transforms
- Spatial Transforms
- Tensor Transforms
- Utility Transforms

11.1 Affine Transforms

rising.transforms.functional.affine.**affine_image_transform**(*image_batch*, *matrix_batch*, *output_size=None*, *adjust_size=False*, *interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*, *reverse_order=False*)

Performs an affine transformation on a batch of images

Parameters

- **image_batch** (Tensor) – the batch to transform. Should have shape of [N, C, NDIM]
- **matrix_batch** (Tensor) – a batch of affine matrices with shape [N, NDIM, NDIM+1]
- **output_size** (Optional[tuple]) – if given, this will be the resulting image size. Defaults to None
- **adjust_size** (bool) – if True, the resulting image size will be calculated dynamically to ensure that the whole image fits.
- **interpolation_mode** (str) – interpolation mode to calculate output values 'bilinear' | 'nearest'. Default: 'bilinear'

- **padding_mode** (str) – padding mode for outside grid values 'zeros' | 'border' | 'reflection'. Default: 'zeros'
- **align_corners** (bool) – Geometrically, we consider the pixels of the input as squares rather than points. If set to True, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to False, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic.

Returns transformed image

Return type torch.Tensor

Warning: When align_corners = True, the grid positions depend on the pixel size relative to the input image size, and so the locations sampled by grid_sample() will differ for the same input given at different resolutions (that is, after being upsampled or downsampled).

Notes

`output_size` and `adjust_size` are mutually exclusive. If None of them is set, the resulting image will have the same size as the input image.

rising.transforms.functional.affine.**affine_point_transform**(*point_batch*, *matrix_batch*)

Function to perform an affine transformation onto point batches

Parameters

- **point_batch** (Tensor) – a point batch of shape [N, NP, NDIM] `NP` is the number of points, `N` is the batch size, `NDIM` is the number of spatial dimensions
- **matrix_batch** (Tensor) – torch.Tensor a batch of affine matrices with shape [N, NDIM, NDIM + 1], N is the batch size and NDIM is the number of spatial dimensions

Returns

the batch of transformed points in cartesian coordinates) [N, NP, NDIM] `NP` is the number of points, `N` is the batch size, `NDIM` is the number of spatial dimensions

Return type torch.Tensor

rising.transforms.functional.affine.**create_rotation**(*rotation*, *batchsize*, *ndim*, *degree=False*, *device=None*, *dtype=None*, *image_transform=True*)

Formats the given scale parameters to a homogeneous transformation matrix

Parameters

- **rotation** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – the rotation factor(s). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per sample, which will be replicated for all dimensions * a parameter per dimension, which will be replicated for all batch samples * a parameter per sampler per dimension * None will be treated as a rotation angle of 0
- **batchsize** (int) – the number of samples per batch
- **ndim** (int) – the dimensionality of the transform

- **degree** (bool) – whether the given rotation(s) are in degrees. Only valid for rotation parameters, which aren't passed as full transformation matrix.
- **device** (Union[device, str, None]) – the device to put the resulting tensor to. Defaults to the torch default device
- **dtype** (Union[dtype, str, None]) – the dtype of the resulting trensor. Defaults to the torch default dtype
- **image_transform** (bool) – bool inverts the rotation matrix to match expected behavior when applied to an image, e.g. rotation > 0 should rotate the image counter clockwise but the grid clockwise

Returns

the homogeneous transformation matrix [N, NDIM + 1, NDIM + 1], N is the batch size and NDIM is the number of spatial dimensions

Return type torch.Tensor

rising.transforms.functional.affine.**create_scale**(*scale*, *batchsize*, *ndim*, *device=None*, *dtype=None*, *image_transform=True*)

Formats the given scale parameters to a homogeneous transformation matrix

Parameters

- **scale** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – the scale factor(s). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per sample, which will be replicated for all dimensions * a parameter per dimension, which will be replicated for all batch samples * a parameter per sampler per dimension * None will be treated as a scaling factor of 1
- **batchsize** (int) – the number of samples per batch
- **ndim** (int) – the dimensionality of the transform
- **device** (Union[device, str, None]) – the device to put the resulting tensor to. Defaults to the torch default device
- **dtype** (Union[dtype, str, None]) – the dtype of the resulting trensor. Defaults to the torch default dtype
- **image_transform** (bool) – inverts the scale matrix to match expected behavior when applied to an image, e.g. scale>1 increases the size of an image but decrease the size of an grid

Returns

the homogeneous transformation matrix [N, NDIM + 1, NDIM + 1], N is the batch size and NDIM is the number of spatial dimensions

Return type torch.Tensor

rising.transforms.functional.affine.**create_translation**(*offset*, *batchsize*, *ndim*, *device=None*, *dtype=None*, *image_transform=True*)

Formats the given translation parameters to a homogeneous transformation matrix

Parameters

- **offset** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – the translation offset(s). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per sample, which will be replicated for all dimensions * a parameter per dimension, which will be replicated for all batch samples * a parameter per sampler per dimension * None will be treated as a translation offset of 0
- **batchsize** (int) – the number of samples per batch
- **ndim** (int) – the dimensionality of the transform
- **device** (Union[device, str, None]) – the device to put the resulting tensor to. Defaults to the torch default device
- **dtype** (Union[dtype, str, None]) – the dtype of the resulting trensor. Defaults to the torch default dtype
- **image_transform** (bool) – bool inverts the translation matrix to match expected behavior when applied to an image, e.g. translation > 0 should move the image in the positive direction of an axis but the grid in the negative direction

Returns

the homogeneous transformation matrix [N, NDIM + 1, NDIM + 1], N is the batch size and NDIM is the number of spatial dimensions

Return type torch.Tensor

rising.transforms.functional.affine.**parametrize_matrix**(*scale*, *rotation*, *translation*, *batchsize*, *ndim*, *degree=False*, *device=None*, *dtype=None*, *image_transform=True*)

Formats the given scale parameters to a homogeneous transformation matrix

Parameters

- **scale** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – the scale factor(s). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per sample, which will be replicated for all dimensions * a parameter per dimension, which will be replicated for all batch samples * a parameter per sampler per dimension * None will be treated as a scaling factor of 1
- **rotation** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – the rotation factor(s). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per sample, which will be replicated for all dimensions * a parameter per dimension, which will be replicated for all batch samples * a parameter per sampler per dimension * None will be treated as a rotation factor of 1
- **translation** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – the translation offset(s). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per sample, which will be replicated for all dimensions * a parameter per dimension, which will be replicated for all batch samples * a parameter per sampler per dimension * None will be treated as a translation offset of 0
- **batchsize** (int) – the number of samples per batch
- **ndim** (int) – the dimensionality of the transform

- **degree** (bool) – whether the given rotation(s) are in degrees. Only valid for rotation parameters, which aren't passed as full transformation matrix.
- **device** (Union[device, str, None]) – the device to put the resulting tensor to. Defaults to the torch default device
- **dtype** (Union[dtype, str, None]) – the dtype of the resulting trensor. Defaults to the torch default dtype
- **image_transform** (bool) – bool adjusts transformation matrices such that they match the expected behavior on images (see *create_scale()* and *create_translation()* for more info)

Returns

the transformation matrix [N, NDIM, NDIM+1], N is the batch size and NDIM is the number of spatial dimensions

Return type torch.Tensor

11.1.1 affine_image_transform

rising.transforms.functional.affine.**affine_image_transform**(*image_batch*, *matrix_batch*, *output_size=None*, *adjust_size=False*, *interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*, *reverse_order=False*)

Performs an affine transformation on a batch of images

Parameters

- **image_batch** (Tensor) – the batch to transform. Should have shape of [N, C, NDIM]
- **matrix_batch** (Tensor) – a batch of affine matrices with shape [N, NDIM, NDIM+1]
- **output_size** (Optional[tuple]) – if given, this will be the resulting image size. Defaults to None
- **adjust_size** (bool) – if True, the resulting image size will be calculated dynamically to ensure that the whole image fits.
- **interpolation_mode** (str) – interpolation mode to calculate output values 'bilinear' | 'nearest'. Default: 'bilinear'
- **padding_mode** (str) – padding mode for outside grid values 'zeros' | 'border' | 'reflection'. Default: 'zeros'
- **align_corners** (bool) – Geometrically, we consider the pixels of the input as squares rather than points. If set to True, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to False, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic.

Returns transformed image

Return type torch.Tensor

Warning: When align_corners = True, the grid positions depend on the pixel size relative to the input image size, and so the locations sampled by grid_sample() will differ for the same input given at different resolutions (that is, after being upsampled or downsampled).

Notes

`output_size` and `adjust_size` are mutually exclusive. If None of them is set, the resulting image will have the same size as the input image.

11.1.2 affine_point_transform

rising.transforms.functional.affine.**affine_point_transform**(*point_batch*, *matrix_batch*)

Function to perform an affine transformation onto point batches

Parameters

- **point_batch** (Tensor) – a point batch of shape [N, NP, NDIM] `NP` is the number of points, `N` is the batch size, `NDIM` is the number of spatial dimensions
- **matrix_batch** (Tensor) – torch.Tensor a batch of affine matrices with shape [N, NDIM, NDIM + 1], N is the batch size and NDIM is the number of spatial dimensions

Returns

the batch of transformed points in cartesian coordinates) [N, NP, NDIM] `NP` is the number of points, `N` is the batch size, `NDIM` is the number of spatial dimensions

Return type torch.Tensor

11.1.3 parametrize_matrix

rising.transforms.functional.affine.**parametrize_matrix**(*scale*, *rotation*, *translation*, *batchsize*, *ndim*, *degree=False*, *device=None*, *dtype=None*, *image_transform=True*)

Formats the given scale parameters to a homogeneous transformation matrix

Parameters

- **scale** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – the scale factor(s). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per sample, which will be replicated for all dimensions * a parameter per dimension, which will be replicated for all batch samples * a parameter per sampler per dimension * None will be treated as a scaling factor of 1
- **rotation** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – the rotation factor(s). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per sample, which will be replicated for all dimensions * a parameter per dimension, which will be replicated for all batch samples * a parameter per sampler per dimension * None will be treated as a rotation factor of 1

- **translation** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – the translation offset(s). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per sample, which will be replicated for all dimensions * a parameter per dimension, which will be replicated for all batch samples * a parameter per sampler per dimension * None will be treated as a translation offset of 0
- **batchsize** (int) – the number of samples per batch
- **ndim** (int) – the dimensionality of the transform
- **degree** (bool) – whether the given rotation(s) are in degrees. Only valid for rotation parameters, which aren't passed as full transformation matrix.
- **device** (Union[device, str, None]) – the device to put the resulting tensor to. Defaults to the torch default device
- **dtype** (Union[dtype, str, None]) – the dtype of the resulting trensor. Defaults to the torch default dtype
- **image_transform** (bool) – bool adjusts transformation matrices such that they match the expected behavior on images (see *create_scale()* and *create_translation()* for more info)

Returns

the transformation matrix [N, NDIM, NDIM+1], N is the batch size and NDIM is the number of spatial dimensions

Return type torch.Tensor

11.1.4 create_rotation

rising.transforms.functional.affine.**create_rotation**(*rotation*, *batchsize*, *ndim*, *degree=False*, *device=None*, *dtype=None*, *image_transform=True*)

Formats the given scale parameters to a homogeneous transformation matrix

Parameters

- **rotation** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – the rotation factor(s). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per sample, which will be replicated for all dimensions * a parameter per dimension, which will be replicated for all batch samples * a parameter per sampler per dimension * None will be treated as a rotation angle of 0
- **batchsize** (int) – the number of samples per batch
- **ndim** (int) – the dimensionality of the transform
- **degree** (bool) – whether the given rotation(s) are in degrees. Only valid for rotation parameters, which aren't passed as full transformation matrix.
- **device** (Union[device, str, None]) – the device to put the resulting tensor to. Defaults to the torch default device
- **dtype** (Union[dtype, str, None]) – the dtype of the resulting trensor. Defaults to the torch default dtype

- **image_transform** (bool) – bool inverts the rotation matrix to match expected behavior when applied to an image, e.g. rotation > 0 should rotate the image counter clockwise but the grid clockwise

Returns

the homogeneous transformation matrix [N, NDIM + 1, NDIM + 1], N is the batch size and NDIM is the number of spatial dimensions

Return type torch.Tensor

11.1.5 create_rotation_2d

rising.transforms.functional.affine.**create_rotation_2d**(*sin*, *cos*)

Create a 2d rotation matrix

Parameters

- **sin** (Tensor) – sin value to use for rotation matrix, [1]
- **cos** (Tensor) – cos value to use for rotation matrix, [1]
- **Returns** – torch.Tensor: rotation matrix, [2, 2]

Return type Tensor

11.1.6 create_rotation_3d

rising.transforms.functional.affine.**create_rotation_3d**(*sin*, *cos*)

Create a 3d rotation matrix which sequentially applies the rotation around axis (rot axis 0 -> rot axis 1 -> rot axis 2)

Parameters

- **sin** (Tensor) – sin values to use for the rotation, (axis 0, axis 1, axis 2)[3]
- **cos** (Tensor) – cos values to use for the rotation, (axis 0, axis 1, axis 2)[3]

