TorchIO Documentation

Release 0.19.6

Fernando Pérez-García

Apr 16, 2024

CONTENTS

TorchIO is an open-source Python library for efficient loading, preprocessing, augmentation and patch-based sampling of 3D medical images in deep learning, following the design of PyTorch.

It includes multiple intensity and spatial transforms for data augmentation and preprocessing. These transforms include typical computer vision operations such as random affine transformations and also domain-specific ones such as simulation of intensity artifacts due to MRI magnetic field inhomogeneity (bias) or k-space motion artifacts.

TorchIO is part of the official PyTorch Ecosystem, and was featured at the PyTorch Ecosystem Day 2021 and the PyTorch Developer Day 2021.

Many groups have used TorchIO for their research. The complete list of citations is available on Google Scholar, and the dependents list is available on GitHub.

The code is available on GitHub. If you like TorchIO, please go to the repository and star it!

See *Getting started* for installation instructions and a usage overview.

Contributions are more than welcome. Please check our contributing guide if you would like to contribute.

If you have questions, feel free to ask in the discussions tab:

If you found a bug or have a feature request, please open an issue:

CHAPTER

ONE

CREDITS

If you use this library for your research, please cite our paper:

F. Pérez-García, R. Sparks, and S. Ourselin. TorchIO: a Python library for efficient loading, preprocessing, augmentation and patch-based sampling of medical images in deep learning. Computer Methods and Programs in Biomedicine (June 2021), p. 106236. ISSN: 0169-2607.doi:10.1016/j.cmpb.2021.106236.

BibTeX:

```
@article{perez-garcia_torchio_2021,
   title = {TorchIO: a Python library for efficient loading, preprocessing, augmentation␣
↪and patch-based sampling of medical images in deep learning},
   journal = {Computer Methods and Programs in Biomedicine},
   pages = {106236},
   year = {2021},
   issn = {0169-2607},
   doi = {https://doi.org/10.1016/j.cmpb.2021.106236},
   url = {https://www.sciencedirect.com/science/article/pii/S0169260721003102},
   author = {P{\'e}rez-Garc{\'i}a, Fernando and Sparks, Rachel and Ourselin, S{\'e}
↪bastien},
}
```

This project is supported by the following institutions:

- Engineering and Physical Sciences Research Council (EPSRC) & UK Research and Innovation (UKRI)
- EPSRC Centre for Doctoral Training in Intelligent, Integrated Imaging In Healthcare (i4health) (University College London)
- Wellcome / EPSRC Centre for Interventional and Surgical Sciences (WEISS) (University College London)
- School of Biomedical Engineering & Imaging Sciences (BMEIS) (King's College London)

This library has been greatly inspired by NiftyNet, which is no longer maintained.

1.1 Getting started

1.1.1 Installation

The Python package is hosted on the Python Package Index (PyPI).

To install the latest PyTorch version before installing TorchIO, it is recommended to use light-the-torch:

```
$ pip install light-the-torch && ltt install torch
```

The latest published version of TorchIO can be installed using Pip Installs Packages (`pip`):

```
$ pip install torchio
```

To upgrade to the latest published version, use:

```
$ pip install --upgrade torchio
```

If you would like to install Matplotlib to use the plotting features, use:

```
$ pip install torchio[plot]
```

If you are on Windows and have trouble installing TorchIO, try installing PyTorch with conda before pip-installing TorchIO.

1.1.2 Hello, World!

This example shows the basic usage of TorchIO, where an instance of `SubjectsDataset` is passed to a PyTorch `DataLoader` to generate training batches of 3D images that are loaded, preprocessed and augmented on the fly, in parallel:

```
import torch
import torchio as tio
from torch.utils.data import DataLoader

# Each instance of tio.Subject is passed arbitrary keyword arguments.
# Typically, these arguments will be instances of tio.Image
subject_a = tio.Subject(
    t1=tio.ScalarImage('subject_a.nii.gz'),
    label=tio.LabelMap('subject_a.nii'),
    diagnosis='positive',
)

# Image files can be in any format supported by SimpleITK or NiBabel, including DICOM
subject_b = tio.Subject(
    t1=tio.ScalarImage('subject_b_dicom_folder'),
    label=tio.LabelMap('subject_b_seg.nrrd'),
    diagnosis='negative',
)

# Images may also be created using PyTorch tensors or NumPy arrays
tensor_4d = torch.rand(4, 100, 100, 100)
subject_c = tio.Subject(
    t1=tio.ScalarImage(tensor=tensor_4d),
    label=tio.LabelMap(tensor=(tensor_4d > 0.5)),
    diagnosis='negative',
)

subjects_list = [subject_a, subject_b, subject_c]

# Let's use one preprocessing transform and one augmentation transform
```

(continues on next page)

(continued from previous page)

```
# This transform will be applied only to scalar images:
rescale = tio.RescaleIntensity(out_min_max=(0, 1))

# As RandomAffine is faster then RandomElasticDeformation, we choose to
# apply RandomAffine 80% of the times and RandomElasticDeformation the rest
# Also, there is a 25% chance that none of them will be applied
spatial = tio.OneOf({
        tio.RandomAffine(): 0.8,
        tio.RandomElasticDeformation(): 0.2,
    },
    p=0.75,
)

# Transforms can be composed as in torchvision.transforms
transforms = [rescale, spatial]
transform = tio.Compose(transforms)

# SubjectsDataset is a subclass of torch.data.utils.Dataset
subjects_dataset = tio.SubjectsDataset(subjects_list, transform=transform)

# Images are processed in parallel thanks to a PyTorch DataLoader
training_loader = DataLoader(subjects_dataset, batch_size=4, num_workers=4)

# Training epoch
for subjects_batch in training_loader:
    inputs = subjects_batch['t1'][tio.DATA]
    target = subjects_batch['label'][tio.DATA]
```

1.1.3 Tutorials

The best way to quickly understand and try the library is the Jupyter Notebooks hosted on Google Colab.

They include multiple examples and visualization of most of the classes, including training of a 3D U-Net for brain segmentation on T_1-weighted MRI with full volumes and with subvolumes (aka patches or windows).

1.2 Data structures

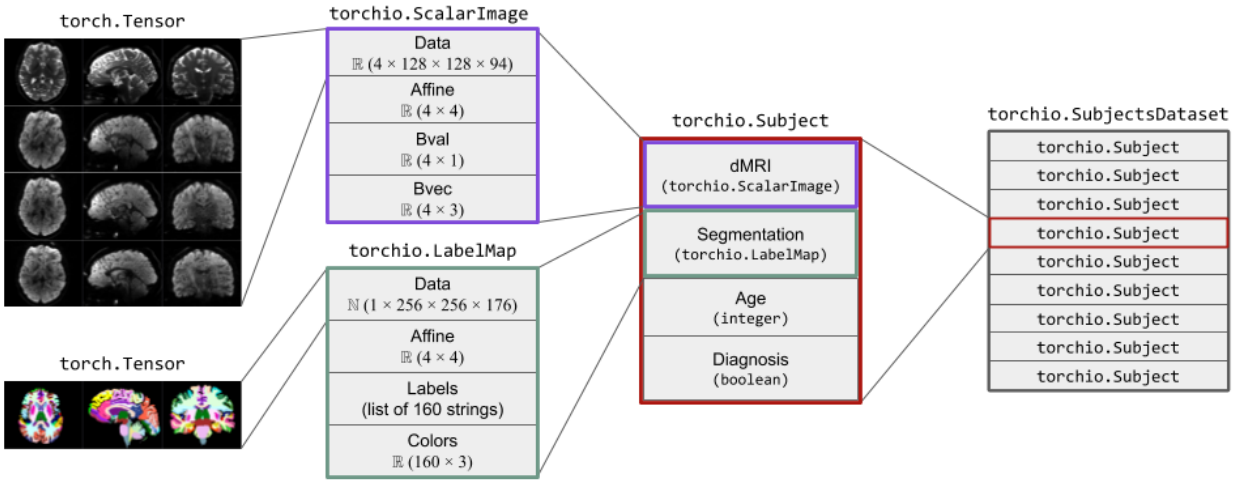
1.2.1 Image

The *Image* class, representing one medical image, stores a 4D tensor, whose voxels encode, e.g., signal intensity or segmentation labels, and the corresponding affine transform, typically a rigid (Euclidean) transform, to convert voxel indices to world coordinates in mm. Arbitrary fields such as acquisition parameters may also be stored.

Subclasses are used to indicate specific types of images, such as *ScalarImage* and *LabelMap*, which are used to store, e.g., CT scans and segmentations, respectively.

An instance of *Image* can be created using a filepath, a PyTorch tensor, or a NumPy array. This class uses lazy loading, i.e., the data is not loaded from disk at instantiation time. Instead, the data is only loaded when needed for an operation (e.g., if a transform is applied to the image).

The figure below shows two instances of *Image*. The instance of *ScalarImage* contains a 4D tensor representing a diffusion MRI, which contains four 3D volumes (one per gradient direction), and the associated affine matrix. Additionally, it stores the strength and direction for each of the four gradients. The instance of *LabelMap* contains a brain parcellation of the same subject, the associated affine matrix, and the name and color of each brain structure.



class torchio.**ScalarImage**(**args*, ***kwargs*)

Bases: *Image*

Image whose pixel values represent scalars.

Example

```
>>> import torch
>>> import torchio as tio
>>> # Loading from a file
>>> t1_image = tio.ScalarImage('t1.nii.gz')
>>> dmri = tio.ScalarImage(tensor=torch.rand(32, 128, 128, 88))
>>> image = tio.ScalarImage('safe_image.nrrd', check_nans=False)
>>> data, affine = image.data, image.affine
>>> affine.shape
(4, 4)
>>> image.data is image[tio.DATA]
True
>>> image.data is image.tensor
True
>>> type(image.data)
torch.Tensor
```

See *Image* for more information.

class torchio.**LabelMap**(**args*, ***kwargs*)

Bases: *Image*

Image whose pixel values represent categorical labels.

Example

```
>>> import torch
>>> import torchio as tio
>>> labels = tio.LabelMap(tensor=torch.rand(1, 128, 128, 68) > 0.5)
>>> labels = tio.LabelMap('t1_seg.nii.gz')  # loading from a file
>>> tpm = tio.LabelMap(                     # loading from files
...     'gray_matter.nii.gz',
...     'white_matter.nii.gz',
...     'csf.nii.gz',
... )
```

Intensity transforms are not applied to these images.

Nearest neighbor interpolation is always used to resample label maps, independently of the specified interpolation type in the transform instantiation.

See *Image* for more information.

class torchio.**Image**(*path: str | ~pathlib.Path | ~typing.Sequence[str | ~pathlib.Path] | None = None, type: str | None = None, tensor: ~torch.Tensor | ~numpy.ndarray | None = None, affine: ~torch.Tensor | ~numpy.ndarray | None = None, check_nans: bool = False, reader: ~typing.Callable = <function read_image>, **kwargs: ~typing.Dict[str, ~typing.Any]*)

Bases: dict

TorchIO image.

For information about medical image orientation, check out NiBabel docs, the 3D Slicer wiki, Graham Wideman's website, FSL docs or SimpleITK docs.

Parameters

- **path** – Path to a file or sequence of paths to files that can be read by `SimpleITK` or `nibabel`, or to a directory containing DICOM files. If *tensor* is given, the data in `path` will not be read. If a sequence of paths is given, data will be concatenated on the channel dimension so spatial dimensions must match.
- **type** – Type of image, such as `torchio.INTENSITY` or `torchio.LABEL`. This will be used by the transforms to decide whether to apply an operation, or which interpolation to use when resampling. For example, preprocessing and augmentation intensity transforms will only be applied to images with type `torchio.INTENSITY`. Spatial transforms will be applied to all types, and nearest neighbor interpolation is always used to resample images with type `torchio.LABEL`. The type `torchio.SAMPLING_MAP` may be used with instances of *WeightedSampler*.
- **tensor** – If `path` is not given, *tensor* must be a 4D `torch.Tensor` or NumPy array with dimensions (C, W, H, D).
- **affine** – 4×4 matrix to convert voxel coordinates to world coordinates. If `None`, an identity matrix will be used. See the NiBabel docs on coordinates for more information.
- **check_nans** – If `True`, issues a warning if NaNs are found in the image. If `False`, images will not be checked for the presence of NaNs.
- **reader** – Callable object that takes a path and returns a 4D tensor and a 2D, 4×4 affine matrix. This can be used if your data is saved in a custom format, such as `.npy` (see example below). If the affine matrix is `None`, an identity matrix will be used.
- ****kwargs** – Items that will be added to the image dictionary, e.g. acquisition parameters.

TorchIO images are lazy loaders, i.e. the data is only loaded from disk when needed.

Example

```
>>> import torchio as tio
>>> import numpy as np
>>> image = tio.ScalarImage('t1.nii.gz')  # subclass of Image
>>> image  # not loaded yet
ScalarImage(path: t1.nii.gz; type: intensity)
>>> times_two = 2 * image.data  # data is loaded and cached here
>>> image
ScalarImage(shape: (1, 256, 256, 176); spacing: (1.00, 1.00, 1.00); orientation:␣
↪PIR+; memory: 44.0 MiB; type: intensity)
>>> image.save('doubled_image.nii.gz')
>>> def numpy_reader(path):
...     data = np.load(path).as_type(np.float32)
...     affine = np.eye(4)
...     return data, affine
>>> image = tio.ScalarImage('t1.npy', reader=numpy_reader)
```

property affine: ndarray

Affine matrix to transform voxel indices into world coordinates.

as_pil(*transpose=True*)

Get the image as an instance of `PIL.Image`.

Note: Values will be clamped to 0-255 and cast to uint8.

Note: To use this method, Pillow needs to be installed: `pip install Pillow`.

as_sitk(***kwargs*) → Image

Get the image as an instance of `sitk.Image`.

axis_name_to_index(*axis: str*) → int

Convert an axis name to an axis index.

Parameters

axis – Possible inputs are `'Left'`, `'Right'`, `'Anterior'`, `'Posterior'`, `'Inferior'`, `'Superior'`. Lower-case versions and first letters are also valid, as only the first letter will be used.

Note: If you are working with animals, you should probably use `'Superior'`, `'Inferior'`, `'Anterior'` and `'Posterior'` for `'Dorsal'`, `'Ventral'`, `'Rostral'` and `'Caudal'`, respectively.

Note: If your images are 2D, you can use `'Top'`, `'Bottom'`, `'Left'` and `'Right'`.

property bounds: ndarray

Position of centers of voxels in smallest and largest indices.

property data: Tensor

Tensor data (same as *Image.tensor*).

static flip_axis(*axis: str*) → str

Return the opposite axis label. For example, `'L'` -> `'R'`.

Parameters

axis – Axis label, such as `'L'` or `'left'`.

classmethod from_sitk(*sitk_image*)

Instantiate a new TorchIO image from a `sitk.Image`.

Example

```
>>> import torchio as tio
>>> import SimpleITK as sitk
>>> sitk_image = sitk.Image(20, 30, 40, sitk.sitkUInt16)
>>> tio.LabelMap.from_sitk(sitk_image)
LabelMap(shape: (1, 20, 30, 40); spacing: (1.00, 1.00, 1.00); orientation: LPS+;
↪ memory: 93.8 KiB; dtype: torch.IntTensor)
>>> sitk_image = sitk.Image((224, 224), sitk.sitkVectorFloat32, 3)
>>> tio.ScalarImage.from_sitk(sitk_image)
ScalarImage(shape: (3, 224, 224, 1); spacing: (1.00, 1.00, 1.00); orientation:␣
↪LPS+; memory: 588.0 KiB; dtype: torch.FloatTensor)
```

get_bounds() → Tuple[Tuple[float, float], Tuple[float, float], Tuple[float, float]]

Get minimum and maximum world coordinates occupied by the image.

get_center(*lps: bool = False*) → Tuple[float, float, float]

Get image center in RAS+ or LPS+ coordinates.

Parameters

lps – If `True`, the coordinates will be in LPS+ orientation, i.e. the first dimension grows towards the left, etc. Otherwise, the coordinates will be in RAS+ orientation.

property height: int

Image height, if 2D.

property itemsize

Element size of the data type.

load() → None

Load the image from disk.

Returns

Tuple containing a 4D tensor of size (C, W, H, D) and a 2D 4×4 affine matrix to convert voxel indices to world coordinates.

property memory: float

Number of Bytes that the tensor takes in the RAM.

property num_channels: int

Get the number of channels in the associated 4D tensor.

numpy() → ndarray

Get a NumPy array containing the image data.

property orientation: `Tuple[str, str, str]`

Orientation codes.

property origin: `Tuple[float, float, float]`

Center of first voxel in array, in mm.

plot(***kwargs*) → None

Plot image.

save(*path: str | Path*, *squeeze: bool | None = None*) → None

Save image to disk.

Parameters

- **path** – String or instance of `pathlib.Path`.
- **squeeze** – Whether to remove singleton dimensions before saving. If `None`, the array will be squeezed if the output format is JP(E)G, PNG, BMP or TIF(F).

set_data(*tensor: Tensor | ndarray*)

Store a 4D tensor in the `data` key and attribute.

Parameters

tensor – 4D tensor with dimensions (C, W, H, D).

property shape: `Tuple[int, int, int, int]`

Tensor shape as (C, W, H, D).

show(*viewer_path: str | Path | None = None*) → None

Open the image using external software.

Parameters

viewer_path – Path to the application used to view the image. If `None`, the value of the environment variable `SITK_SHOW_COMMAND` will be used. If this variable is also not set, TorchIO will try to guess the location of ITK-SNAP and 3D Slicer.

Raises

`RuntimeError` – If the viewer is not found.

property spacing: `Tuple[float, float, float]`

Voxel spacing in mm.

property spatial_shape: `Tuple[int, int, int]`

Tensor spatial shape as (W, H, D).

property tensor: `Tensor`

Tensor data (same as `Image.data`).

to_gif(*axis: int*, *duration: float*, *output_path: str | Path*, *loop: int = 0*, *rescale: bool = True*, *optimize: bool = True*, *reverse: bool = False*) → None

Save an animated GIF of the image.

Parameters

- **axis** – Spatial axis (0, 1 or 2).
- **duration** – Duration of the full animation in seconds.
- **output_path** – Path to the output GIF file.
- **loop** – Number of times the GIF should loop. `0` means that it will loop forever.

- **rescale** – Use *RescaleIntensity* to rescale the intensity values to $[0, 255]$.
- **optimize** – If `True`, attempt to compress the palette by eliminating unused colors. This is only useful if the palette can be compressed to the next smaller power of 2 elements.
- **reverse** – Reverse the temporal order of frames.

unload() → None

Unload the image from memory.

Raises

RuntimeError – If the images has not been loaded yet or if no path is available.

property width: int

Image width, if 2D.

1.2.2 Subject

The *Subject* is a data structure used to store images associated with a subject and any other metadata necessary for processing.

All transforms applied to a *Subject* are saved in its `history` attribute (see *Reproducibility*).

class torchio.**Subject**(**args*, ***kwargs: Dict[str, Any]*)

Bases: `dict`

Class to store information about the images corresponding to a subject.

Parameters

- ***args** – If provided, a dictionary of items.
- ****kwargs** – Items that will be added to the subject sample.