Returns rotation matrix, [3, 3]

Return type torch.Tensor

11.1.7 create_rotation_3d_0

rising.transforms.functional.affine.**create_rotation_3d_0**(*sin*, *cos*)

Create a rotation matrix around the zero-th axis

Parameters

- **sin** (Tensor) – sin value to use for rotation matrix, [1]
- **cos** (Tensor) – cos value to use for rotation matrix, [1]

Returns rotation matrix, [3, 3]

Return type torch.Tensor

11.1.8 create_rotation_3d_1

rising.transforms.functional.affine.**create_rotation_3d_1**(*sin*, *cos*)
Create a rotation matrix around the first axis

Parameters

- **sin** (Tensor) – sin value to use for rotation matrix, [1]
- **cos** (Tensor) – cos value to use for rotation matrix, [1]

Returns rotation matrix, [3, 3]

Return type torch.Tensor

11.1.9 create_rotation_3d_2

rising.transforms.functional.affine.**create_rotation_3d_2**(*sin*, *cos*)
Create a rotation matrix around the second axis

Parameters

- **sin** (Tensor) – sin value to use for rotation matrix, [1]
- **cos** (Tensor) – cos value to use for rotation matrix, [1]

Returns rotation matrix, [3, 3]

Return type torch.Tensor

11.1.10 create_scale

rising.transforms.functional.affine.**create_scale**(*scale*, *batchsize*, *ndim*, *device=None*, *dtype=None*, *image_transform=True*)
Formats the given scale parameters to a homogeneous transformation matrix

Parameters

- **scale** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – the scale factor(s). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per sample, which will be replicated for all dimensions * a parameter per dimension, which will be replicated for all batch samples * a parameter per sampler per dimension * None will be treated as a scaling factor of 1
- **batchsize** (int) – the number of samples per batch
- **ndim** (int) – the dimensionality of the transform
- **device** (Union[device, str, None]) – the device to put the resulting tensor to. Defaults to the torch default device
- **dtype** (Union[dtype, str, None]) – the dtype of the resulting trensor. Defaults to the torch default dtype
- **image_transform** (bool) – inverts the scale matrix to match expected behavior when applied to an image, e.g. scale>1 increases the size of an image but decrease the size of an grid

Returns

the homogeneous transformation matrix [N, NDIM + 1, NDIM + 1], N is the batch size and NDIM is the number of spatial dimensions

Return type torch.Tensor

11.1.11 create_translation

rising.transforms.functional.affine.**create_translation**(*offset*, *batchsize*, *ndim*, *device=None*, *dtype=None*, *image_transform=True*)

Formats the given translation parameters to a homogeneous transformation matrix

Parameters

- **offset** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – the translation offset(s). Supported are: * a single parameter (as float or int), which will be replicated for all dimensions and batch samples * a parameter per sample, which will be replicated for all dimensions * a parameter per dimension, which will be replicated for all batch samples * a parameter per sampler per dimension * None will be treated as a translation offset of 0
- **batchsize** (int) – the number of samples per batch
- **ndim** (int) – the dimensionality of the transform
- **device** (Union[device, str, None]) – the device to put the resulting tensor to. Defaults to the torch default device
- **dtype** (Union[dtype, str, None]) – the dtype of the resulting trensor. Defaults to the torch default dtype
- **image_transform** (bool) – bool inverts the translation matrix to match expected behavior when applied to an image, e.g. translation > 0 should move the image in the positive direction of an axis but the grid in the negative direction

Returns

the homogeneous transformation matrix [N, NDIM + 1, NDIM + 1], N is the batch size and NDIM is the number of spatial dimensions

Return type torch.Tensor

11.1.12 expand_scalar_param

rising.transforms.functional.affine.**expand_scalar_param**(*param*, *batchsize*, *ndim*)

Bring affine params to shape (batchsize, ndim)

Parameters

- **param** (Union[int, Sequence[int], float, Sequence[float], Tensor, *AbstractParameter*, Sequence[*AbstractParameter*]]) – affine parameter
- **batchsize** (int) – size of batch
- **ndim** (int) – number of spatial dimensions

Returns affine params in correct shape

Return type torch.Tensor

11.2 Channel Transforms

rising.transforms.functional.channel.**one_hot_batch**(*target*, *num_classes=None*, *dtype=None*)

Compute one hot for input tensor (assumed to a be batch and thus saved into first dimension -> input should only have one channel)

Parameters

- **target** (Tensor) – long tensor to be converted
- **num_classes** (Optional[int]) – number of classes. If num_classes is None, the maximum of target is used
- **dtype** (Optional[dtype]) – optionally changes the dtype of the onehot encoding

Returns one hot encoded tensor

Return type torch.Tensor

11.2.1 one_hot_batch

rising.transforms.functional.channel.**one_hot_batch**(*target*, *num_classes=None*, *dtype=None*)

Compute one hot for input tensor (assumed to a be batch and thus saved into first dimension -> input should only have one channel)

Parameters

- **target** (Tensor) – long tensor to be converted
- **num_classes** (Optional[int]) – number of classes. If num_classes is None, the maximum of target is used
- **dtype** (Optional[dtype]) – optionally changes the dtype of the onehot encoding

Returns one hot encoded tensor

Return type torch.Tensor

11.3 Cropping Transforms

rising.transforms.functional.crop.**crop**(*data*, *corner*, *size*)

Extract crop from last dimensions of data

Args: data: input tensor corner: top left corner point size: size of patch

Returns cropped data

Return type torch.Tensor

rising.transforms.functional.crop.**center_crop**(*data*, *size*)

Crop patch from center

Args: data: input tensor size: size of patch

Returns output tensor cropped from input tensor

Return type torch.Tensor

rising.transforms.functional.crop.**random_crop**(*data*, *size*, *dist=0*)
Crop random patch/volume from input tensor

Parameters

- **data** (Tensor) – input tensor
- **size** (Union[int, Sequence[int]]) – size of patch/volume
- **dist** (Union[int, Sequence[int]]) – minimum distance to border. By default zero

Returns cropped output List[int]: top left corner used for crop

Return type torch.Tensor

11.3.1 crop

rising.transforms.functional.crop.**crop**(*data*, *corner*, *size*)
Extract crop from last dimensions of data

Args: data: input tensor corner: top left corner point size: size of patch

Returns cropped data

Return type torch.Tensor

11.3.2 center_crop

rising.transforms.functional.crop.**center_crop**(*data*, *size*)
Crop patch from center

Args: data: input tensor size: size of patch

Returns output tensor cropped from input tensor

Return type torch.Tensor

11.3.3 random_crop

rising.transforms.functional.crop.**random_crop**(*data*, *size*, *dist=0*)
Crop random patch/volume from input tensor

Parameters

- **data** (Tensor) – input tensor
- **size** (Union[int, Sequence[int]]) – size of patch/volume
- **dist** (Union[int, Sequence[int]]) – minimum distance to border. By default zero

Returns cropped output List[int]: top left corner used for crop

Return type torch.Tensor

11.4 Intensity Transforms

rising.transforms.functional.intensity.**norm_range**(*data*, *min*, *max*, *per_channel=True*, *out=None*)

Scale range of tensor

Parameters

- **data** (Tensor) – input data. Per channel option supports [C,H,W] and [C,H,W,D].
- **min** (float) – minimal value
- **max** (float) – maximal value
- **per_channel** (bool) – range is normalized per channel
- **out** (Optional[Tensor]) – if provided, result is saved in here

Returns normalized data

Return type torch.Tensor

rising.transforms.functional.intensity.**norm_min_max**(*data*, *per_channel=True*, *out=None*, *eps=1e-08*)

Scale range to [0,1]

Parameters

- **data** (Tensor) – input data. Per channel option supports [C,H,W] and [C,H,W,D].
- **per_channel** (bool) – range is normalized per channel
- **out** (Optional[Tensor]) – if provided, result is saved in here
- **eps** (Optional[float]) – small constant for numerical stability. If None, no factor constant will be added

Returns scaled data

Return type torch.Tensor

rising.transforms.functional.intensity.**norm_zero_mean_unit_std**(*data*, *per_channel=True*, *out=None*, *eps=1e-08*)

Normalize mean to zero and std to one

Parameters

- **data** (Tensor) – input data. Per channel option supports [C,H,W] and [C,H,W,D].
- **per_channel** (bool) – range is normalized per channel
- **out** (Optional[Tensor]) – if provided, result is saved in here
- **eps** (Optional[float]) – small constant for numerical stability. If None, no factor constant will be added

Returns normalized data

Return type torch.Tensor

rising.transforms.functional.intensity.**norm_mean_std**(*data*, *mean*, *std*, *per_channel=True*, *out=None*)

Normalize mean and std with provided values

Parameters

- **data** (Tensor) – input data. Per channel option supports [C,H,W] and [C,H,W,D].
- **mean** (Union[float, Sequence]) – used for mean normalization
- **std** (Union[float, Sequence]) – used for std normalization
- **per_channel** (bool) – range is normalized per channel
- **out** (Optional[Tensor]) – if provided, result is saved into out

Returns normalized data

Return type torch.Tensor

rising.transforms.functional.intensity.**add_noise**(*data*, *noise_type*, *out=None*, ***kwargs*)

Add noise to input

Parameters

- **data** (Tensor) – input data
- **noise_type** (str) – supports all inplace functions of a pytorch tensor
- **out** (Optional[Tensor]) – if provided, result is saved in here
- **kwargs** – keyword arguments passed to generating function

Returns data with added noise

Return type torch.Tensor

See also:

torch.Tensor.normal_(), torch.Tensor.exponential_()

rising.transforms.functional.intensity.**add_value**(*data*, *value*, *out=None*)

Increase brightness additively by value (currently this functions is intended as an interface in case additional functionality should be added to transform)

Parameters

- **data** (Tensor) – input data
- **value** (float) – additive value
- **out** (Optional[Tensor]) – if provided, result is saved in here

Returns augmented data

Return type torch.Tensor

rising.transforms.functional.intensity.**gamma_correction**(*data*, *gamma*)

Apply gamma correction to data (currently this functions is intended as an interface in case additional functionality should be added to transform)

Parameters

- **data** (Tensor) – input data
- **gamma** (float) – gamma for correction

Returns gamma corrected data

Return type torch.Tensor

rising.transforms.functional.intensity.**scale_by_value**(*data*, *value*, *out=None*)

Increase brightness scaled by value (currently this functions is intended as an interface in case additional functionality should be added to transform)

Parameters

- **data** (Tensor) – input data
- **value** (float) – scaling value
- **out** (Optional[Tensor]) – if provided, result is saved in here

Returns augmented data

Return type torch.Tensor

rising.transforms.functional.intensity.**clamp**(*data*, *min*, *max*, *out=None*)

Clamp tensor to minimal and maximal value

Parameters

- **data** (Tensor) – tensor to clamp
- **min** (float) – lower limit
- **max** (float) – upper limit
- **out** (Optional[Tensor]) – output tensor

Returns clamped tensor

Return type Tensor

11.4.1 norm_range

rising.transforms.functional.intensity.**norm_range**(*data*, *min*, *max*, *per_channel=True*, *out=None*)

Scale range of tensor

Parameters

- **data** (Tensor) – input data. Per channel option supports [C,H,W] and [C,H,W,D].
- **min** (float) – minimal value
- **max** (float) – maximal value
- **per_channel** (bool) – range is normalized per channel
- **out** (Optional[Tensor]) – if provided, result is saved in here

Returns normalized data

Return type torch.Tensor

11.4.2 norm_min_max

rising.transforms.functional.intensity.**norm_min_max**(*data*, *per_channel=True*, *out=None*, *eps=1e-08*)

Scale range to [0,1]

Parameters

- **data** (Tensor) – input data. Per channel option supports [C,H,W] and [C,H,W,D].
- **per_channel** (bool) – range is normalized per channel
- **out** (Optional[Tensor]) – if provided, result is saved in here

- **eps** (Optional[float]) – small constant for numerical stability. If None, no factor constant will be added

Returns scaled data

Return type torch.Tensor

11.4.3 norm_zero_mean_unit_std

rising.transforms.functional.intensity.**norm_zero_mean_unit_std**(*data*, *per_channel=True*, *out=None*, *eps=1e-08*)

Normalize mean to zero and std to one

Parameters

- **data** (Tensor) – input data. Per channel option supports [C,H,W] and [C,H,W,D].
- **per_channel** (bool) – range is normalized per channel
- **out** (Optional[Tensor]) – if provided, result is saved in here
- **eps** (Optional[float]) – small constant for numerical stability. If None, no factor constant will be added

Returns normalized data

Return type torch.Tensor

11.4.4 norm_mean_std

rising.transforms.functional.intensity.**norm_mean_std**(*data*, *mean*, *std*, *per_channel=True*, *out=None*)

Normalize mean and std with provided values

Parameters

- **data** (Tensor) – input data. Per channel option supports [C,H,W] and [C,H,W,D].
- **mean** (Union[float, Sequence]) – used for mean normalization
- **std** (Union[float, Sequence]) – used for std normalization
- **per_channel** (bool) – range is normalized per channel
- **out** (Optional[Tensor]) – if provided, result is saved into out