Example

```
>>> import torchio as tio
>>> # One way:
>>> subject = tio.Subject(
...     one_image=tio.ScalarImage('path_to_image.nii.gz'),
...     a_segmentation=tio.LabelMap('path_to_seg.nii.gz'),
...     age=45,
...     name='John Doe',
...     hospital='Hospital Juan Negrín',
... )
>>> # If you want to create the mapping before, or have spaces in the keys:
>>> subject_dict = {
...     'one image': tio.ScalarImage('path_to_image.nii.gz'),
...     'a segmentation': tio.LabelMap('path_to_seg.nii.gz'),
...     'age': 45,
...     'name': 'John Doe',
...     'hospital': 'Hospital Juan Negrín',
... }
>>> subject = tio.Subject(subject_dict)
```

add_image(*image:* Image, *image_name: str*) → None

Add an image to the subject instance.

apply_inverse_transform(***kwargs*) → *Subject*

Apply the inverse of all applied transforms, in reverse order.

Parameters

****kwargs** – Keyword arguments passed on to `get_inverse_transform()`.

check_consistent_attribute(*attribute: str*, *relative_tolerance: float = 1e-06*, *absolute_tolerance: float = 1e-06*, *message: str | None = None*) → None

Check for consistency of an attribute across all images.

Parameters

- **attribute** – Name of the image attribute to check
- **relative_tolerance** – Relative tolerance for `numpy.allclose()`
- **absolute_tolerance** – Absolute tolerance for `numpy.allclose()`

Example

```
>>> import numpy as np
>>> import torch
>>> import torchio as tio
>>> scalars = torch.randn(1, 512, 512, 100)
>>> mask = torch.tensor(scalars > 0).type(torch.int16)
>>> af1 = np.eye([0.8, 0.8, 2.50000000000001, 1])
>>> af2 = np.eye([0.8, 0.8, 2.49999999999999, 1])  # small difference here (e.g.
↪ due to different reader)
>>> subject = tio.Subject(
...   image = tio.ScalarImage(tensor=scalars, affine=af1),
...   mask = tio.LabelMap(tensor=mask, affine=af2)
... )
>>> subject.check_consistent_attribute('spacing')  # no error as tolerances are␣
↪> 0
```

Note: To check that all values for a specific attribute are close between all images in the subject, `numpy.allclose()` is used. This function returns `True` if $|a_i - b_i| \leq t_{abs} + t_{rel} * |b_i|$, where a_i and b_i are the i-th element of the same attribute of two images being compared, t_{abs} is the `absolute_tolerance` and t_{rel} is the `relative_tolerance`.

get_inverse_transform(*warn: bool = True*, *ignore_intensity: bool = False*, *image_interpolation: str | None = None*) → *Compose*

Get a reversed list of the inverses of the applied transforms.

Parameters

- **warn** – Issue a warning if some transforms are not invertible.
- **ignore_intensity** – If `True`, all instances of `IntensityTransform` will be ignored.
- **image_interpolation** – Modify interpolation for scalar images inside transforms that perform resampling.

load() → None

Load images in subject on RAM.

plot(***kwargs*) → None

Plot images using matplotlib.

Parameters

****kwargs** – Keyword arguments that will be passed on to *plot()*.

remove_image(*image_name: str*) → None

Remove an image from the subject instance.

property shape

Return shape of first image in subject.

Consistency of shapes across images in the subject is checked first.

Example

```
>>> import torchio as tio
>>> colin = tio.datasets.Colin27()
>>> colin.shape
(1, 181, 217, 181)
```

property spacing

Return spacing of first image in subject.

Consistency of spacings across images in the subject is checked first.

Example

```
>>> import torchio as tio
>>> colin = tio.datasets.Slicer()
>>> colin.spacing
(1.0, 1.0, 1.2999954223632812)
```

property spatial_shape

Return spatial shape of first image in subject.

Consistency of spatial shapes across images in the subject is checked first.

Example

```
>>> import torchio as tio
>>> colin = tio.datasets.Colin27()
>>> colin.spatial_shape
(181, 217, 181)
```

unload() → None

Unload images in subject.

1.2.3 Dataset

SubjectsDataset

class torchio.data.**SubjectsDataset**(*subjects: Sequence[Subject]*, *transform: Callable | None = None*, *load_getitem: bool = True*)

Bases: Dataset

Base TorchIO dataset.

Reader of 3D medical images that directly inherits from the PyTorch Dataset. It can be used with a PyTorch DataLoader for efficient loading and augmentation. It receives a list of instances of *Subject* and an optional transform applied to the volumes after loading.

Parameters

- **subjects** – List of instances of *Subject*.
- **transform** – An instance of *Transform* that will be applied to each subject.
- **load_getitem** – Load all subject images before returning it in `__getitem__()`. Set it to `False` if some of the images will not be needed during training.

Example

```
>>> import torchio as tio
>>> subject_a = tio.Subject(
...     t1=tio.ScalarImage('t1.nrrd',),
...     t2=tio.ScalarImage('t2.mha',),
...     label=tio.LabelMap('t1_seg.nii.gz'),
...     age=31,
...     name='Fernando Perez',
... )
>>> subject_b = tio.Subject(
...     t1=tio.ScalarImage('colin27_t1_tal_lin.minc',),
...     t2=tio.ScalarImage('colin27_t2_tal_lin_dicom',),
...     label=tio.LabelMap('colin27_seg1.nii.gz'),
...     age=56,
...     name='Colin Holmes',
... )
>>> subjects_list = [subject_a, subject_b]
>>> transforms = [
...     tio.RescaleIntensity(out_min_max=(0, 1)),
...     tio.RandomAffine(),
... ]
>>> transform = tio.Compose(transforms)
>>> subjects_dataset = tio.SubjectsDataset(subjects_list, transform=transform)
>>> subject = subjects_dataset[0]
```

Tip: To quickly iterate over the subjects without loading the images, use *dry_iter()*.

dry_iter()

Return the internal list of subjects.

This can be used to iterate over the subjects without loading the data and applying any transforms:

```
>>> names = [subject.name for subject in dataset.dry_iter()]
```

classmethod from_batch(*batch: dict*) → *SubjectsDataset*

Instantiate a dataset from a batch generated by a data loader.

Parameters

batch – Dictionary generated by a data loader, containing data that can be converted to instances of *Subject*.

set_transform(*transform: Callable | None*) → None

Set the `transform` attribute.

Parameters

transform – Callable object, typically an subclass of *torchio.transforms.Transform*.

1.3 Patch-based pipelines

The number of pixels in 2D images used in deep learning is rarely larger than one million. For example, the input size of several popular image classification models is 224 × 224 × 3 = 150 528 pixels (588 KiB if 32 bits per pixel are used). In contrast, 3D medical images often contain hundreds of millions of voxels, and downsampling might not be acceptable when small details should be preserved. For example, the size of a high-resolution lung CT-scan used for quantifying chronic obstructive pulmonary disease damage in a research setting, with spacing 0.66 × 0.66 × 0.30 mm, is 512 × 512 × 1069 = 280 231 936 voxels (1.04 GiB if 32 bits per voxel are used).

In computer vision applications, images used for training are grouped in batches whose size is often in the order of hundreds or even thousands of training instances, depending on the available GPU memory. In medical image applications, batches rarely contain more than one or two training instances due to their larger memory footprint compared to natural images. This reduces the utility of techniques such as batch normalization, which rely on batches being large enough to estimate dataset variance appropriately. Moreover, large image size and small batches result in longer training time, hindering the experimental cycle that is necessary for hyperparameter optimization. In cases where GPU memory is limited and the network architecture is large, it is possible that not even the entirety of a single volume can be processed during a single training iteration. To overcome this challenge, it is common in medical imaging to train using subsets of the image, or image *patches*, randomly extracted from the volumes.

Networks can be trained with 2D slices extracted from 3D volumes, aggregating the inference results to generate a 3D volume. This can be seen as a specific case of patch-based training, where the size of the patches along a dimension is one. Other methods extract volumetric patches for training, that are often cubes, if the voxel spacing is isotropic, or cuboids adapted to the anisotropic spacing of the training images.

1.3.1 Training

Patch samplers

Samplers are used to randomly extract patches from volumes. They are called with a sample generated by a *SubjectsDataset* and return a Python generator that yields cropped versions of the sample.

For more information about patch-based training, see this NiftyNet tutorial.

class torchio.data.**UniformSampler**(*patch_size: int | Tuple[int, int, int]*)

Bases: RandomSampler

Randomly extract patches from a volume with uniform probability.

Parameters
patch_size – See *PatchSampler*.

class torchio.data.**WeightedSampler**(*patch_size: int | Tuple[int, int, int]*, *probability_map: str | None*)

Bases: RandomSampler

Randomly extract patches from a volume given a probability map.

The probability of sampling a patch centered on a specific voxel is the value of that voxel in the probability map. The probabilities need not be normalized. For example, voxels can have values 0, 1 and 5. Voxels with value 0 will never be at the center of a patch. Voxels with value 5 will have 5 times more chance of being at the center of a patch that voxels with a value of 1.

Parameters

- **patch_size** – See *PatchSampler*.
- **probability_map** – Name of the image in the input subject that will be used as a sampling probability map.

Raises
RuntimeError – If the probability map is empty.

Example

```
>>> import torchio as tio
>>> subject = tio.Subject(
...     t1=tio.ScalarImage('t1_mri.nii.gz'),
...     sampling_map=tio.Image('sampling.nii.gz', type=tio.SAMPLING_MAP),
... )
>>> patch_size = 64
>>> sampler = tio.data.WeightedSampler(patch_size, 'sampling_map')
>>> for patch in sampler(subject):
...     print(patch[tio.LOCATION])
```

Note: The index of the center of a patch with even size s is arbitrarily set to $s/2$. This is an implementation detail that will typically not make any difference in practice.

Note: Values of the probability map near the border will be set to 0 as the center of the patch cannot be at the border (unless the patch has size 1 or 2 along that axis).

class torchio.data.**LabelSampler**(*patch_size: int | Tuple[int, int, int]*, *label_name: str | None = None*, *label_probabilities: Dict[int, float] | None = None*)

Bases: *WeightedSampler*

Extract random patches with labeled voxels at their center.

This sampler yields patches whose center value is greater than 0 in the `label_name`.

Parameters

- **patch_size** – See *PatchSampler*.
- **label_name** – Name of the label image in the subject that will be used to generate the sampling probability map. If `None`, the first image of type `torchio.LABEL` found in the subject subject will be used.
- **label_probabilities** – Dictionary containing the probability that each class will be sampled. Probabilities do not need to be normalized. For example, a value of `{0: 0, 1: 2, 2: 1, 3: 1}` will create a sampler whose patches centers will have 50% probability of being labeled as 1, 25% of being 2 and 25% of being 3. If `None`, the label map is binarized and the value is set to `{0: 0, 1: 1}`. If the input has multiple channels, a value of `{0: 0, 1: 2, 2: 1, 3: 1}` will create a sampler whose patches centers will have 50% probability of being taken from a non zero value of channel 1, 25% from channel 2 and 25% from channel 3.

Example

```
>>> import torchio as tio
>>> subject = tio.datasets.Colin27()
>>> subject
Colin27(Keys: ('t1', 'head', 'brain'); images: 3)
>>> probabilities = {0: 0.5, 1: 0.5}
>>> sampler = tio.data.LabelSampler(
...     patch_size=64,
...     label_name='brain',
...     label_probabilities=probabilities,
... )
>>> generator = sampler(subject)
>>> for patch in generator:
...     print(patch.shape)
```

If you want a specific number of patches from a volume, e.g. 10:

```
>>> generator = sampler(subject, num_patches=10)
>>> for patch in iterator:
...     print(patch.shape)
```

class torchio.data.**PatchSampler**(*patch_size: int | Tuple[int, int, int]*)

Bases: `object`

Base class for TorchIO samplers.

Parameters

patch_size – Tuple of integers (w, h, d) to generate patches of size $w \times h \times d$. If a single number n is provided, $w = h = d = n$.

> **Warning:** This is an abstract class that should only be instantiated using child classes such as *UniformSampler* and *WeightedSampler*.

class torchio.data.**GridSampler**(*subject: Subject*, *patch_size: int | Tuple[int, int, int]*, *patch_overlap: int | Tuple[int, int, int] = (0, 0, 0)*, *padding_mode: str | float | None = None*)

Bases: *PatchSampler*

Extract patches across a whole volume.

Grid samplers are useful to perform inference using all patches from a volume. It is often used with a *GridAggregator*.

Parameters

- **subject** – Instance of `Subject` from which patches will be extracted.
- **patch_size** – Tuple of integers (w, h, d) to generate patches of size $w \times h \times d$. If a single number n is provided, $w = h = d = n$.
- **patch_overlap** – Tuple of even integers (w_o, h_o, d_o) specifying the overlap between patches for dense inference. If a single number n is provided, $w_o = h_o = d_o = n$.
- **padding_mode** – Same as `padding_mode` in *Pad*. If `None`, the volume will not be padded before sampling and patches at the border will not be cropped by the aggregator. Otherwise, the volume will be padded with $\left(\frac{w_o}{2}, \frac{h_o}{2}, \frac{d_o}{2}\right)$ on each side before sampling. If the sampler is passed to a *GridAggregator*, it will crop the output to its original size.

Example

```
>>> import torchio as tio
>>> colin = tio.datasets.Colin27()
>>> sampler = tio.GridSampler(colin, patch_size=88)
>>> for i, patch in enumerate(sampler()):
...     patch.t1.save(f'patch_{i}.nii.gz')
...
>>> # To figure out the number of patches beforehand:
>>> sampler = tio.GridSampler(colin, patch_size=88)
>>> len(sampler)
8
```

> **Note:** Adapted from NiftyNet. See this NiftyNet tutorial for more information about patch based sampling. Note that `patch_overlap` is twice `border` in NiftyNet tutorial.

Queue

class torchio.data.**Queue**(*subjects_dataset: SubjectsDataset*, *max_length: int*, *samples_per_volume: int*, *sampler: PatchSampler*, *subject_sampler: Sampler | None = None*, *num_workers: int = 0*, *shuffle_subjects: bool = True*, *shuffle_patches: bool = True*, *start_background: bool = True*, *verbose: bool = False*)

Bases: `Dataset`

Queue used for stochastic patch-based training.

A training iteration (i.e., forward and backward pass) performed on a GPU is usually faster than loading, preprocessing, augmenting, and cropping a volume on a CPU. Most preprocessing operations could be performed using a GPU, but these devices are typically reserved for training the CNN so that batch size and input tensor size can be as large as possible. Therefore, it is beneficial to prepare (i.e., load, preprocess and augment) the volumes using multiprocessing CPU techniques in parallel with the forward-backward passes of a training iteration. Once a volume is appropriately prepared, it is computationally beneficial to sample multiple patches from a volume rather than having to prepare the same volume each time a patch needs to be extracted. The sampled patches are then stored in a buffer or *queue* until the next training iteration, at which point they are loaded onto the GPU for inference. For this, TorchIO provides the *Queue* class, which also inherits from the PyTorch `Dataset`. In this queueing system, samplers behave as generators that yield patches from random locations in volumes contained in the *SubjectsDataset*.

The end of a training epoch is defined as the moment after which patches from all subjects have been used for training. At the beginning of each training epoch, the subjects list in the *SubjectsDataset* is shuffled, as is typically done in machine learning pipelines to increase variance of training instances during model optimization. A PyTorch loader queries the datasets copied in each process, which load and process the volumes in parallel on the CPU. A patches list is filled with patches extracted by the sampler, and the queue is shuffled once it has reached a specified maximum length so that batches are composed of patches from different subjects. The internal data loader continues querying the *SubjectsDataset* using multiprocessing. The patches list, when emptied, is refilled with new patches. A second data loader, external to the queue, may be used to collate batches of patches stored in the queue, which are passed to the neural network.

Parameters

- **subjects_dataset** – Instance of *SubjectsDataset*.
- **max_length** – Maximum number of patches that can be stored in the queue. Using a large number means that the queue needs to be filled less often, but more CPU memory is needed to store the patches.
- **samples_per_volume** – Default number of patches to extract from each volume. If a subject contains an attribute `num_samples`, it will be used instead of `samples_per_volume`. A small number of patches ensures a large variability in the queue, but training will be slower.
- **sampler** – A subclass of `PatchSampler` used to extract patches from the volumes.
- **subject_sampler** – Sampler to get subjects from the dataset. It should be an instance of `DistributedSampler` when running distributed training.
- **num_workers** – Number of subprocesses to use for data loading (as in `torch.utils.data.DataLoader`). `0` means that the data will be loaded in the main process.
- **shuffle_subjects** – If `True`, the subjects dataset is shuffled at the beginning of each epoch, i.e. when all patches from all subjects have been processed.
- **shuffle_patches** – If `True`, patches are shuffled after filling the queue.
- **start_background** – If `True`, the loader will start working in the background as soon as the queue is instantiated.
- **verbose** – If `True`, some debugging messages will be printed.

This diagram represents the connection between a *SubjectsDataset*, a *Queue* and the DataLoader used to pop batches from the queue.

This sketch can be used to experiment and understand how the queue works. In this case, shuffle_subjects is False and shuffle_patches is True.

Note: num_workers refers to the number of workers used to load and transform the volumes. Multiprocessing is not needed to pop patches from the queue, so you should always use num_workers=0 for the DataLoader you instantiate to generate training batches.

Example:

```
>>> import torch
>>> import torchio as tio
>>> from torch.utils.data import DataLoader
>>> patch_size = 96
>>> queue_length = 300
>>> samples_per_volume = 10
>>> sampler = tio.data.UniformSampler(patch_size)
>>> subject = tio.datasets.Colin27()
>>> subjects_dataset = tio.SubjectsDataset(10 * [subject])
>>> patches_queue = tio.Queue(
...     subjects_dataset,
...     queue_length,
...     samples_per_volume,
...     sampler,
...     num_workers=4,
... )
>>> patches_loader = DataLoader(
...     patches_queue,
...     batch_size=16,
...     num_workers=0,  # this must be 0
... )
>>> num_epochs = 2
>>> model = torch.nn.Identity()
>>> for epoch_index in range(num_epochs):
...     for patches_batch in patches_loader:
...         inputs = patches_batch['t1'][tio.DATA]  # key 't1' is in subject
...         targets = patches_batch['brain'][tio.DATA]  # key 'brain' is in subject
...         logits = model(inputs)  # model being an instance of torch.nn.Module
```

Example:

```
>>> # Usage with distributed training
>>> import torch.distributed as dist
>>> from torch.utils.data.distributed import DistributedSampler
>>> # Assume a process running on distributed node 3
>>> rank = 3
>>> patch_sampler = tio.data.UniformSampler(patch_size)
>>> subject = tio.datasets.Colin27()
>>> subjects_dataset = tio.SubjectsDataset(10 * [subject])
>>> subject_sampler = dist.DistributedSampler(
...     subjects_dataset,
```

(continues on next page)

(continued from previous page)

```
...     rank=local_rank,
...     shuffle=True,
...     drop_last=True,
... )
>>> # Each process is assigned (len(subjects_dataset) // num_processes) subjects
>>> patches_queue = tio.Queue(
...     subjects_dataset,
...     queue_length,
...     samples_per_volume,
...     patch_sampler,
...     num_workers=4,
...     subject_sampler=subject_sampler,
... )
>>> patches_loader = DataLoader(
...     patches_queue,
...     batch_size=16,
...     num_workers=0,  # this must be 0
... )
>>> num_epochs = 2
>>> model = torch.nn.Identity()
>>> for epoch_index in range(num_epochs):
...     subject_sampler.set_epoch(epoch_index)
...     for patches_batch in patches_loader:
...         inputs = patches_batch['t1'][tio.DATA]  # key 't1' is in subject
...         targets = patches_batch['brain'][tio.DATA]  # key 'brain' is in subject
...         logits = model(inputs)  # model being an instance of torch.nn.Module
```

get_max_memory(*subject:* Subject | *None* = *None*) → int

Get the maximum RAM occupied by the patches queue in bytes.