Returns normalized data

Return type torch.Tensor

11.4.5 add_noise

rising.transforms.functional.intensity.**add_noise**(*data*, *noise_type*, *out=None*, ***kwargs*)

Add noise to input

Parameters

- **data** (Tensor) – input data
- **noise_type** (str) – supports all inplace functions of a pytorch tensor
- **out** (Optional[Tensor]) – if provided, result is saved in here
- **kwargs** – keyword arguments passed to generating function

Returns data with added noise

Return type torch.Tensor

See also:

`torch.Tensor.normal_()`, `torch.Tensor.exponential_()`

11.4.6 add_value

rising.transforms.functional.intensity.**add_value**(*data*, *value*, *out=None*)

Increase brightness additively by value (currently this functions is intended as an interface in case additional functionality should be added to transform)

Parameters

- **data** (Tensor) – input data
- **value** (float) – additive value
- **out** (Optional[Tensor]) – if provided, result is saved in here

Returns augmented data

Return type torch.Tensor

11.4.7 gamma_correction

rising.transforms.functional.intensity.**gamma_correction**(*data*, *gamma*)

Apply gamma correction to data (currently this functions is intended as an interface in case additional functionality should be added to transform)

Parameters

- **data** (Tensor) – input data
- **gamma** (float) – gamma for correction

Returns gamma corrected data

Return type torch.Tensor

11.4.8 scale_by_value

rising.transforms.functional.intensity.**scale_by_value**(*data*, *value*, *out=None*)
Increase brightness scaled by value (currently this functions is intended as an interface in case additional functionality should be added to transform)

Parameters

- **data** (Tensor) – input data
- **value** (float) – scaling value
- **out** (Optional[Tensor]) – if provided, result is saved in here

Returns augmented data

Return type torch.Tensor

11.5 Spatial Transforms

rising.transforms.functional.spatial.**mirror**(*data*, *dims*)
Mirror data at dims

Parameters

- **data** (Tensor) – input data
- **dims** (Union[int, Sequence[int]]) – dimensions to mirror

Returns tensor with mirrored dimensions

Return type torch.Tensor

rising.transforms.functional.spatial.**rot90**(*data*, *k*, *dims*)
Rotate 90 degrees around dims

Parameters

- **data** (Tensor) – input data
- **k** (int) – number of times to rotate
- **dims** (Union[int, Sequence[int]]) – dimensions to mirror

Returns tensor with mirrored dimensions

Return type torch.Tensor

rising.transforms.functional.spatial.**resize_native**(*data*, *size=None*, *scale_factor=None*, *mode='nearest'*, *align_corners=None*, *preserve_range=False*)
Down/up-sample sample to either the given `size` or the given `scale_factor` The modes available for resizing are: nearest, linear (3D-only), bilinear, bicubic (4D-only), trilinear (5D-only), area

Parameters

- **data** (Tensor) – input tensor of shape batch x channels x height x width x [depth]
- **size** (Union[int, Sequence[int], None]) – spatial output size (excluding batch size and number of channels)

- **scale_factor** (Union[float, Sequence[float], None]) – multiplier for spatial size
- **mode** (str) – one of nearest, linear, bilinear, bicubic, trilinear, area (for more inforamtion see torch.nn.functional.interpolate())
- **align_corners** (Optional[bool]) – input and output tensors are aligned by the center points of their corners pixels, preserving the values at the corner pixels.
- **preserve_range** (bool) – output tensor has same range as input tensor

Returns interpolated tensor

Return type torch.Tensor

See also:

torch.nn.functional.interpolate()

11.5.1 mirror

rising.transforms.functional.spatial.**mirror**(*data*, *dims*)

Mirror data at dims

Parameters

- **data** (Tensor) – input data
- **dims** (Union[int, Sequence[int]]) – dimensions to mirror

Returns tensor with mirrored dimensions

Return type torch.Tensor

11.5.2 rot90

rising.transforms.functional.spatial.**rot90**(*data*, *k*, *dims*)

Rotate 90 degrees around dims

Parameters

- **data** (Tensor) – input data
- **k** (int) – number of times to rotate
- **dims** (Union[int, Sequence[int]]) – dimensions to mirror

Returns tensor with mirrored dimensions

Return type torch.Tensor

11.5.3 resize_native

rising.transforms.functional.spatial.**resize_native**(*data*, *size=None*, *scale_factor=None*, *mode='nearest'*, *align_corners=None*, *preserve_range=False*)

Down/up-sample sample to either the given `size` or the given `scale_factor` The modes available for resizing are: nearest, linear (3D-only), bilinear, bicubic (4D-only), trilinear (5D-only), area

Parameters

- **data** (Tensor) – input tensor of shape batch x channels x height x width x [depth]
- **size** (Union[int, Sequence[int], None]) – spatial output size (excluding batch size and number of channels)
- **scale_factor** (Union[float, Sequence[float], None]) – multiplier for spatial size
- **mode** (str) – one of `nearest`, `linear`, `bilinear`, `bicubic`, `trilinear`, `area` (for more inforamtion see `torch.nn.functional.interpolate()`)
- **align_corners** (Optional[bool]) – input and output tensors are aligned by the center points of their corners pixels, preserving the values at the corner pixels.
- **preserve_range** (bool) – output tensor has same range as input tensor

Returns interpolated tensor

Return type torch.Tensor

See also:

`torch.nn.functional.interpolate()`

11.6 Tensor Transforms

rising.transforms.functional.tensor.**tensor_op**(*data*, *fn*, **args*, ***kwargs*)

Invokes a function form a tensor

Parameters

- **data** (Union[Tensor, List[Tensor], Tuple[Tensor], Mapping[Hashable, Tensor]]) – data which should be pushed to device. Sequence and mapping items are mapping individually to gpu
- **fn** (str) – tensor function
- ***args** – positional arguments passed to tensor function
- ****kwargs** – keyword arguments passed to tensor function

Returns data which was pushed to device

Return type Union[torch.Tensor, Sequence, Mapping]

rising.transforms.functional.tensor.**to_device_dtype**(*data*, *dtype=None*, *device=None*, ***kwargs*)

Pushes data to device

Parameters

- **data** (Union[Tensor, List[Tensor], Tuple[Tensor], Mapping[Hashable, Tensor]]) – data which should be pushed to device. Sequence and mapping items are mapping individually to gpu
- **device** (Union[device, str, None]) – target device
- **kwargs** – keyword arguments passed to assigning function

Returns data which was pushed to device

Return type Union[torch.Tensor, Sequence, Mapping]

11.6.1 tensor_op

rising.transforms.functional.tensor.**tensor_op**(*data*, *fn*, **args*, ***kwargs*)

Invokes a function form a tensor

Parameters

- **data** (Union[Tensor, List[Tensor], Tuple[Tensor], Mapping[Hashable, Tensor]]) – data which should be pushed to device. Sequence and mapping items are mapping individually to gpu
- **fn** (str) – tensor function
- ***args** – positional arguments passed to tensor function
- ****kwargs** – keyword arguments passed to tensor function

Returns data which was pushed to device

Return type Union[torch.Tensor, Sequence, Mapping]

11.6.2 to_device_dtype

rising.transforms.functional.tensor.**to_device_dtype**(*data*, *dtype=None*, *device=None*, ***kwargs*)

Pushes data to device

Parameters

- **data** (Union[Tensor, List[Tensor], Tuple[Tensor], Mapping[Hashable, Tensor]]) – data which should be pushed to device. Sequence and mapping items are mapping individually to gpu
- **device** (Union[device, str, None]) – target device
- **kwargs** – keyword arguments passed to assigning function

Returns data which was pushed to device

Return type Union[torch.Tensor, Sequence, Mapping]

11.7 Utility Transforms

rising.transforms.functional.utility.**box_to_seg**(*boxes*, *shape=None*, *dtype=None*, *device=None*, *out=None*)

Convert a sequence of bounding boxes to a segmentation

Parameters

- **boxes** (Sequence[Sequence[int]]) – sequence of bounding boxes encoded as (dim0_min, dim1_min, dim0_max, dim1_max, [dim2_min, dim2_max]). Supported bounding boxes for 2D (4 entries per box) and 3d (6 entries per box)
- **shape** (Optional[Sequence[int]]) – if `out` is not provided, shape of output tensor must be specified
- **dtype** (Union[dtype, str, None]) – if `out` is not provided, dtype of output tensor must be specified
- **device** (Union[device, str, None]) – if `out` is not provided, device of output tensor must be specified
- **out** (Optional[Tensor]) – if not None, the segmentation will be saved inside this tensor

Returns bounding boxes encoded as a segmentation

Return type torch.Tensor

rising.transforms.functional.utility.**seg_to_box**(*seg*, *dim*)

Convert instance segmentation to bounding boxes

Parameters

- **seg** (Tensor) – segmentation of individual classes (index should start from one and be continuous)
- **dim** (int) – number of spatial dimensions

Returns

list of bounding boxes tuple with classes for bounding boxes

Return type list

rising.transforms.functional.utility.**instance_to_semantic**(*instance*, *cls*)

Convert an instance segmentation to a semantic segmentation

Parameters

- **instance** (Tensor) – instance segmentation of objects (objects need to start from 1, 0 background)
- **cls** (Sequence[int]) – mapping from indices from instance segmentation to real classes.

Returns semantic segmentation

Return type torch.Tensor

> **Warning:** `instance` needs to encode objects starting from 1 and the indices need to be continuous (0 is interpreted as background)

rising.transforms.functional.utility.**pop_keys**(*data*, *keys*, *return_popped=False*)

Pops keys from a given data dict

Parameters

- **data** (dict) – the dictionary to pop the keys from
- **keys** (Union[Callable, Sequence]) – if callable it must return a boolean for each key indicating whether it should be popped from the dict. if sequence of strings, the strings shall be the keys to be popped
- **return_popped** – whether to also return the popped values
- **(default** – False)

Returns the data without the popped values dict: the popped values; only if :attr`return_popped` is True

Return type dict

rising.transforms.functional.utility.**filter_keys**(*data*, *keys*, *return_popped=False*)

Filters keys from a given data dict

Parameters

- **data** (dict) – the dictionary to pop the keys from
- **keys** (Union[Callable, Sequence]) – if callable it must return a boolean for each key indicating whether it should be retained in the dict. if sequence of strings, the strings shall be the keys to be retained
- **return_popped** – whether to also return the popped values (default: False)

Returns the data without the popped values dict: the popped values; only if `return_popped` is True

Return type dict

11.7.1 box_to_seg

rising.transforms.functional.utility.**box_to_seg**(*boxes*, *shape=None*, *dtype=None*, *device=None*, *out=None*)

Convert a sequence of bounding boxes to a segmentation

Parameters

- **boxes** (Sequence[Sequence[int]]) – sequence of bounding boxes encoded as (dim0_min, dim1_min, dim0_max, dim1_max, [dim2_min, dim2_max]). Supported bounding boxes for 2D (4 entries per box) and 3d (6 entries per box)
- **shape** (Optional[Sequence[int]]) – if `out` is not provided, shape of output tensor must be specified
- **dtype** (Union[dtype, str, None]) – if `out` is not provided, dtype of output tensor must be specified
- **device** (Union[device, str, None]) – if `out` is not provided, device of output tensor must be specified
- **out** (Optional[Tensor]) – if not None, the segmentation will be saved inside this tensor

Returns bounding boxes encoded as a segmentation

Return type torch.Tensor

11.7.2 seg_to_box

rising.transforms.functional.utility.**seg_to_box**(*seg*, *dim*)
Convert instance segmentation to bounding boxes

Parameters

- **seg** (Tensor) – segmentation of individual classes (index should start from one and be continuous)
- **dim** (int) – number of spatial dimensions

Returns

list of bounding boxes tuple with classes for bounding boxes

Return type list

11.7.3 instance_to_semantic

rising.transforms.functional.utility.**instance_to_semantic**(*instance*, *cls*)
Convert an instance segmentation to a semantic segmentation

Parameters

- **instance** (Tensor) – instance segmentation of objects (objects need to start from 1, 0 background)
- **cls** (Sequence[int]) – mapping from indices from instance segmentation to real classes.

Returns semantic segmentation

Return type torch.Tensor

Warning: instance needs to encode objects starting from 1 and the indices need to be continuous (0 is interpreted as background)

11.7.4 pop_keys

rising.transforms.functional.utility.**pop_keys**(*data*, *keys*, *return_popped=False*)
Pops keys from a given data dict

Parameters

- **data** (dict) – the dictionary to pop the keys from
- **keys** (Union[Callable, Sequence]) – if callable it must return a boolean for each key indicating whether it should be popped from the dict. if sequence of strings, the strings shall be the keys to be popped
- **return_popped** – whether to also return the popped values
- **(default** – False)

Returns the data without the popped values dict: the popped values; only if :attr`return_popped` is True

Return type dict

11.7.5 filter_keys

rising.transforms.functional.utility.**filter_keys**(*data*, *keys*, *return_popped=False*)
Filters keys from a given data dict

Parameters

- **data** (dict) – the dictionary to pop the keys from
- **keys** (Union[Callable, Sequence]) – if callable it must return a boolean for each key indicating whether it should be retained in the dict. if sequence of strings, the strings shall be the keys to be retained
- **return_popped** – whether to also return the popped values (default: False)

Returns the data without the popped values dict: the popped values; only if return_popped is True

Return type dict

CHAPTER

TWELVE

RISING.UTILS

12.1 Affines

rising.utils.affine.**deg_to_rad**(*angles*)
Converts from degree to radians.