Parameters
subject – Sample subject to compute the size of a patch.

get_max_memory_pretty(*subject:* Subject | *None* = *None*) → str

Get human-readable maximum RAM occupied by the patches queue.

Parameters
subject – Sample subject to compute the size of a patch.

1.3.2 Inference

Here is an example that uses a grid sampler and aggregator to perform dense inference across a 3D image using patches:

```
>>> import torch
>>> import torch.nn as nn
>>> import torchio as tio
>>> patch_overlap = 4, 4, 4  # or just 4
>>> patch_size = 88, 88, 60
>>> subject = tio.datasets.Colin27()
>>> subject
Colin27(Keys: ('t1', 'head', 'brain'); images: 3)
>>> grid_sampler = tio.inference.GridSampler(
```

(continues on next page)

(continued from previous page)

```
...     subject,
...     patch_size,
...     patch_overlap,
... )
>>> patch_loader = torch.utils.data.DataLoader(grid_sampler, batch_size=4)
>>> aggregator = tio.inference.GridAggregator(grid_sampler)
>>> model = nn.Identity().eval()
>>> with torch.no_grad():
...     for patches_batch in patch_loader:
...         input_tensor = patches_batch['t1'][tio.DATA]
...         locations = patches_batch[tio.LOCATION]
...         logits = model(input_tensor)
...         labels = logits.argmax(dim=tio.CHANNELS_DIMENSION, keepdim=True)
...         outputs = labels
...         aggregator.add_batch(outputs, locations)
>>> output_tensor = aggregator.get_output_tensor()
```

Grid sampler

GridSampler

class torchio.data.**GridSampler**(*subject:* Subject, *patch_size: int | Tuple[int, int, int]*, *patch_overlap: int | Tuple[int, int, int] = (0, 0, 0)*, *padding_mode: str | float | None = None*)

Bases: *PatchSampler*

Extract patches across a whole volume.

Grid samplers are useful to perform inference using all patches from a volume. It is often used with a *GridAggregator*.

Parameters

- **subject** – Instance of `Subject` from which patches will be extracted.
- **patch_size** – Tuple of integers (w, h, d) to generate patches of size $w \times h \times d$. If a single number n is provided, $w = h = d = n$.
- **patch_overlap** – Tuple of even integers (w_o, h_o, d_o) specifying the overlap between patches for dense inference. If a single number n is provided, $w_o = h_o = d_o = n$.
- **padding_mode** – Same as `padding_mode` in *Pad*. If `None`, the volume will not be padded before sampling and patches at the border will not be cropped by the aggregator. Otherwise, the volume will be padded with $\left(\frac{w_o}{2}, \frac{h_o}{2}, \frac{d_o}{2}\right)$ on each side before sampling. If the sampler is passed to a *GridAggregator*, it will crop the output to its original size.

Example

```
>>> import torchio as tio
>>> colin = tio.datasets.Colin27()
>>> sampler = tio.GridSampler(colin, patch_size=88)
>>> for i, patch in enumerate(sampler()):
...     patch.t1.save(f'patch_{i}.nii.gz')
...
>>> # To figure out the number of patches beforehand:
>>> sampler = tio.GridSampler(colin, patch_size=88)
>>> len(sampler)
8
```

Note: Adapted from NiftyNet. See this NiftyNet tutorial for more information about patch based sampling. Note that `patch_overlap` is twice `border` in NiftyNet tutorial.

Grid aggregator

GridAggregator

class torchio.data.**GridAggregator**(*sampler:* GridSampler, *overlap_mode: str = 'crop'*)

Bases: `object`

Aggregate patches for dense inference.

This class is typically used to build a volume made of patches after inference of batches extracted by a *GridSampler*.

Parameters

- **sampler** – Instance of *GridSampler* used to extract the patches.
- **overlap_mode** – If `'crop'`, the overlapping predictions will be cropped. If `'average'`, the predictions in the overlapping areas will be averaged with equal weights. If `'hann'`, the predictions in the overlapping areas will be weighted with a Hann window function. See the grid aggregator tests for a raw visualization of the three modes.

Note: Adapted from NiftyNet. See this NiftyNet tutorial for more information about patch-based sampling.

add_batch(*batch_tensor: Tensor*, *locations: Tensor*) → None

Add batch processed by a CNN to the output prediction volume.

Parameters

- **batch_tensor** – 5D tensor, typically the output of a convolutional neural network, e.g. `batch['image'][torchio.DATA]`.
- **locations** – 2D tensor with shape $(B, 6)$ representing the patch indices in the original image. They are typically extracted using `batch[torchio.LOCATION]`.

get_output_tensor() → Tensor

Get the aggregated volume after dense inference.

1.4 Transforms

TorchIO transforms take as input instances of *Subject* or *Image* (and its subclasses), 4D PyTorch tensors, 4D NumPy arrays, SimpleITK images, NiBabel images, or Python dictionaries (see *Transform*).

For example:

```
>>> import torch
>>> import numpy as np
>>> import torchio as tio
>>> affine_transform = tio.RandomAffine()
>>> tensor = torch.rand(1, 256, 256, 159)
>>> transformed_tensor = affine_transform(tensor)
>>> type(transformed_tensor)
<class 'torch.Tensor'>
>>> array = np.random.rand(1, 256, 256, 159)
>>> transformed_array = affine_transform(array)
>>> type(transformed_array)
<class 'numpy.ndarray'>
>>> subject = tio.datasets.Colin27()
>>> transformed_subject = affine_transform(subject)
>>> transformed_subject
Subject(Keys: ('t1', 'head', 'brain'); images: 3)
```

Transforms can also be applied from the command line using *torchio-transform*.

All transforms inherit from *torchio.transforms.Transform*:

class torchio.transforms.**Transform**(*p: float = 1*, *copy: bool = True*, *include: Sequence[str] | None = None*, *exclude: Sequence[str] | None = None*, *keys: Sequence[str] | None = None*, *keep: Dict[str, str] | None = None*, *parse_input: bool = True*, *label_keys: Sequence[str] | None = None*)

Bases: ABC

Abstract class for all TorchIO transforms.

When called, the input can be an instance of *torchio.Subject*, *torchio.Image*, numpy.ndarray, torch.Tensor, SimpleITK.Image, or dict containing 4D tensors as values.

All subclasses must overwrite apply_transform(), which takes an instance of *Subject*, modifies it and returns the result.

Parameters

- **p** – Probability that this transform will be applied.
- **copy** – Make a shallow copy of the input before applying the transform.
- **include** – Sequence of strings with the names of the only images to which the transform will be applied. Mandatory if the input is a dict.
- **exclude** – Sequence of strings with the names of the images to which the the transform will not be applied, apart from the ones that are excluded because of the transform type. For example, if a subject includes an MRI, a CT and a label map, and the CT is added to the list of exclusions of an intensity transform such as *RandomBlur*, the transform will be only applied to the MRI, as the label map is excluded by default by spatial transforms.

- **keep** – Dictionary with the names of the input images that will be kept in the output and their new names. For example: `{'t1': 't1_original'}`. This might be useful for autoencoders or registration tasks.
- **parse_input** – If `True`, the input will be converted to an instance of *Subject*. This is used internally by some special transforms like *Compose*.
- **label_keys** – If the input is a dictionary, names of images that correspond to label maps.

__call__(*data:* Subject | Image | *Tensor* | *ndarray* | *Image* | *dict* | *Nifti1Image*) → *Subject* | *Image* | Tensor | ndarray | Image | dict | Nifti1Image

Transform data and return a result of the same type.

Parameters

data – Instance of *torchio.Subject*, 4D `torch.Tensor` or `numpy.ndarray` with dimensions (C, W, H, D), where C is the number of channels and W, H, D are the spatial dimensions. If the input is a tensor, the affine matrix will be set to identity. Other valid input types are a SimpleITK image, a *torchio.Image*, a NiBabel Nifti1 image or a `dict`. The output type is the same as the input type.

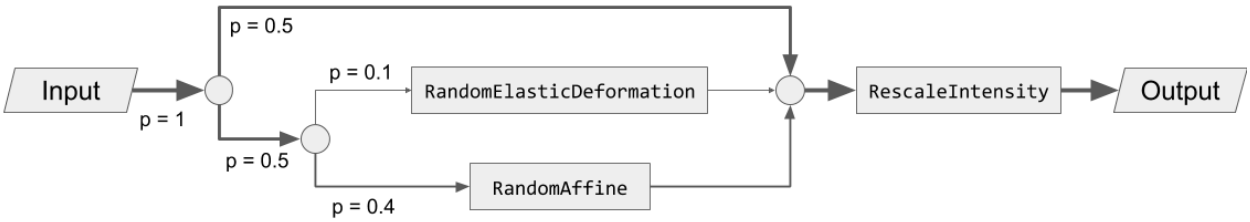
static validate_keys_sequence(*keys:* *Sequence[str]* | *None*, *name:* *str*) → None

Ensure that the input is not a string but a sequence of strings.

1.4.1 Composability

Transforms can be composed to create directed acyclic graphs defining the probability that each transform will be applied.

For example, to obtain the following graph:



We can type:

```
>>> import torchio as tio
>>> spatial_transforms = {
...     tio.RandomElasticDeformation(): 0.2,
...     tio.RandomAffine(): 0.8,
... }
>>> transform = tio.Compose([
...     tio.OneOf(spatial_transforms, p=0.5),
...     tio.RescaleIntensity(out_min_max=(0, 1)),
... ])
```

1.4.2 Reproducibility

When transforms are instantiated, we typically need to pass values that will be used to sample the transform parameters when the *__call__()* method of the transform is called, i.e., when the transform instance is called.

All random transforms have a corresponding deterministic class, that can be applied again to obtain exactly the same result. The *Subject* class contains some convenient methods to reproduce transforms:

```
>>> import torchio as tio
>>> subject = tio.datasets.FPG()
>>> transforms = (
...     tio.CropOrPad((100, 200, 300)),
...     tio.RandomFlip(axes=['LR', 'AP', 'IS']),
...     tio.OneOf([tio.RandomAnisotropy(), tio.RandomElasticDeformation()]),
... )
>>> transform = tio.Compose(transforms)
>>> transformed = transform(subject)
>>> reproduce_transform = transformed.get_composed_history()
>>> reproduce_transform
Compose(
    Pad(padding=(0, 0, 0, 0, 62, 62), padding_mode=constant)
    Crop(cropping=(78, 78, 28, 28, 0, 0))
    Flip(axes=(...))
    Resample(target=(...), image_interpolation=nearest, pre_affine_name=None)
    Resample(target=ScalarImage(...), image_interpolation=linear, pre_affine_name=None)
)
>>> reproduced = reproduce_transform(subject)
```

1.4.3 Invertibility

Inverting transforms can be especially useful in scenarios in which one needs to apply some transformation, infer a segmentation on the transformed data and apply the inverse transform to the inference in order to bring it back to the original space.

This is particularly useful, for example, for test-time augmentation or aleatoric uncertainty estimation:

```
>>> import torchio as tio
>>> # Mock a segmentation CNN
>>> def model(x):
...     return x
...
>>> subject = tio.datasets.Colin27()
>>> transform = tio.RandomAffine()
>>> segmentations = []
>>> num_segmentations = 10
>>> for _ in range(num_segmentations):
...     transform = tio.RandomAffine(image_interpolation='bspline')
...     transformed = transform(subject)
...     segmentation = model(transformed)
...     transformed_native_space = segmentation.apply_inverse_transform(image_
↪interpolation='linear')
...     segmentations.append(transformed_native_space)
...
```

Transforms can be classified in three types, according to their degree of invertibility:

- Lossless: transforms that can be inverted with no loss of information, such as *RandomFlip*, *Pad*, or *RandomNoise*.
- Lossy: transforms that can be inverted with some loss of information, such as *RandomAffine*, or *Crop*.
- Impossible: transforms that cannot be inverted, such as *RandomBlur*.

Non-invertible transforms will be ignored by the *apply_inverse_transform()* method of *Subject*.

1.4.4 Interpolation

Some transforms such as *RandomAffine* or *RandomMotion* need to interpolate intensity values during resampling.

The available interpolation strategies can be inferred from the elements of *Interpolation*.

`'linear'` interpolation, the default in TorchIO for scalar images, is usually a good compromise between image quality and speed. It is therefore a good choice for data augmentation during training.

Methods such as `'bspline'` or `'lanczos'` generate high-quality results, but are generally slower. They can be used to obtain optimal resampling results during offline data preprocessing.

`'nearest'` can be used for quick experimentation as it is very fast, but produces relatively poor results for scalar images. It is the default interpolation type for label maps, as categorical values for the different labels need to preserved after interpolation.

When instantiating transforms, it is possible to specify independently the interpolation type for label maps and scalar images, as shown in the documentation for, e.g., *Resample*.

Visit the SimpleITK docs for technical documentation and Cambridge in Colour for some further general explanations of digital image interpolation.

class torchio.transforms.interpolation.**Interpolation**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Interpolation techniques available in ITK.

For a full quantitative comparison of interpolation methods, you can read Meijering et al. 1999, Quantitative Comparison of Sinc-Approximating Kernels for Medical Image Interpolation

Example

```
>>> import torchio as tio
>>> transform = tio.RandomAffine(image_interpolation='bspline')
```

BLACKMAN: str = 'sitkBlackmanWindowedSinc'

Blackman windowed sinc kernel.

BSPLINE: str = 'sitkBSpline'

B-Spline of order 3 (cubic) interpolation.

COSINE: str = 'sitkCosineWindowedSinc'

Cosine windowed sinc kernel.

CUBIC: str = 'sitkBSpline'

Same as nearest.

GAUSSIAN: str = 'sitkGaussian'

Gaussian interpolation. Sigma is set to 0.8 input pixels and alpha is 4

HAMMING: str = 'sitkHammingWindowedSinc'

Hamming windowed sinc kernel.

LABEL_GAUSSIAN: str = 'sitkLabelGaussian'

Smoothly interpolate multi-label images. Sigma is set to 1 input pixel and alpha is 1

LANCZOS: str = 'sitkLanczosWindowedSinc'

Lanczos windowed sinc kernel.

LINEAR: str = 'sitkLinear'

Linearly interpolates image intensity at a non-integer pixel position.

NEAREST: str = 'sitkNearestNeighbor'

Interpolates image intensity at a non-integer pixel position by copying the intensity for the nearest neighbor.

WELCH: str = 'sitkWelchWindowedSinc'

Welch windowed sinc kernel.

1.4.5 Transforms API

Preprocessing

Intensity

RescaleIntensity

class torchio.transforms.**RescaleIntensity**(*out_min_max: Tuple[float, float] = (0, 1)*, *percentiles: Tuple[float, float] = (0, 100)*, *masking_method: str | Callable[[Tensor], Tensor] | int | Tuple[int, int, int] | Tuple[int, int, int, int, int, int] | None = None*, *in_min_max: Tuple[float, float] | None = None*, ***kwargs*)

Bases: *NormalizationTransform*

Rescale intensity values to a certain range.

Parameters

- **out_min_max** – Range (n_{min}, n_{max}) of output intensities. If only one value d is provided, $(n_{min}, n_{max}) = (-d, d)$.
- **percentiles** – Percentile values of the input image that will be mapped to (n_{min}, n_{max}). They can be used for contrast stretching, as in this scikit-image example. For example, Isensee et al. use (0.5, 99.5) in their nn-UNet paper. If only one value d is provided, $(n_{min}, n_{max}) = (0, d)$.
- **masking_method** – See *NormalizationTransform*.
- **in_min_max** – Range (m_{min}, m_{max}) of input intensities that will be mapped to (n_{min}, n_{max}). If None, the minimum and maximum input intensities will be used.
- ****kwargs** – See *Transform* for additional keyword arguments.

Example

```
>>> import torchio as tio
>>> ct = tio.ScalarImage('ct_scan.nii.gz')
>>> ct_air, ct_bone = -1000, 1000
>>> rescale = tio.RescaleIntensity(
...     out_min_max=(-1, 1), in_min_max=(ct_air, ct_bone))
>>> ct_normalized = rescale(ct)
```

ZNormalization

class torchio.transforms.**ZNormalization**(*masking_method: str | Callable[[Tensor], Tensor] | int | Tuple[int, int, int] | Tuple[int, int, int, int, int, int] | None = None*, ***kwargs*)

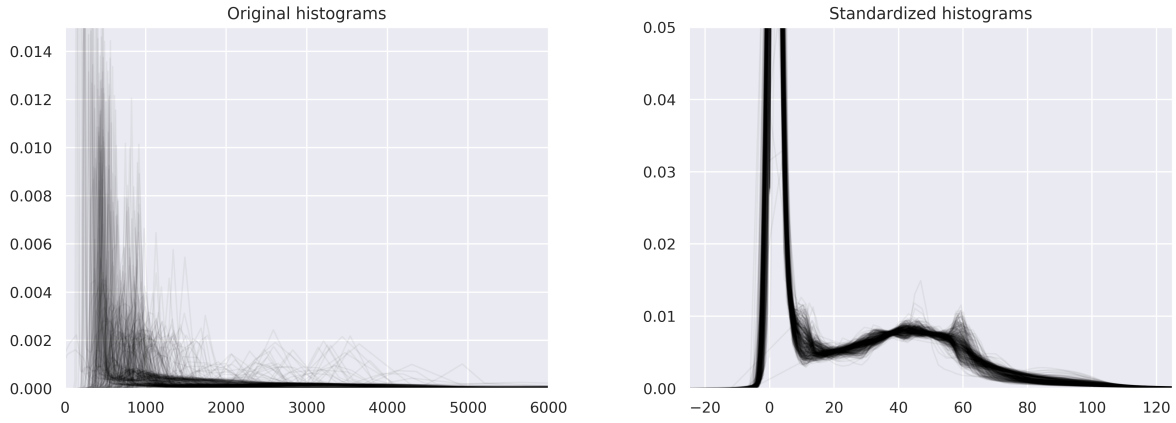
Bases: *NormalizationTransform*

Subtract mean and divide by standard deviation.

Parameters

- **masking_method** – See *NormalizationTransform*.
- ****kwargs** – See *Transform* for additional keyword arguments.

HistogramStandardization



class torchio.transforms.**HistogramStandardization**(*landmarks: str | Path | Dict[str, str | Path | ndarray], masking_method: str | Callable[[Tensor], Tensor] | int | Tuple[int, int, int] | Tuple[int, int, int, int, int, int] | None = None*, ***kwargs*)

Bases: *NormalizationTransform*

Perform histogram standardization of intensity values.