Parameters **angles** (Union[Tensor, float, int]) – the (vectorized) angles to convert

Returns the transformed (vectorized) angles

Return type torch.Tensor

rising.utils.affine.**get_batched_eye**(*batchsize*, *ndim*, *device=None*, *dtype=None*)
Produces a batched matrix containing 1s on the diagonal

Parameters

- **batchsize** (int) – int the batchsize (first dimension)
- **ndim** (int) – int the dimensionality of the eyes (second and third dimension)
- **device** (Union[device, str, None]) – torch.device, str, optional the device to put the resulting tensor to. Defaults to the default device
- **dtype** (Union[dtype, str, None]) – torch.dtype, str, optional the dtype of the resulting trensor. Defaults to the default dtype

Returns batched eye matrix

Return type torch.Tensor

rising.utils.affine.**matrix_revert_coordinate_order**(*batch*)
Reverts the coordinate order of a matrix (e.g. from xyz to zyx).

Parameters **batch** (Tensor) – the batched transformation matrices; Should be of shape BATCHSIZE x NDIM x NDIM

Returns

the matrix performing the same transformation on vectors with a reversed coordinate order

Return type torch.Tensor

rising.utils.affine.**matrix_to_cartesian**(*batch*, *keep_square=False*)
Transforms a matrix for a homogeneous transformation back to cartesian coordinates.

Parameters

- **batch** (Tensor) – the batch oif matrices to convert back

- **keep_square** (bool) – if False: returns a NDIM x NDIM+1 matrix to keep the translation part if True: returns a NDIM x NDIM matrix but looses the translation part. defaults to False.

Returns the given matrix in cartesian coordinates

Return type torch.Tensor

rising.utils.affine.**matrix_to_homogeneous**(*batch*)
Transforms a given transformation matrix to a homogeneous transformation matrix.

Parameters **batch** (Tensor) – the batch of matrices to convert [N, dim, dim]

Returns the converted batch of matrices

Return type torch.Tensor

rising.utils.affine.**points_to_cartesian**(*batch*)
Transforms a batch of points in homogeneous coordinates back to cartesian coordinates.

Parameters **batch** (Tensor) – batch of points in homogeneous coordinates. Should be of shape BATCHSIZE x NUMPOINTS x NDIM+1

Returns the batch of points in cartesian coordinates

Return type torch.Tensor

rising.utils.affine.**points_to_homogeneous**(*batch*)
Transforms points from cartesian to homogeneous coordinates

Parameters **batch** (Tensor) – the batch of points to transform. Should be of shape BATCHSIZE x NUMPOINTS x DIM.

Returns the batch of points in homogeneous coordinates

Return type torch.Tensor

rising.utils.affine.**unit_box**(*n*, *scale=None*)
Create a (scaled) version of a unit box

Parameters

- **n** (int) – number of dimensions
- **scale** (Optional[Tensor]) – scaling of each dimension

Returns scaled unit box

Return type torch.Tensor

12.1.1 points_to_homogeneous

rising.utils.affine.**points_to_homogeneous**(*batch*)
Transforms points from cartesian to homogeneous coordinates

Parameters **batch** (Tensor) – the batch of points to transform. Should be of shape BATCHSIZE x NUMPOINTS x DIM.

Returns the batch of points in homogeneous coordinates

Return type torch.Tensor

12.1.2 matrix_to_homogeneous

rising.utils.affine.**matrix_to_homogeneous**(*batch*)
Transforms a given transformation matrix to a homogeneous transformation matrix.

Parameters **batch** (Tensor) – the batch of matrices to convert [N, dim, dim]

Returns the converted batch of matrices

Return type torch.Tensor

12.1.3 matrix_to_cartesian

rising.utils.affine.**matrix_to_cartesian**(*batch*, *keep_square=False*)
Transforms a matrix for a homogeneous transformation back to cartesian coordinates.

Parameters

- **batch** (Tensor) – the batch oif matrices to convert back
- **keep_square** (bool) – if False: returns a NDIM x NDIM+1 matrix to keep the translation part if True: returns a NDIM x NDIM matrix but looses the translation part. defaults to False.

Returns the given matrix in cartesian coordinates

Return type torch.Tensor

12.1.4 points_to_cartesian

rising.utils.affine.**points_to_cartesian**(*batch*)
Transforms a batch of points in homogeneous coordinates back to cartesian coordinates.

Parameters **batch** (Tensor) – batch of points in homogeneous coordinates. Should be of shape BATCHSIZE x NUMPOINTS x NDIM+1

Returns the batch of points in cartesian coordinates

Return type torch.Tensor

12.1.5 matrix_revert_coordinate_order

rising.utils.affine.**matrix_revert_coordinate_order**(*batch*)
Reverts the coordinate order of a matrix (e.g. from xyz to zyx).

Parameters **batch** (Tensor) – the batched transformation matrices; Should be of shape BATCHSIZE x NDIM x NDIM

Returns

the matrix performing the same transformation on vectors with a reversed coordinate order

Return type torch.Tensor

12.1.6 get_batched_eye

rising.utils.affine.**get_batched_eye**(*batchsize*, *ndim*, *device=None*, *dtype=None*)
Produces a batched matrix containing 1s on the diagonal

Parameters

- **batchsize** (int) – int the batchsize (first dimension)
- **ndim** (int) – int the dimensionality of the eyes (second and third dimension)
- **device** (Union[device, str, None]) – torch.device, str, optional the device to put the resulting tensor to. Defaults to the default device
- **dtype** (Union[dtype, str, None]) – torch.dtype, str, optional the dtype of the resulting trensor. Defaults to the default dtype

Returns batched eye matrix

Return type torch.Tensor

12.1.7 deg_to_rad

rising.utils.affine.**deg_to_rad**(*angles*)
Converts from degree to radians.

Parameters **angles** (Union[Tensor, float, int]) – the (vectorized) angles to convert

Returns the transformed (vectorized) angles

Return type torch.Tensor

12.1.8 unit_box

rising.utils.affine.**unit_box**(*n*, *scale=None*)
Create a (scaled) version of a unit box

Parameters

- **n** (int) – number of dimensions
- **scale** (Optional[Tensor]) – scaling of each dimension

Returns scaled unit box

Return type torch.Tensor

12.2 Type Checks

rising.utils.checktype.**check_scalar**(*x*)
Provide interface to check for scalars

Parameters **x** (Union[Any, float, int]) – object to check for scalar

Return type bool

Returns bool" True if input is scalar

12.2.1 check_scalar

rising.utils.checktype.**check_scalar**(*x*)
Provide interface to check for scalars

Parameters **x** (Union[Any, float, int]) – object to check for scalar

Return type bool

Returns bool" True if input is scalar

12.3 Reshaping

rising.utils.shape.**reshape**(*value*, *size*)
Reshape sequence (list or tensor) to given size

Parameters

- **value** (Union[list, Tensor]) – sequence to reshape
- **size** (Union[Sequence, Size]) – size to reshape to

Returns reshaped sequence

Return type Union[torch.Tensor, list]

rising.utils.shape.**reshape_list**(*flat_list*, *size*)
Reshape a (nested) list to a given shape

Parameters

- **flat_list** (list) – (nested) list to reshape
- **size** (Union[Size, tuple]) – shape to reshape to

Returns reshape list

Return type list

12.3.1 reshape

rising.utils.shape.**reshape**(*value*, *size*)
Reshape sequence (list or tensor) to given size

Parameters

- **value** (Union[list, Tensor]) – sequence to reshape
- **size** (Union[Sequence, Size]) – size to reshape to

Returns reshaped sequence

Return type Union[torch.Tensor, list]

12.3.2 reshape_list

rising.utils.shape.**reshape_list**(*flat_list*, *size*)
Reshape a (nested) list to a given shape

Parameters

- **flat_list** (list) – (nested) list to reshape
- **size** (Union[Size, tuple]) – shape to reshape to

Returns reshape list

Return type list

CHAPTER

THIRTEEN

TUTORIALS & EXAMPLES

This Page contains a collection of curated tutorials and examples on how to use rising to its full extent.

13.1 Segmentation with `rising` and `PytorchLightning`

This example will show you how to build a proper training pipeline with PyTorch Lightning and rising. But first let's configure this notebook correctly:

```
[ ]: %reload_ext autoreload
     %autoreload 2
     %matplotlib inline
```

and install all our dependencies:

```
[ ]: !pip install --upgrade --quiet pytorch-lightning # for training
     !pip install --upgrade --quiet git+https://github.com/PhoenixDL/rising # for data␣
     ↪handling
     !pip install --upgrade --quiet SimpleITK # for loading medical data
     !pip install --upgrade --quiet tensorboard # for monitoring training
     !pip install --upgrade --quiet gdown # to download data cross platform
```

13.1.1 Data

Once this is done, we need to take care of our training data. To show risings full capabilities, we will be using 3D data from medical decathlon (specifically Task 4: Hippocampus).

Download

We will use the data provided on Google Drive and download it:

```
[ ]: import os
     import SimpleITK as sitk
     import json
     import tempfile
     import numpy as np
     import tarfile
     import time
     import gdown
```

(continues on next page)

(continued from previous page)

```
temp_dir = tempfile.mkdtemp()

# generate dummy data for ci/cd
if 'CI' in os.environ:
    data_dir = os.path.join(temp_dir, 'DummyData')
    os.makedirs(os.path.join(data_dir, 'training'), exist_ok=True)
    data_paths = []

    for idx in range(50):
        img = np.random.randint(-500, 500, (32, 64, 32), np.int16)
        mask = np.random.randint(0, 1, (32, 64, 32), np.int16)

        img_path = os.path.join(data_dir, 'training', 'img_%03d.nii.gz' % idx)
        mask_path = os.path.join(data_dir, 'training', 'mask_%03d.nii.gz' % idx)
        sitk.WriteImage(sitk.GetImageFromArray(img), img_path)
        sitk.WriteImage(sitk.GetImageFromArray(mask), mask_path)

        data_paths.append({'image': img_path, 'label': mask_path})

    with open(os.path.join(data_dir, 'dataset.json'), 'w') as f:
        json.dump({'training': data_paths}, f, sort_keys=True, indent=4)

else:

    data_url = "https://drive.google.com/uc?export=download&id=1RzPB1_bqzQhlWvU-
↪YGvZzhx2omcDh38C"

    data_dir = os.path.join(temp_dir, 'Task04_Hippocampus')
    download_path = os.path.join(temp_dir, 'data.tar')

    gdown.download(data_url, download_path)

    tarfile.TarFile(download_path).extractall(temp_dir)
```

Great! We got our data. Now we can work on loading it. For loading data, `rising` follows the same principle as PyTorch: It separates the dataset, which provides the logic of loading a single sample, from the dataloader for automatted handling of parallel loading and batching.

In fact we at `rising` thought that there is no need to reinvent the wheel. This is why we internally use PyTorch's data structure and just extend it a bit. We'll come to these extensions later.

Dataset

Our dataset is fairly simple. It just loads the Nifti Data we downloaded before and returns each sample as a dict:

```
[ ]: import SimpleITK as sitk
import json
from rising import loading
from rising.loading import Dataset
import torch
class NiiDataset(Dataset):
    def __init__(self, train: bool, data_dir: str):
        """
        Args:
```

(continues on next page)

(continued from previous page)

```
            train: whether to use the training or the validation split
            data_dir: directory containing the data
        """
        with open(os.path.join(data_dir, 'dataset.json')) as f:
            content = json.load(f)['training']

            num_train_samples = int(len(content) * 0.9)

            # Split train data into training and validation,
            # since test data contains no ground truth
            if train:
                data = content[:num_train_samples]
            else:
                data = content[num_train_samples:]

            self.data = data
            self.data_dir = data_dir

    def __getitem__(self, item: int) -> dict:
        """
        Loads and Returns a single sample

        Args:
            item: index specifying which item to load

        Returns:
            dict: the loaded sample
        """
        sample = self.data[item]
        img = sitk.GetArrayFromImage(
            sitk.ReadImage(os.path.join(self.data_dir, sample['image'])))

        # add channel dim if necesary
        if img.ndim == 3:
            img = img[None]

        label = sitk.GetArrayFromImage(
            sitk.ReadImage(os.path.join(self.data_dir, sample['label'])))

        # convert multiclass to binary task by combining all positives
        label = label > 0

        # add channel dim if necessary
        if label.ndim == 3:
            label = label[None]
        return {'data': torch.from_numpy(img).float(),
                'label': torch.from_numpy(label).float()}

    def __len__(self) -> int:
        """
        Adds a length to the dataset

        Returns:
            int: dataset's length
        """
        return len(self.data)
```

For compatibility each `rising` dataset must hold the same attributes as a `PyTorch` dataset. This basically comes down to be indexeable. This means, each Sequence-like data (e.g. lists, tuples, tensors or arrays) could also directly be used as a dataset. Ideally each dataset also has a length, since the dataloader tries to use this length to calculate/estimate its own length.

13.1.2 Integration With PyTorch Lightning: Model and Training

After obtaining our data and implementing a way to load it, we now need a model we can train. For this, we will use a fairly simple implementation of the U-Net, which basically is an encoder-decoder network with skip connections. In `Lightning` all modules should be derived from a `LightningModule`, which itself is a subclass of the `torch.nn.Module`. For further details on the `LightningModule` please refer to the project itself or it's documentation.

Model

For now we will only define the network's logic and omit the training logic, which we'll add later.

[ ]:
```
import pytorch_lightning as pl
import torch

class Unet(pl.LightningModule):
    """Simple U-Net without training logic"""
    def __init__(self, hparams: dict):
        """
        Args:
            hparams: the hyperparameters needed to construct the network.
                Specifically these are:
                * start_filts (int)
                * depth (int)
                * in_channels (int)
                * num_classes (int)
        """
        super().__init__()
        # 4 downsample layers
        out_filts = hparams.get('start_filts', 16)
        depth = hparams.get('depth', 3)
        in_filts = hparams.get('in_channels', 1)
        num_classes = hparams.get('num_classes', 2)

        for idx in range(depth):
            down_block = torch.nn.Sequential(
                torch.nn.Conv3d(in_filts, out_filts, kernel_size=3, padding=1),
                torch.nn.ReLU(inplace=True),
                torch.nn.Conv3d(out_filts, out_filts, kernel_size=3, padding=1),
                torch.nn.ReLU(inplace=True)
            )
            in_filts = out_filts
            out_filts *= 2

            setattr(self, 'down_block_%d' % idx, down_block)

        out_filts = out_filts // 2
        in_filts = in_filts // 2
        out_filts, in_filts = in_filts, out_filts

        for idx in range(depth-1):
```

(continues on next page)

(continued from previous page)

```
            up_block = torch.nn.Sequential(
                torch.nn.Conv3d(in_filts + out_filts, out_filts, kernel_size=3,␣
↪padding=1),
                torch.nn.ReLU(inplace=True),
                torch.nn.Conv3d(out_filts, out_filts, kernel_size=3, padding=1),
                torch.nn.ReLU(inplace=True)
            )

            in_filts = out_filts
            out_filts = out_filts // 2

            setattr(self, 'up_block_%d' % idx, up_block)

        self.final_conv = torch.nn.Conv3d(in_filts, num_classes, kernel_size=1)
        self.max_pool = torch.nn.MaxPool3d(2, stride=2)
        self.up_sample = torch.nn.Upsample(scale_factor=2)
        self.hparams = hparams

    def forward(self, input_tensor: torch.Tensor) -> torch.Tensor:
        """
        Forwards the :attr`input_tensor` through the network to obtain a prediction

        Args:
            input_tensor: the network's input

        Returns:
            torch.Tensor: the networks output given the :attr`input_tensor`
        """
        depth = self.hparams.get('depth', 3)

        intermediate_outputs = []

        # Compute all the encoder blocks' outputs
        for idx in range(depth):
            intermed = getattr(self, 'down_block_%d' % idx)(input_tensor)
            if idx < depth - 1:
                # store intermediate values for usage in decoder
                intermediate_outputs.append(intermed)
                input_tensor = getattr(self, 'max_pool')(intermed)
            else:
                input_tensor = intermed

        # Compute all the decoder blocks' outputs
        for idx in range(depth-1):
            input_tensor = getattr(self, 'up_sample')(input_tensor)

            # use intermediate values from encoder
            from_down = intermediate_outputs.pop(-1)
            intermed = torch.cat([input_tensor, from_down], dim=1)
            input_tensor = getattr(self, 'up_block_%d' % idx)(intermed)

        return getattr(self, 'final_conv')(input_tensor)
```

Okay, that was easy, right? Now let's just check if everything in our network is fine:

```
[ ]: net = Unet({'num_classes': 2, 'in_channels': 1, 'depth': 2, 'start_filts': 2})
     print(net(torch.rand(1, 1, 16, 16, 16)).shape)
```

So what did we do here? We initialized a network accepting input images with one channel. This network will then predict a segmentation map for 2 classes (of which one is the background class). It does so with 3 resolution stages.

When we tested the network, we forwarded a tensor with random values of size `(1, 1, 16, 16, 16)` through it. The first `1` here is the batch dim, the second `1` the channel dim (as we specified one input channel) and the three `16` are the spatial dimension (depth, height and width).

The output has the same dimensons except the channel dimension now holding `2` channels (one per class).

Training Criterions and Metrics

For training we will use the combination of CrossEntropyLoss and the SoftDiceLoss (see below).

For more details on this, I'd recommend Jeremy Jordan's Blog on semantic segmentation.

[ ]:
```
import rising
from typing import Sequence, Optional, Union
import torch

# Taken from https://github.com/justusschock/dl-utils/blob/master/dlutils/losses/soft_
↪dice.py
class SoftDiceLoss(torch.nn.Module):
    """Soft Dice Loss"""
    def __init__(self, square_nom: bool = False,
                 square_denom: bool = False,
                 weight: Optional[Union[Sequence, torch.Tensor]] = None,
                 smooth: float = 1.):
        """
        Args:
            square_nom: whether to square the nominator
            square_denom: whether to square the denominator
            weight: additional weighting of individual classes
            smooth: smoothing for nominator and denominator

        """
        super().__init__()
        self.square_nom = square_nom
        self.square_denom = square_denom

        self.smooth = smooth

        if weight is not None:
            if not isinstance(weight, torch.Tensor):
                weight = torch.tensor(weight)

            self.register_buffer("weight", weight)
        else:
            self.weight = None

    def forward(self, predictions: torch.Tensor, targets: torch.Tensor) -> torch.
↪Tensor:
        """
        Computes SoftDice Loss

        Args:
            predictions: the predictions obtained by the network
            targets: the targets (ground truth) for the :attr:`predictions`
```

(continues on next page)

(continued from previous page)

```
        Returns:
            torch.Tensor: the computed loss value
        """
        # number of classes for onehot
        n_classes = predictions.shape[1]
        with torch.no_grad():
            targets_onehot = rising.transforms.functional.channel.one_hot_batch(
                targets.unsqueeze(1), num_classes=n_classes)
        # sum over spatial dimensions
        dims = tuple(range(2, predictions.dim()))

        # compute nominator
        if self.square_nom:
            nom = torch.sum((predictions * targets_onehot.float()) ** 2, dim=dims)
        else:
            nom = torch.sum(predictions * targets_onehot.float(), dim=dims)
        nom = 2 * nom + self.smooth

        # compute denominator
        if self.square_denom:
            i_sum = torch.sum(predictions ** 2, dim=dims)
            t_sum = torch.sum(targets_onehot ** 2, dim=dims)
        else:
            i_sum = torch.sum(predictions, dim=dims)
            t_sum = torch.sum(targets_onehot, dim=dims)

        denom = i_sum + t_sum.float() + self.smooth

        # compute loss
        frac = nom / denom

        # apply weight for individual classesproperly
        if self.weight is not None:
            frac = self.weight * frac

        # average over classes
        frac = - torch.mean(frac, dim=1)

        return frac
```

Okay, now that we are able to properly calculate the loss function, we still lack a metric to monitor, that describes our performance. For segmentation tasks, this usually comes down to the dice coefficient. So let's implement this one as well:

[ ]:
```
# Taken from https://github.com/justusschock/dl-utils/blob/master/dlutils/metrics/
↪dice.py
def binary_dice_coefficient(pred: torch.Tensor, gt: torch.Tensor,
                            thresh: float = 0.5, smooth: float = 1e-7) -> torch.
↪Tensor:
    """
    computes the dice coefficient for a binary segmentation task

    Args:
        pred: predicted segmentation (of shape Nx(Dx)HxW)
```

(continues on next page)

(continued from previous page)

```
        gt: target segmentation (of shape NxCx(Dx)HxW)
        thresh: segmentation threshold
        smooth: smoothing value to avoid division by zero

    Returns:
        torch.Tensor: dice score
    """

    assert pred.shape == gt.shape

    pred_bool = pred > thresh

    intersec = (pred_bool * gt).float()
    return 2 * intersec.sum() / (pred_bool.float().sum()
                                 + gt.float().sum() + smooth)
```

Neat! So far we defined all criterions and metrics necessary for proper training and monitoring. But there are still two major parts of our pipeline missing:

1.) Data Preprocessing and Augmentation

2.) what to do for parameter update

Let's deal with the first point now.

Data Preprocessing

Since all samples in our dataset are of different size, we cannot collate them to a batch directly. Instead we need to resize them. Frameworks like torchvision do this inside the dataset. With `rising` however, we opted for moving this part outside the dataset (but still apply it on each sample separately before batching) for these reasons.

1.) The dataset get's more reusable for different settings

2.) The transforms don't have to be implemented into each dataset, which means it is easier to switch datasets without code duplication

3.) Applying different transforms is as easy as changing an argument of the loader; no need to deal with this manually in the dataset

This kind of transforms kann be passed to the dataloader with `sample_transforms`. If you have an implementation that usually works on batched data, we got you. All you need to do is specifying `pseudo_batch_dim` and we will take care of the rest. We will then automatically add a pseudo batch dim to all kind of data (tensors, arrays and all kind of built-in python containers containing a mixture thereof) before applying these transforms and remove it afterwards.

For now, we use our batched implementation of native torch resizing:

[ ]:
```
from rising.transforms import Compose, ResizeNative

def common_per_sample_trafos():
        return Compose(ResizeNative(size=(32, 64, 32), keys=('data',), mode='trilinear
↪'),
                       ResizeNative(size=(32, 64, 32), keys=('label',), mode='nearest
↪'))
```

Data Augmentation

Now that we have defined our preprocessing, let's come to data augmentation. To enrich our dataset, we randomly apply an affine. While `rising` already contains an implementation of Affine transforms that can also handle random inputs pretty well, we will implement a basic random parameter sampling by ourselves, since this also serves as an educational example.

Basically this is really straight forward. We just derive the `BaseAffine` class, overwrite the way the matrix is assembled by adding the sampling before we call the actual assembly method. We leave the rest to the already defined class:

[ ]:
```
from rising.transforms.affine import BaseAffine
import random
from typing import Optional, Sequence

class RandomAffine(BaseAffine):
    """Base Affine with random parameters for scale, rotation and translation"""
    def __init__(self, scale_range: Optional[tuple] = None,
                 rotation_range: Optional[tuple] = None,
                 translation_range: Optional[tuple] = None,
                 degree: bool = True,
                 image_transform: bool = True,
                 keys: Sequence = ('data',),
                 grad: bool = False,
                 output_size: Optional[tuple] = None,
                 adjust_size: bool = False,
                 interpolation_mode: str = 'nearest',
                 padding_mode: str = 'zeros',
                 align_corners: bool = False,
                 reverse_order: bool = False,
                 **kwargs,):

        """
        Args:
            scale_range: tuple containing minimum and maximum values for scale.
                Actual values will be sampled from uniform distribution with these
                constraints.
            rotation_range: tuple containing minimum and maximum values for rotation.
                Actual values will be sampled from uniform distribution with these
                constraints.
            translation_range: tuple containing minimum and maximum values for
↪translation.
                Actual values will be sampled from uniform distribution with these
                constraints.
            keys: keys which should be augmented
            grad: enable gradient computation inside transformation
            degree: whether the given rotation(s) are in degrees.
                Only valid for rotation parameters, which aren't passed
                as full transformation matrix.
            output_size: if given, this will be the resulting image size.
                Defaults to ``None``
            adjust_size: if True, the resulting image size will be
                calculated dynamically to ensure that the whole image fits.
            interpolation_mode: interpolation mode to calculate output values
                ``'bilinear'`` | ``'nearest'``. Default: ``'bilinear'``
            padding_mode: padding mode for outside grid values
                ``'zeros'`` | ``'border'`` | ``'reflection'``.
                Default: ``'zeros'``
```

(continues on next page)

(continued from previous page)

```
            align_corners: Geometrically, we consider the pixels of the
                input as squares rather than points. If set to True,
                the extrema (-1 and 1)  are considered as referring to the
                center points of the input's corner pixels. If set to False,
                they are instead considered as referring to the corner points
                of the input's corner pixels, making the sampling more
                resolution agnostic.
            reverse_order: reverses the coordinate order of the
                transformation to conform to the pytorch convention:
                transformation params order [W,H(,D)] and
                batch order [(D,)H,W]
            **kwargs: additional keyword arguments passed to the
                affine transf
        """
        super().__init__(scale=None, rotation=None, translation=None,
                         degree=degree,
                         image_transform=image_transform,
                         keys=keys,
                         grad=grad,
                         output_size=output_size,
                         adjust_size=adjust_size,
                         interpolation_mode=interpolation_mode,
                         padding_mode=padding_mode,
                         align_corners=align_corners,
                         reverse_order=reverse_order,
                         **kwargs)

        self.scale_range = scale_range
        self.rotation_range = rotation_range
        self.translation_range = translation_range

    def assemble_matrix(self, **data) -> torch.Tensor:
        """
        Samples Parameters for scale, rotation and translation
        before actual matrix assembly.

        Args:
            **data: dictionary containing a batch

        Returns:
            torch.Tensor: assembled affine matrix
        """
        ndim = data[self.keys[0]].ndim - 2

        if self.scale_range is not None:
            self.scale = [random.uniform(*self.scale_range) for _ in range(ndim)]

        if self.translation_range is not None:
            self.translation = [random.uniform(*self.translation_range) for _ in␣
↪range(ndim)]

        if self.rotation_range is not None:
            if ndim == 3:
                self.rotation = [random.uniform(*self.rotation_range) for _ in␣
↪range(ndim)]
            elif ndim == 1:
                self.rotation = random.uniform(*self.rotation_range)
```

(continues on next page)

(continued from previous page)

```
        return super().assemble_matrix(**data)
```

Also not that hard... So, now we have a custom implementation of a randomly parametrized affine transformation. This is all we will use as data augmentation for now.