Implementation of New variants of a method of MRI scale standardization.

See example in *torchio.transforms.HistogramStandardization.train()*.

Parameters

- **landmarks** – Dictionary (or path to a PyTorch file with `.pt` or `.pth` extension in which a dictionary has been saved) whose keys are image names in the subject and values are NumPy arrays or paths to NumPy arrays defining the landmarks after training with *torchio.transforms.HistogramStandardization.train()*.
- **masking_method** – See *NormalizationTransform*.
- ****kwargs** – See *Transform* for additional keyword arguments.

Example

```
>>> import torch
>>> import torchio as tio
>>> landmarks = {
...     't1': 't1_landmarks.npy',
...     't2': 't2_landmarks.npy',
... }
>>> transform = tio.HistogramStandardization(landmarks)
>>> torch.save(landmarks, 'path_to_landmarks.pth')
>>> transform = tio.HistogramStandardization('path_to_landmarks.pth')
```

classmethod train(*images_paths: Sequence[str | Path]*, *cutoff: Tuple[float, float] | None = None*, *mask_path: str | Path | Sequence[str | Path] | None = None*, *masking_function: Callable | None = None*, *output_path: str | Path | None = None*) → ndarray

Extract average histogram landmarks from images used for training.

Parameters

- **images_paths** – List of image paths used to train.
- **cutoff** – Optional minimum and maximum quantile values, respectively, that are used to select a range of intensity of interest. Equivalent to pc_1 and pc_2 in Nyúl and Udupa's paper.
- **mask_path** – Path (or list of paths) to a binary image that will be used to select the voxels use to compute the stats during histogram training. If `None`, all voxels in the image will be used.
- **masking_function** – Function used to extract voxels used for histogram training.
- **output_path** – Optional file path with extension `.txt` or `.npy`, where the landmarks will be saved.

Example

```
>>> import torch
>>> import numpy as np
>>> from pathlib import Path
>>> from torchio.transforms import HistogramStandardization
>>>
>>> t1_paths = ['subject_a_t1.nii', 'subject_b_t1.nii.gz']
>>> t2_paths = ['subject_a_t2.nii', 'subject_b_t2.nii.gz']
>>>
>>> t1_landmarks_path = Path('t1_landmarks.npy')
```

(continues on next page)

(continued from previous page)

```
>>> t2_landmarks_path = Path('t2_landmarks.npy')
>>>
>>> t1_landmarks = (
...     t1_landmarks_path
...     if t1_landmarks_path.is_file()
...     else HistogramStandardization.train(t1_paths)
... )
>>> np.save(t1_landmarks_path, t1_landmarks)
>>>
>>> t2_landmarks = (
...     t2_landmarks_path
...     if t2_landmarks_path.is_file()
...     else HistogramStandardization.train(t2_paths)
... )
>>> np.save(t2_landmarks_path, t2_landmarks)
>>>
>>> landmarks_dict = {
...     't1': t1_landmarks,
...     't2': t2_landmarks,
... }
>>>
>>> transform = HistogramStandardization(landmarks_dict)
```

Mask

class torchio.transforms.**Mask**(*masking_method: str | Callable[[Tensor], Tensor] | int | Tuple[int, int, int] | Tuple[int, int, int, int, int, int] | None*, *outside_value: float = 0*, *labels: Sequence[int] | None = None*, ***kwargs*)

Bases: `IntensityTransform`

Set voxels outside of mask to a constant value.

Parameters

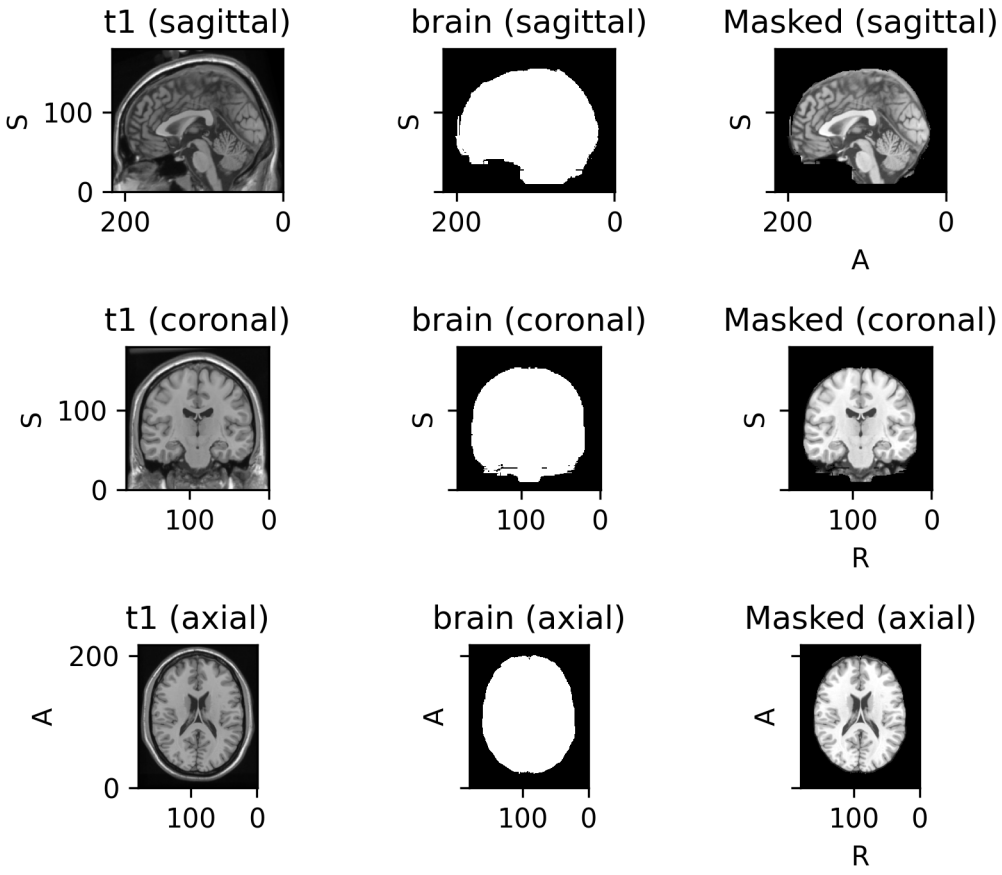
- **masking_method** – See *NormalizationTransform*.
- **outside_value** – Value to set for all voxels outside of the mask.
- **labels** – If a label map is used to generate the mask, sequence of labels to consider. If `None`, all values larger than zero will be used for the mask.
- ****kwargs** – See *Transform* for additional keyword arguments.

Raises

`RuntimeWarning` – If a 4D image is masked with a 3D mask, the mask will be expanded along the channels (first) dimension, and a warning will be raised.

Example

```
>>> import torchio as tio
>>> subject = tio.datasets.Colin27()
>>> subject
Colin27(Keys: ('t1', 'head', 'brain'); images: 3)
>>> mask = tio.Mask(masking_method='brain')  # Use "brain" image to mask
>>> transformed = mask(subject)  # Set voxels outside of the brain to 0
```



Clamp

class torchio.transforms.**Clamp**(*out_min: float | None = None*, *out_max: float | None = None*, ***kwargs*)

Bases: `IntensityTransform`

Clamp intensity values into a range $[a, b]$.

For more information, see `torch.clamp()`.

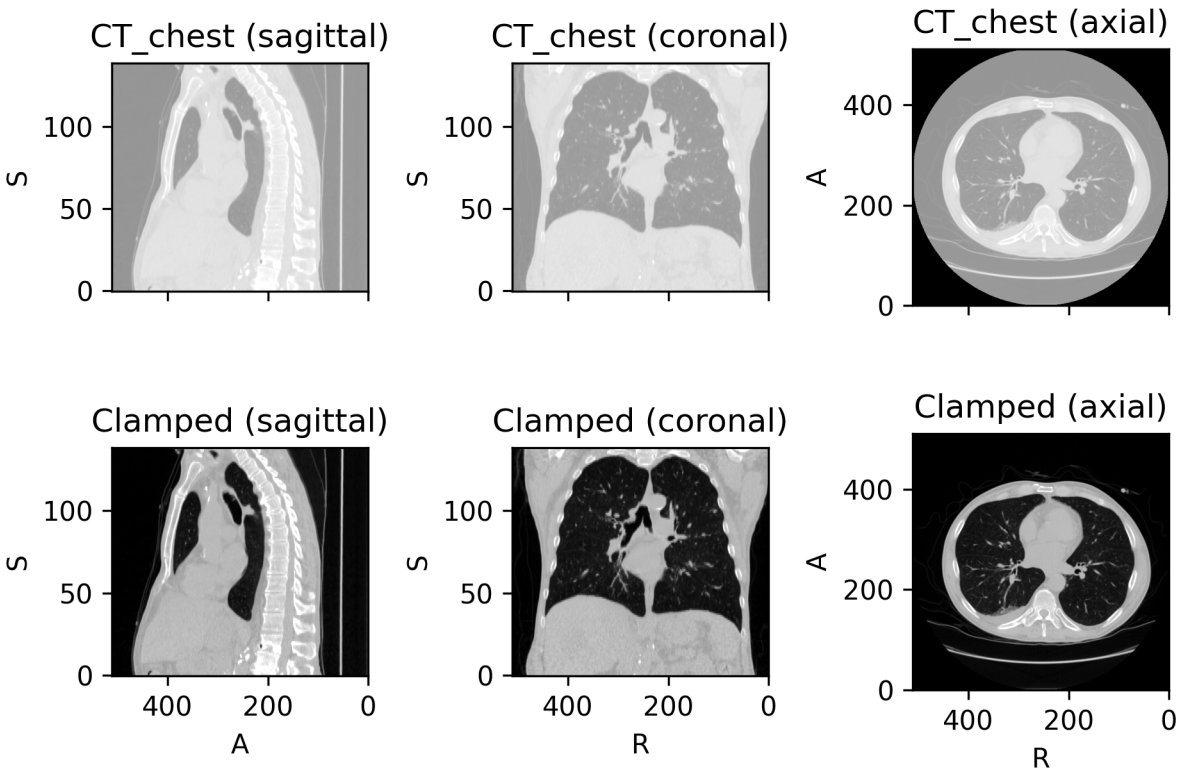
Parameters

- **out_min** – Minimum value a of the output image. If `None`, the minimum of the image is used.