Batched Transforms that shall be executed on CPU in a multiprocessed way should be specified to the dataloader as `batch_transforms`. If they should be executed on GPU, you can pass them as `gpu_transforms`. Unfortnuately it is not possible to add GPU transforms in a multiprocessing environment. Thus the internal computation order is like this:

1.) Extract sample from dataset

2.) Apply per-sample transforms to it (with or without pseudo batch dim)

3.) Collate to batch

4.) Apply batch transforms

5.) Apply GPU transforms

Steps 1.-4. can be executed in a multiprocessing environment. If this is the case, the results will be synced back to the main process before applying GPU transforms.

Training Logic

The only remaining step is now to integrate this to the training logic of `PyTorchLightning`.

The only things we did not yet discuss is how to setup optimizers, logging and train/validation step.

The optimizer setup is done by a function `configure_optimizers` that should return the created optimizers.

Logging can either be done automatically (all values for the key `log` in the dict returned from `validation_epoch_end` and `training_epoch_end` will autoamtically be logged) or manually (explicitly calling the logger in any of these functions). We show both examples below.

For setting up the actual training logic we need to specify `training_step` (and `validation_step` for validation). The complete example is below:

[ ]:
```
from rising.transforms import NormZeroMeanUnitStd
from rising.loading import DataLoader
import torch
from tqdm import tqdm

class TrainableUNet(Unet):
    """A trainable UNet (extends the base class by training logic)"""
    def __init__(self, hparams: Optional[dict] = None):
        """
        Args:
            hparams: the hyperparameters needed to construct and train the network.
                Specifically these are:
                * start_filts (int)
                * depth (int)
                * in_channels (int)
                * num_classes (int)
                * min_scale (float)
                * max_scale (float)
                * min_rotation (int, float)
                * max_rotation (int, float)
```

(continues on next page)

(continued from previous page)

```
                * batch_size (int)
                * num_workers(int)
                * learning_rate (float)

                For all of them exist usable default parameters.
        """
        if hparams is None:
            hparams = {}
        super().__init__(hparams)

        # define loss functions
        self.dice_loss = SoftDiceLoss(weight=[0., 1.])
        self.ce_loss = torch.nn.CrossEntropyLoss()

    def train_dataloader(self) -> DataLoader:
        """
        Specifies the train dataloader

        Returns:
            DataLoader: the train dataloader
        """
        # construct dataset
        dataset = NiiDataset(train=True, data_dir=data_dir)

        # specify batch transforms
        batch_transforms = Compose([
            RandomAffine(scale_range=(self.hparams.get('min_scale', 0.9), self.
↪hparams.get('max_scale', 1.1)),
                         rotation_range=(self.hparams.get('min_rotation', -10), self.
↪hparams.get('max_rotation', 10)),
                        keys=('data', 'label')),
            NormZeroMeanUnitStd(keys=('data',))
        ])

        # construct loader
        dataloader = DataLoader(dataset,
                                batch_size=self.hparams.get('batch_size', 1),
                                batch_transforms=batch_transforms,
                                shuffle=True,
                                sample_transforms=common_per_sample_trafos(),
                                pseudo_batch_dim=True,
                                num_workers=self.hparams.get('num_workers', 4))
        return dataloader

    def val_dataloader(self) -> DataLoader:
        # construct dataset
        dataset = NiiDataset(train=False, data_dir=data_dir)

        # specify batch transforms (no augmentation here)
        batch_transforms = NormZeroMeanUnitStd(keys=('data',))

        # construct loader
        dataloader = DataLoader(dataset,
                                batch_size=self.hparams.get('batch_size', 1),
                                batch_transforms=batch_transforms,
                                shuffle=False,
                                sample_transforms=common_per_sample_trafos(),
```

(continues on next page)

(continued from previous page)

```
                                pseudo_batch_dim=True,
                                num_workers=self.hparams.get('num_workers', 4))

        return dataloader

    def configure_optimizers(self) -> torch.optim.Optimizer:
        """
        Configures the optimier to use for training

        Returns:
            torch.optim.Optimier: the optimizer for updating the model's parameters
        """
        return torch.optim.Adam(self.parameters(), lr=self.hparams.get('learning_rate
↪', 1e-3))

    def training_step(self, batch: dict, batch_idx: int) -> dict:
        """
        Defines the training logic

        Args:
            batch: contains the data (inputs and ground truth)
            batch_idx: the number of the current batch

        Returns:
            dict: the current loss value
        """
        x, y = batch['data'], batch['label']

        # remove channel dim from gt (was necessary for augmentation)
        y = y[:, 0].long()

        # obtain predictions
        pred = self(x)
        softmaxed_pred = torch.nn.functional.softmax(pred, dim=1)

        # Calculate losses
        ce_loss = self.ce_loss(pred, y)
        dice_loss = self.dice_loss(softmaxed_pred, y)
        total_loss = (ce_loss + dice_loss) / 2

        # calculate dice coefficient
        dice_coeff = binary_dice_coefficient(torch.argmax(softmaxed_pred, dim=1), y)

        # log values
        self.logger.experiment.add_scalar('Train/DiceCoeff', dice_coeff)
        self.logger.experiment.add_scalar('Train/CE', ce_loss)
        self.logger.experiment.add_scalar('Train/SoftDiceLoss', dice_loss)
        self.logger.experiment.add_scalar('Train/TotalLoss', total_loss)

        return {'loss': total_loss}

    def validation_step(self, batch: dict, batch_idx: int) -> dict:
        """
        Defines the validation logic

        Args:
            batch: contains the data (inputs and ground truth)
```

(continues on next page)

(continued from previous page)

```
            batch_idx: the number of the current batch

        Returns:
            dict: the current loss and metric values
        """
        x, y = batch['data'], batch['label']

        # remove channel dim from gt (was necessary for augmentation)
        y = y[:, 0].long()

        # obtain predictions
        pred = self(x)
        softmaxed_pred = torch.nn.functional.softmax(pred, dim=1)

        # calculate losses
        ce_loss = self.ce_loss(pred, y)
        dice_loss = self.dice_loss(softmaxed_pred, y)
        total_loss = (ce_loss + dice_loss) / 2

        # calculate dice coefficient
        dice_coeff = binary_dice_coefficient(torch.argmax(softmaxed_pred, dim=1), y)

        # log values
        self.logger.experiment.add_scalar('Val/DiceCoeff', dice_coeff)
        self.logger.experiment.add_scalar('Val/CE', ce_loss)
        self.logger.experiment.add_scalar('Val/SoftDiceLoss', dice_loss)
        self.logger.experiment.add_scalar('Val/TotalLoss', total_loss)

        return {'val_loss': total_loss, 'dice': dice_coeff}

    def validation_epoch_end(self, outputs: list) -> dict:
        """Aggregates data from each validation step

        Args:
            outputs: the returned values from each validation step

        Returns:
            dict: the aggregated outputs
        """
        mean_outputs = {}
        for k in outputs[0].keys():
            mean_outputs[k] = torch.stack([x[k] for x in outputs]).mean()

        tqdm.write('Dice: \t%.3f' % mean_outputs['dice'].item())
        return mean_outputs
```

Most of this stuff is relevant for `PyTorch Lightning`. But the dataloader setup nicely shows the integration of `rising` with any existing framework working on `PyTorch Dataloaders` (like `PyTorch Lightning` or `PyTorch Ignite`) for batched and sample transforms.

13.1.3 Training

We've finally finished all the pipeline definition. Now let's just load the tensorboard extension to monitor our training. For this we will define a common output dir for lightning:

[ ]:
```
output_dir = 'logs'
os.makedirs(output_dir, exist_ok=True)
```

[ ]:
```
# Start tensorboard.

%reload_ext tensorboard
%tensorboard --logdir {output_dir}
```

And now it's finally time to train!

On a GPU in colab, training takes approximately 40 seconds per epoch, which is a total of 33 minutes (2000 seconds) for training, if early stopping doesn't kick in. For me it kicks in after 25 Epochs which takes around 16 Minutes on a colab GPU

[ ]:
```
from pytorch_lightning.callbacks import EarlyStopping
from pytorch_lightning import Trainer

early_stop_callback = EarlyStopping(monitor='dice', min_delta=0.001, patience=10,␣
↪verbose=False, mode='max')


if torch.cuda.is_available():
    gpus = 1
else:
    gpus = None

nb_epochs = 50
num_start_filts = 16
num_workers = 4

if 'CI' in os.environ:
    nb_epochs = 1
    num_start_filts = 2
    num_workers = 0

model = TrainableUNet({'start_filts': num_start_filts, 'num_workers': num_workers})

trainer = Trainer(gpus=gpus, default_save_path=output_dir, early_stop_callback=early_
↪stop_callback, max_nb_epochs=nb_epochs)
trainer.fit(model)
```

In the end, you should see a dice coefficient of 0.88 after 25 Epochs.

[ ]:
```
```

13.2 2D Classification Example on MedNIST and rising

Welcome to this rising example, where we will build a 2D classification pipeline with rising and pyorch lightning. The dataset part of this notebook was inspired by the Monai MedNIST example, so make sure to check them out, too :D

13.2.1 Preparation

Let's start with some basic preparations of our environment and download the MedNIST data.

First, we will install rising's master branch to get the latest features (if your a not planning to extend rising you can easily install out pypi package with `pip install rising`).

[ ]:
```
!pip install --upgrade --quiet git+https://github.com/PhoenixDL/rising # for data␣
↪handling
!pip install --upgrade --quiet pytorch-lightning # for easy training
!pip install --upgrade --quiet scikit-learn # for classification metrics
```

Next, we will add some magic to our notebook in case your are running them locally and do not want refresh it all the time.

[ ]:
```
%reload_ext autoreload
%autoreload 2
%matplotlib inline
```

Finally, we download the MedNIST dataset and undpack it.

[ ]:
```
import os

# Only check after the else statement for the data download :)
if 'CI' in os.environ:
    # our notebooks are executed to test our example
    # for this we need to create some dummy data
    import matplotlib.pyplot as plt
    from pathlib import Path
    import numpy as np
    from PIL import Image

    # create dummy data for our CI
    base_dir = Path("./MedNIST")
    base_dir.mkdir(exist_ok=True)
    cls_path1 = base_dir / "AbdomenCT"
    cls_path1.mkdir(exist_ok=True)
    cls_path2 = base_dir / "BreastMRI"
    cls_path2.mkdir(exist_ok=True)

    for i in range(100):
        np_array = np.zeros((64, 64)).astype(np.uint8)
        img = Image.fromarray(np_array)
        img.save(cls_path1 / f"img{i}.png")
        # plt.imsave(str(cls_path1 / f"img{i}.png"), np_array, cmap='Greys')
    for i in range(100):
        np_array = np.ones((64, 64)).astype(np.uint8)
        img = Image.fromarray(np_array)
        img.save(cls_path2 / f"img{i}.png")
        # plt.imsave(str(cls_path2 / f"img{i}.png"), np_array, cmap='Greys')
else:
```

(continues on next page)

(continued from previous page)

```
    # download MedNIST
    !curl -L -o MedNIST.tar.gz 'https://www.dropbox.com/s/5wwskxctvcxiuea/MedNIST.tar.
↪gz'

    # unzip the '.tar.gz' file to the current directory
    import tarfile
    datafile = tarfile.open("MedNIST.tar.gz")
    datafile.extractall()
    datafile.close()
```

13.2.2 Preparing our datasets

If you already wrote your own datasets with PyTorch this well be very familiar because `rising` uses the same dataset structure as PyTorch. The only difference between native PyTorch and `rising` is the transformation part. While PyTorch embeds its transformation into the dataset, we opted to move the transformations to our dataloder (which is a direct subclass of PyTorch's dataloader) to make our datasets easily interchangeable between multiple tasks.

Let's start by searching for the paths of the image files and defining their classes.

[ ]:
```
import os
from pathlib import Path
from PIL import Image

data_dir = Path('./MedNIST/')
class_names = sorted([p.stem for p in data_dir.iterdir() if p.is_dir()])
num_class = len(class_names)

image_files = [[x for x in (data_dir / class_name).iterdir()] for class_name in class_
↪names]

image_file_list = []
image_label_list = []
for i, class_name in enumerate(class_names):
    image_file_list.extend(image_files[i])
    image_label_list.extend([i] * len(image_files[i]))

num_total = len(image_label_list)

print('Total image count:', num_total)
print("Label names:", class_names)
print("Label counts:", [len(image_files[i]) for i in range(num_class)])
```

The output should look like this:

```
Total image count: 58954
Label names: ['AbdomenCT', 'BreastMRI', 'CXR', 'ChestCT', 'Hand', 'HeadCT']
Label counts: [10000, 8954, 10000, 10000, 10000, 10000]
```

The downloaded data needs to be divided into 3 subsets for training, validation and testing. Because the dataset is fairly large we can opt for an 80/10/10 split.

[ ]:
```
import numpy as np

valid_frac, test_frac = 0.1, 0.1
trainX, trainY = [], []
```

(continues on next page)

(continued from previous page)

```
valX, valY = [], []
testX, testY = [], []

for i in range(num_total):
    rann = np.random.random()
    if rann < valid_frac:
        valX.append(image_file_list[i])
        valY.append(image_label_list[i])
    elif rann < test_frac + valid_frac:
        testX.append(image_file_list[i])
        testY.append(image_label_list[i])
    else:
        trainX.append(image_file_list[i])
        trainY.append(image_label_list[i])

print("Training count =",len(trainX),"Validation count =", len(valX), "Test count =",
↪len(testX))
```

The MedNIST dataset now just needs to load the specified files. We use PIL to load the individual image file and convert it to a tensor afterwards.

[ ]:
```
import torch
import numpy as np

from typing import Sequence, Dict
from torch.utils.data import Dataset


class MedNISTDataset(Dataset):
  """
  Simple dataset to load individual samples from the dataset
  """

  def __init__(self, image_files: Sequence[str], labels: Sequence[int]):
    """
    Args:
      image_files: paths to the image files
      labels: label for each file
    """
    assert len(image_files) == len(labels), "Every file needs a label"
    self.image_files = image_files
    self.labels = labels

  def __len__(self) -> int:
    """
    Number of samples inside the dataset

    Returns:
      int: length
    """
    return len(self.image_files)

  def __getitem__(self, index: int) -> Dict[str, torch.Tensor]:
    """
    Select an individual sample from the dataset
```

(continues on next page)

(continued from previous page)

```
    Args:
      index: index of sample to draw

    Return:
      Dict[str, torch.Tensor]: single sample
        * `data`: image data
        * `label`: label for sample
    """
    data_np = np.array(Image.open(self.image_files[index]))
    return {"data": torch.from_numpy(data_np)[None].float(),
            "label": torch.tensor(self.labels[index]).long()}

train_ds = MedNISTDataset(trainX, trainY)
val_ds = MedNISTDataset(valX, valY)
test_ds = MedNISTDataset(testX, testY)
```

Let see some basic statistics of a single sample.

```
[ ]: print(f'Single image min: {train_ds[0]["data"].min()}')
print(f'Single image max: {train_ds[0]["data"].max()}')
print(f'Single image mean: {train_ds[0]["data"].shape} (C, W, H)')
print(f'Exaple label {train_ds[0]["label"]}')
print(f'Example data: {train_ds[0]["data"]}')
```

The output could look something like this:

```
Single image min: 87.0
Single image max: 255.0
Single image mean: torch.Size([1, 64, 64]) (C, W, H)
Exaple label 0
Example data: tensor([[[101., 101., 101.,  ..., 101., 101., 101.],
         [101., 101., 101.,  ..., 101., 101., 101.],
         [101., 101., 101.,  ..., 101., 101., 101.],
         ...,
         [102., 101.,  99.,  ..., 111., 103.,  98.],
         [102., 101., 100.,  ...,  99.,  98.,  98.],
         [ 99., 100., 102.,  ..., 101., 103., 105.]]])
```

13.2.3 Setting Up our Dataloading and Transformations

In this section we will define our transformations and plug our dataset into the dataloader of `rising`.

First we setup our transformation. In general these can be split into two parts: transformations which are applied as preprocessing and transformations which are applied as augmentations. All transformations are applied in a batched fashion to the dataset to fully utilize vectorization to speed up augmentation. In case your dataset needs additional preprocessing on a per sample basis you can also add those to the dataloder with `sample_transforms`. Check out or 3D Segmentation Tutorial for more infroamtion about that.

```
[ ]: import rising.transforms as rtr
from rising.random import UniformParameter

transforms_prep = []
transforms_augment = []

# preprocessing transforms
```

(continues on next page)

(continued from previous page)

```
# transforms_prep.append(rtr.NormZeroMeanUnitStd())
transforms_prep.append(rtr.NormMinMax()) # visualization looks nicer :)

# augmentation transforms
transforms_augment.append(rtr.GaussianNoise(0., 0.01))
transforms_augment.append(rtr.GaussianSmoothing(
    in_channels=1, kernel_size=3, std=0.5, padding=1))
transforms_augment.append(rtr.Rot90((0, 1)))
transforms_augment.append(rtr.Mirror(dims=(0, 1)))
transforms_augment.append(rtr.BaseAffine(
    scale=UniformParameter(0.8, 1.2),
    rotation=UniformParameter(-30, 30), degree=True,
    # translation in base affine is normalized to image size
    # Translation transform offers to option to swith to pixels
    translation=UniformParameter(-0.02, 0.02),
))
```

In contrast to native PyTorch we add our transformations to the dataloder of rising. There are three main types of transformations which can be added: * `sample_transforms`: these transforms are applied per sample. In case the transformation assumes a batch of data `pseudo_batch_dim` can be activated to automatically add a batch dim to single samples. * `batch_transforms`: these transforms are executed per batch inside the multiprocessig context of the CPU (like `sample_transforms`). * `gpu_transforms`: these transforms are executed on the GPU. In case you have multiple GPUs make sure to set the correct `device`, otherwise rising could use the wrong GPU.

[ ]:
```
from rising.loading import DataLoader

tr_transform = rtr.Compose(transforms_prep + transforms_augment)
dataloader_tr = DataLoader(train_ds, batch_size=32, shuffle=True,
                           gpu_transforms=tr_transform)

val_transform = rtr.Compose(transforms_prep)
dataloader_val = DataLoader(val_ds, batch_size=32,
                            gpu_transforms=val_transform)

test_transform = rtr.Compose(transforms_prep)
dataloader_ts = DataLoader(test_ds, batch_size=32,
                           gpu_transforms=test_transform)
```

Looking at some example outputs

In this short section we will visualize some of the batches to look at the influence of the augmentations.

[ ]:
```
# helper function to visualize batches of images
import torch
import torchvision
import matplotlib.pyplot as plt

def show_batch(batch: torch.Tensor, norm: bool = True):
  """
  Visualize a single batch of images

  Args:
    batch: batch of data
    norm: normalized to range 0,1 for visualization purposes
  """
```

(continues on next page)

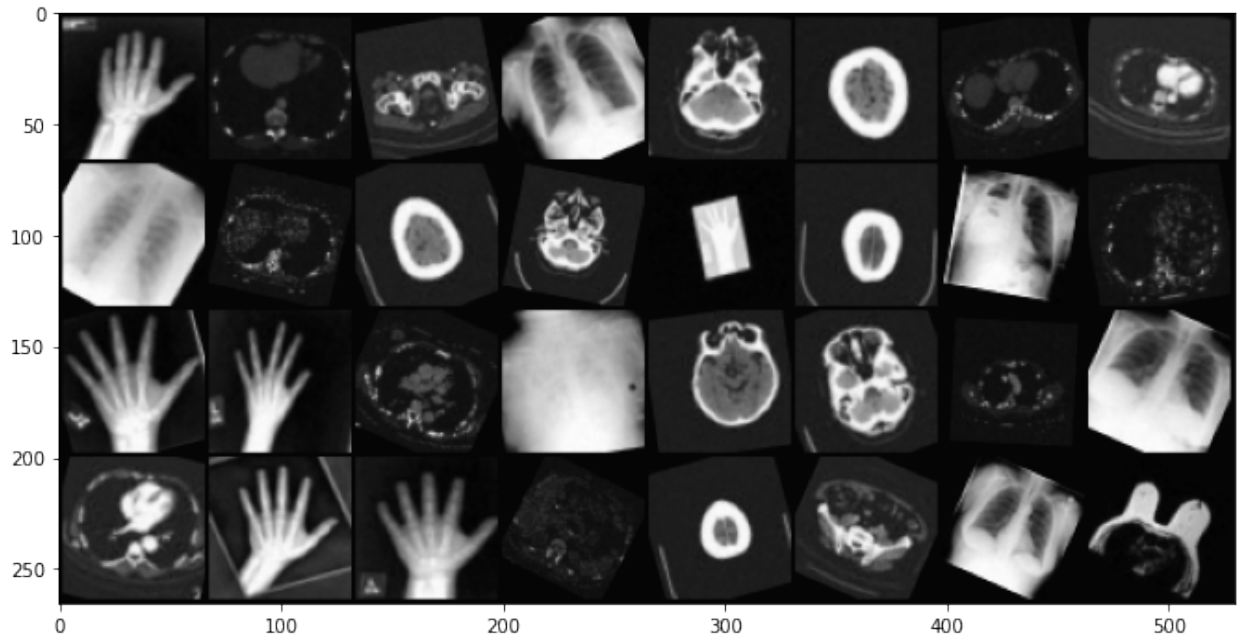
(continued from previous page)

```
  grid = torchvision.utils.make_grid(batch.cpu(), nrow=8)

  grid -= grid.min()
  m = grid.max()
  if m > 1e-6:
    grid = grid / m

  plt.figure(figsize=(10,5))
  plt.imshow(grid[0], cmap='gray', vmin=0, vmax=1)
  plt.tight_layout()
  plt.show()
```

[ ]:
```
# make dataset iterable
_iter = iter(dataloader_tr)
```

[ ]:
```
# visualize batch of images
batch = next(_iter)
print({f'{key}_shape: {tuple(batch[key].shape)}' for key, item in batch.items()})
print(f'Batch labels: \n{batch["label"]}')
print(f'Batch mean {batch["data"].mean()}')
print(f'Batch min {batch["data"].min()}')
print(f'Batch max {batch["data"].max()}')

show_batch(batch["data"], norm=True)
```

The output of the visualization could look something like this:



The exact images will vary because the batch was selected from the training dataloader which shuffles the data.

13.2.4 Defining our Lightning Module

We will use pytorch-lightning as our trainer framework to save some time and to standardize our pipeline.

In lightning the training models are derived from `pytorch_lightning.LightningModule` which enforces a specific structure of the code to increase reproducibility and stardization across the community. For simplicity we will simply load a torchvision model and overwrite the basic `*_step` functions of lightning. If you want more information how to build pipelines with pytorch lightning, please check out their documentation.

[ ]:
```
import torch.nn as nn
import torchvision.models as models

if 'CI' in os.environ:
    # use a very small model for CI
    class SuperSmallModel(nn.Module):
        def __init__(self):
            super().__init__()
            self.conv1 = nn.Conv2d(1, 16, 3, 1, 1)
            self.conv2 = nn.Conv2d(16, 32, 3, 1, 1)
            self.pool1 = nn.AdaptiveAvgPool2d((1, 1))
            self.fc = nn.Linear(32, num_class)

        def forward(self, x):
            x = self.conv1(x)
            x = self.conv2(x)
            x = torch.flatten(self.pool1(x), 1)
            return self.fc(x)
    resnet = SuperSmallModel()
else:
    # resnet18 for normal runs
    resnet = models.resnet18(pretrained=False)
    # change first layer
    resnet.conv1 = torch.nn.Conv2d(
        1, 64, kernel_size=7, stride=2, padding=3, bias=False)
    # change last layer
    fc_in = resnet.fc.in_features
    resnet.fc = nn.Linear(fc_in, num_class)
```

[ ]:
```
import torch.nn.functional as F
import pytorch_lightning as pl

from sklearn.metrics import classification_report
from typing import Dict, Optional


class SimpleClassifier(pl.LightningModule):
  def __init__(self, hparams: Optional[dict] = None):
    """
    Hyperparameters for our model

    Args:
      hparams: hyperparameters for model
        `lr`: learning rate for optimizer
    """
    super().__init__()
    if hparams is None:
        hparams = {}
    self.hparams = hparams
```

(continues on next page)

(continued from previous page)

```
    self.model = resnet

  def forward(self, x: torch.Tensor) -> torch.Tensor:
    """
    Forward input batch of data through model

    Args:
      x: input batch of data [N, C, H, W]
        N batch size (here 32); C number of channels (here 1);
        H,W spatial dimensions of images (here 64x64)

    Returns:
      torch.Tensor: classification logits [N, num_classes]
    """
    return self.model(x)

  def training_step(self, batch: Dict[str, torch.Tensor], batch_idx: int) -> Dict:
    """
    Forward batch and compute loss for a single step (used for training)

    Args:
      batch: batch to process
        `data`: input data
        `label`: expected labels
      batch_idx: index of batch
    """
    x, y = batch["data"], batch["label"]
    y_hat = self(x)
    loss = F.cross_entropy(y_hat, y)
    tensorboard_logs = {'train_loss': loss}
    return {'loss': loss, 'log': tensorboard_logs}

  def validation_step(self, batch: Dict[str, torch.Tensor], batch_idx: int) -> Dict:
    """
    Forward batch and compute loss for a single step (used for validation)

    Args:
      batch: batch to process
        `data`: input data
        `label`: expected labels
      batch_idx: index of batch
    """
    x, y = batch["data"], batch["label"]
    y_hat = self(x)
    val_loss = F.cross_entropy(y_hat, y)
    return {'val_loss': val_loss}

  def validation_epoch_end(self, outputs):
    """
    Compute average validation loss during epoch
    """
    avg_loss = torch.stack([x['val_loss'] for x in outputs]).mean()
    tensorboard_logs = {'val_loss': avg_loss}
    return {'val_loss': avg_loss, 'log': tensorboard_logs}

  def test_step(self, batch: Dict[str, torch.Tensor], batch_idx: int) -> Dict:
    """
```

(continues on next page)

(continued from previous page)

```
    Forward batch and compute loss for a single step (used for validation)

    Args:
      batch: batch to process
        `data`: input data
        `label`: expected labels
      batch_idx: index of batch
    """
    x, y = batch["data"], batch["label"]
    y_hat = self(x)
    val_loss = F.cross_entropy(y_hat, y)
    return {'test_loss': val_loss,
            "pred_label": y_hat.max(dim=1)[1].detach().cpu(),
            "label": y.detach().cpu()}

  def test_epoch_end(self, outputs):
    """
    Compute average test loss and classification metrics
    """
    avg_loss = torch.stack([x['test_loss'] for x in outputs]).mean()
    tensorboard_logs = {'test_loss': avg_loss}

    all_pred_label = torch.cat([x['pred_label'] for x in outputs])
    all_label = torch.cat([x['label'] for x in outputs])
    print(classification_report(all_label.numpy(),
                                all_pred_label.numpy(),
                                target_names=class_names, digits=4))

    return {'test_loss': avg_loss, 'log': tensorboard_logs}

  def configure_optimizers(self):
    """
    Setup optimizer for training
    """
    return torch.optim.Adam(self.parameters(), lr=self.hparams.get("lr", 1e-5))
```

We can visualize our training progress and hyperparameters in tensorboard to easily compare multiple runs of our classifier.

[ ]:
```
# Start tensorboard.
%reload_ext tensorboard
%tensorboard --logdir lightning_logs/
```

Let's start our training :D

[ ]:
```
from pytorch_lightning import Trainer

model = SimpleClassifier()

if torch.cuda.is_available():
    gpus = [0]
else:
    gpus=None

# most basic trainer, uses good defaults
trainer = Trainer(gpus=gpus, progress_bar_refresh_rate=10, max_epochs=4, weights_
↪summary=None)
```

(continues on next page)

(continued from previous page)

```
trainer.fit(model, train_dataloader=dataloader_tr, val_dataloaders=dataloader_val)
```

After training our model we can test it on our test data.

[ ]:
```
trainer.test(test_dataloaders=dataloader_ts)
```

The results on the test data should look similar to this:

```
              precision    recall  f1-score   support

   AbdomenCT     0.9536    0.9990    0.9758      1008
   BreastMRI     1.0000    1.0000    1.0000       830
         CXR     0.9960    0.9872    0.9916      1013
     ChestCT     1.0000    0.9490    0.9738       961
        Hand     0.9877    0.9887    0.9882       975
      HeadCT     0.9912    1.0000    0.9956      1019

    accuracy                         0.9873      5806
   macro avg     0.9881    0.9873    0.9875      5806
weighted avg     0.9876    0.9873    0.9872      5806
```

[ ]:
```
%reload_ext autoreload
%autoreload 2
%matplotlib inline
```

13.3 Using transformation from external libraries inside `rising`

Note: Some external augmentation libraries are only supported at the beginning of the transformation pipeline. Generally speaking, if you need to resort to an external library for augmentations, consider creating an issue in `rising` and there is a high chance we will add the transformation in the future :)

[ ]:
```
# lets prepare a basic dataset (e.g. one from `torchvision`)
import os
import torchvision
import numpy as np
import torch

def to_array(inp):
    """
    We need a small helper in this example because torchvision datasets output PIL
    images. When using them in combination with `rising`,
    just add `torchvision.transforms.ToTensor()`to the transform of the dataset

    Returns
    -------
    numpy.ndarray
        converted data
    """
    from PIL import Image
    if isinstance(inp, Image.Image):
        return np.array(inp, np.float32, copy=False)[None]
    elif isinstance(inp, torch.Tensor):
```

(continues on next page)

(continued from previous page)

```
        return inp.detach().cpu().numpy()
    else:
        return inp

dataset = torchvision.datasets.MNIST(
    os.getcwd(), train=True, download=True, transform=to_array)
```

[ ]:
```
#  plot shape
print(dataset[0][0].shape)
# visualize a single image
import matplotlib.pyplot as plt

plt.imshow(dataset[0][0][0], cmap='gray')
plt.colorbar()
plt.show()
```

[ ]:
```
# helper function to visualize batches of images
import torch

def show_batch(batch: torch.Tensor):
    grid = torchvision.utils.make_grid(batch)
    plt.imshow(grid[0], cmap='gray')
    # plt.colorbar()
    plt.show()
```

13.3.1 Integration of `batchgenerators` transformations into the augmentation pipeline.

Note: when batchgenerator transformations are integrated, gradients can not be propagated through the transformation pipeline.

`batchgenerators` transformations are based on numpy to be framework agnostic. They are also based on dictionaries which are modified through the transformations.

There are two steps which need to be integrated into your pipelin in order to the `batchgenerators` transforms

1. Exchange the `default_collate` function inside the dataloder with `numpy_collate`
2. When switching from `batchgenerators` transformations to `rising` transformations, insdert `ToTensor` transformation

[ ]:
```
# setup transforms
from rising.transforms import *
from rising.random import UniformParameter
from batchgenerators.transforms import ZeroMeanUnitVarianceTransform

transforms = []
# convert tuple into dict
transforms.append(SeqToMap("data", "label"))
# batchgenerators transforms
transforms.append(ZeroMeanUnitVarianceTransform())
# convert to tensor
transforms.append(ToTensor())
# rising transforms
transforms.append(Rot90((0, 1)))
```

(continues on next page)

(continued from previous page)

```
transforms.append(Mirror(dims=(0, 1)))
transforms.append(Rotate([UniformParameter(0, 180)], adjust_size=True, degree=True))
transforms.append(Scale([UniformParameter(0.8, 1.2)]))
```

[ ]:
```
from rising.loading import DataLoader, default_transform_call, numpy_collate
from rising.transforms import Compose

composed = Compose(transforms, transform_call=default_transform_call)
dataloader = DataLoader(dataset, batch_size=8, batch_transforms=composed,
                        num_workers=0, collate_fn=numpy_collate)
```

[ ]:
```
_iter = iter(dataloader)
batch = next(_iter)
show_batch(batch["data"])
```

13.3.2 More libraries will be added in the future :)

13.4 Transformations

[ ]:
```
!pip install napari
!pip install SimpleITK
```

[ ]:
```
%reload_ext autoreload
%autoreload 2
%matplotlib inline
%gui qt
import os
if 'TEST_ENV' in os.environ:
    TEST_ENV = os.environ['TEST_ENV'].lower() == "true"
else:
    TEST_ENV = 0
print(f"Running test environment: {bool(TEST_ENV)}")
```

[ ]:
```
from io import BytesIO
from zipfile import ZipFile
from urllib.request import urlopen

resp = urlopen("http://www.fmrib.ox.ac.uk/primers/intro_primer/ExBox3/ExBox3.zip")
zipfile = ZipFile(BytesIO(resp.read()))

img_file = zipfile.extract("ExBox3/T1_brain.nii.gz")
mask_file = zipfile.extract("ExBox3/T1_brain_seg.nii.gz")
```

[ ]:
```
import SimpleITK as sitk
import numpy as np

# load image and mask
img_file = "./ExBox3/T1_brain.nii.gz"
```

(continues on next page)

(continued from previous page)

```
mask_file = "./ExBox3/T1_brain_seg.nii.gz"
img = sitk.GetArrayFromImage(sitk.ReadImage(img_file))
img = img.astype(np.float32)
mask = mask = sitk.GetArrayFromImage(sitk.ReadImage(mask_file))
mask = mask.astype(np.float32)

assert mask.shape == img.shape
print(f"Image shape {img.shape}")
print(f"Image shape {mask.shape}")
```

[ ]:
```
if TEST_ENV:
    def view_batch(batch):
        pass
else:
    %gui qt
    import napari
    def view_batch(batch):
            viewer = napari.view_image(batch["data"].cpu().numpy(), name="data")
            viewer.add_image(batch["mask"].cpu().numpy(), name="mask", opacity=0.2)
```

[ ]:
```
import torch
from rising.transforms import *

batch = {
    "data": torch.from_numpy(img).float()[None, None],
    "mask": torch.from_numpy(mask).long()[None, None],
}

def apply_transform(trafo, batch):
    transformed = trafo(**batch)
    print(f"Transformed data shape: {transformed['data'].shape}")
    print(f"Transformed mask shape: {transformed['mask'].shape}")
    print(f"Transformed data min: {transformed['data'].min()}")
    print(f"Transformed data max: {transformed['data'].max()}")
    print(f"Transformed data mean: {transformed['data'].mean()}")
    return transformed
```

[ ]:
```
print(f"Transformed data shape: {batch['data'].shape}")
print(f"Transformed mask shape: {batch['mask'].shape}")
print(f"Transformed data min: {batch['data'].min()}")
print(f"Transformed data max: {batch['data'].max()}")
print(f"Transformed data mean: {batch['data'].mean()}")
```

[ ]:
```
trafo = Scale(1.5, adjust_size=False)
transformed = apply_transform(trafo, batch)
view_batch(transformed)
```

[ ]:
```
trafo = Rotate([0, 0, 45], degree=True, adjust_size=False)
transformed = apply_transform(trafo, batch)
view_batch(transformed)
```

[ ]:
```
trafo = Translate([0.1, 0, 0], adjust_size=False)
transformed = apply_transform(trafo, batch)
view_batch(transformed)
```

[ ]:

CHAPTER

FOURTEEN

CONTRIBUTING TO `rising`

If you are interested in contributing to `rising`, you can either implement a new feature or fix a bug.

For both types of contributions, the process is roughly the same:

1. Open an issue in this repo and discuss the issue with us! Maybe we can give you some hints towards implementation/fixing.
2. If you're not part of the core development team, we need you to create your own fork of this repo, implement it there and create a PR to this repo afterwards.
3. Create a new branch (in your fork if necessary) for the implementation of your issue. Make sure to include basic unittests.
4. After finishing the implementation, send a pull request to the correct branch of this repo (probably master branch).
5. Afterwards, have a look at your pull request since we might suggest some changes.

If you are not familiar with creating a pull request, here are some guides:

- http://stackoverflow.com/questions/14680711/how-to-do-a-github-pull-request
- https://help.github.com/articles/creating-a-pull-request/

14.1 Development Install

To develop `rising` on your machine, here are some tips:

1. Uninstall all existing installs of `rising`:

```
pip uninstall rising
pip uninstall rising # run this command twice
```

1. Clone a copy of `rising` from source:

```
git clone https://github.com/PhoenixDL/rising.git
cd rising
```

1. Install `rising` in `build develop` mode:

Install it via

```
python setup.py build develop
```

or

```
pip install -e .
```

This mode will symlink the python files from the current local source tree into the python install.

Hence, if you modify a python file, you do not need to reinstall `rising` again and again

In case you want to reinstall, make sure that you uninstall `rising` first by running `pip uninstall rising` and `python setup.py clean`. Then you can install in `build develop` mode again.

14.2 Code Style

- To improve readability and maintainability, PEP8 Style should always be followed
  - maximum code line length is 120
  - maximum doc string line length is 80
- All imports inside the package should be absolute
- If you add a feature, you should also add it to the documentation
- Every module must have an `__all__` section
- All functions should be typed
- Keep functions short and give them meaningful names

14.3 Unit testing

Unittests are located under `tests/`. Run the entire test suite with

```
python -m unittest
```

from the `rising` root directory or run individual test files, like `python test/test_dummy.py`, for individual test suites.

14.3.1 Better local unit tests with unittest

Testing is done with a `unittest` suite

You can run your tests with coverage by installing ´coverage´ and executing

```
coverage run -m unittest; coverage report -m;
```

inside the terminal. Pycharm Professional supports `Run with coverage` directly. Furthermore, the coverage is always computed and uploaded when you execute `git push` and can be seen on github.

14.4 Writing documentation

`rising` uses an adapted version of google style for formatting docstrings. Opposing to the original google style we opted to not duplicate the typing from the function signature to the docstrings. Length of line inside docstrings block must be limited to 80 characters to fit into Jupyter documentation popups.

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

Z