- **out_max** – Maximum value b of the output image. If None, the maximum of the image is used.

Example

```
>>> import torchio as tio
>>> ct = tio.datasets.Slicer('CTChest').CT_chest
>>> HOUNSFIELD_AIR, HOUNSFIELD_BONE = -1000, 1000
>>> clamp = tio.Clamp(out_min=HOUNSFIELD_AIR, out_max=HOUNSFIELD_BONE)
>>> ct_clamped = clamp(ct)
```



NormalizationTransform

class torchio.transforms.preprocessing.intensity.**NormalizationTransform**(*masking_method: str | Callable[[Tensor], Tensor] | int | Tuple[int, int, int] | Tuple[int, int, int, int, int, int] | None = None*, ***kwargs*)

Bases: IntensityTransform

Base class for intensity preprocessing transforms.

Parameters

- **masking_method** – Defines the mask used to compute the normalization statistics. It can be one of:
  - `None`: the mask image is all ones, i.e. all values in the image are used.
  - A string: key to a *torchio.LabelMap* in the subject which is used as a mask, OR an anatomical label: `'Left'`, `'Right'`, `'Anterior'`, `'Posterior'`, `'Inferior'`, `'Superior'` which specifies a side of the mask volume to be ones.
  - A function: the mask image is computed as a function of the intensity image. The function must receive and return a `torch.Tensor`
- ****kwargs** – See *Transform* for additional keyword arguments.

Example

```
>>> import torchio as tio
>>> subject = tio.datasets.Colin27()
>>> subject
Colin27(Keys: ('t1', 'head', 'brain'); images: 3)
>>> transform = tio.ZNormalization()  # ZNormalization is a subclass of␣
↪NormalizationTransform
>>> transformed = transform(subject)  # use all values to compute mean and std
>>> transform = tio.ZNormalization(masking_method='brain')
>>> transformed = transform(subject)  # use only values within the brain
>>> transform = tio.ZNormalization(masking_method=lambda x: x > x.mean())
>>> transformed = transform(subject)  # use values above the image mean
```

Spatial

CropOrPad

class torchio.transforms.**CropOrPad**(*target_shape: int | Tuple[int, int, int] | None = None*, *padding_mode: str | float = 0*, *mask_name: str | None = None*, *labels: Sequence[int] | None = None*, ***kwargs*)

Bases: `SpatialTransform`

Modify the field of view by cropping or padding to match a target shape.

This transform modifies the affine matrix associated to the volume so that physical positions of the voxels are maintained.

Parameters

- **target_shape** – Tuple (W, H, D). If a single value N is provided, then $W = H = D = N$. If `None`, the shape will be computed from the `mask_name` (and the `labels`, if `labels` is not `None`).
- **padding_mode** – Same as `padding_mode` in *Pad*.
- **mask_name** – If `None`, the centers of the input and output volumes will be the same. If a string is given, the output volume center will be the center of the bounding box of non-zero values in the image named `mask_name`.

- **labels** – If a label map is used to generate the mask, sequence of labels to consider.
- ****kwargs** – See *Transform* for additional keyword arguments.

Example

```
>>> import torchio as tio
>>> subject = tio.Subject(
...     chest_ct=tio.ScalarImage('subject_a_ct.nii.gz'),
...     heart_mask=tio.LabelMap('subject_a_heart_seg.nii.gz'),
... )
>>> subject.chest_ct.shape
torch.Size([1, 512, 512, 289])
>>> transform = tio.CropOrPad(
...     (120, 80, 180),
...     mask_name='heart_mask',
... )
>>> transformed = transform(subject)
>>> transformed.chest_ct.shape
torch.Size([1, 120, 80, 180])
```

Warning: If `target_shape` is `None`, subjects in the dataset will probably have different shapes. This is probably fine if you are using patch-based training. If you are using full volumes for training and a batch size larger than one, an error will be raised by the `DataLoader` while trying to collate the batches.

static _get_six_bounds_parameters(*parameters: ndarray*) → Tuple[int, int, int, int, int, int]

Compute bounds parameters for ITK filters.

Parameters

parameters – Tuple (w, h, d) with the number of voxels to be cropped or padded.

Returns

Tuple $(w_{ini}, w_{fin}, h_{ini}, h_{fin}, d_{ini}, d_{fin})$, where $n_{ini} = \lceil \frac{n}{2} \rceil$ and $n_{fin} = \lfloor \frac{n}{2} \rfloor$.

Example

```
>>> p = np.array((4, 0, 7))
>>> CropOrPad._get_six_bounds_parameters(p)
(2, 2, 0, 0, 4, 3)
```

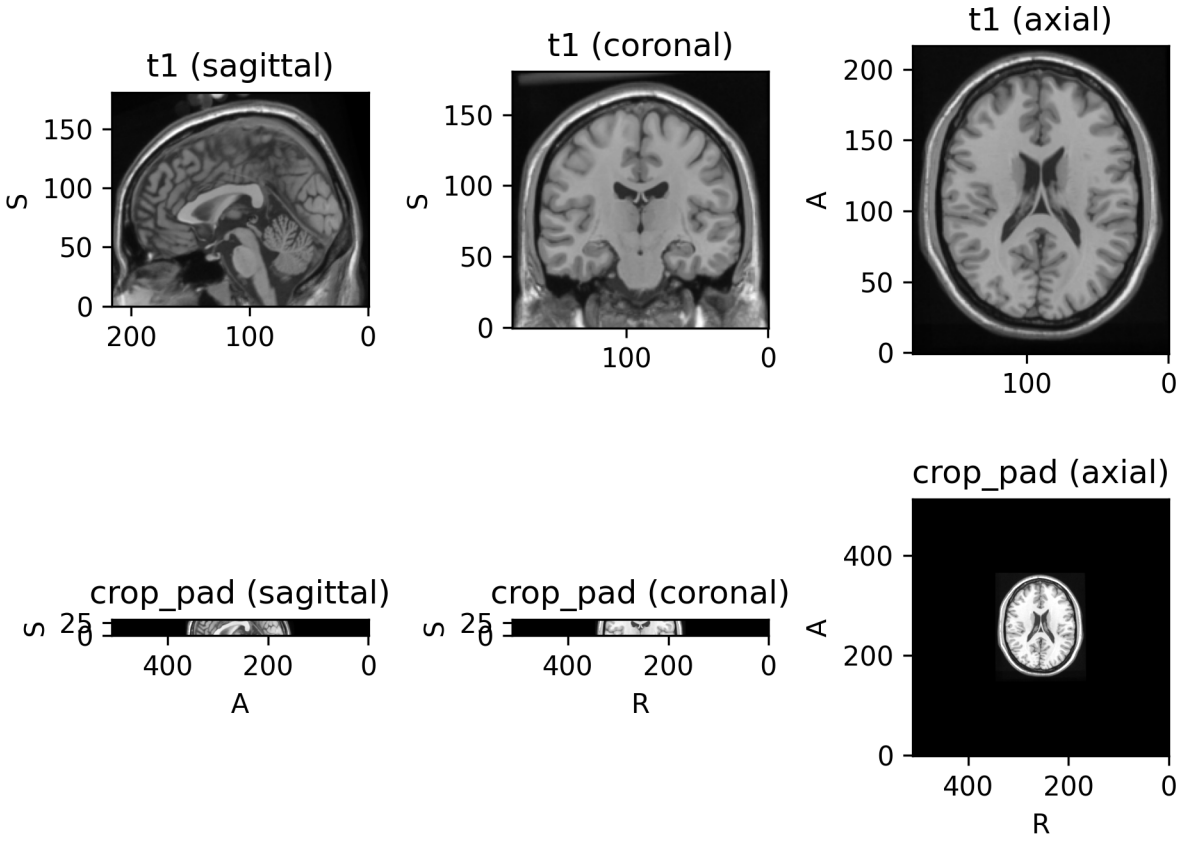
ToCanonical

class torchio.transforms.**ToCanonical**(*p: float = 1*, *copy: bool = True*, *include: Sequence[str] | None = None*, *exclude: Sequence[str] | None = None*, *keys: Sequence[str] | None = None*, *keep: Dict[str, str] | None = None*, *parse_input: bool = True*, *label_keys: Sequence[str] | None = None*)

Bases: `SpatialTransform`

Reorder the data to be closest to canonical (RAS+) orientation.

This transform reorders the voxels and modifies the affine matrix so that the voxel orientations are nearest to:


t1 (sagittal)
t1 (coronal)
t1 (axial)
crop_pad (sagittal)
crop_pad (coronal)
crop_pad (axial)
S
A
R

1. First voxel axis goes from left to Right
2. Second voxel axis goes from posterior to Anterior
3. Third voxel axis goes from inferior to Superior

See NiBabel docs about image orientation for more information.

Parameters

****kwargs** – See *Transform* for additional keyword arguments.

Note: The reorientation is performed using `nibabel.as_closest_canonical()`.

Resample

class torchio.transforms.**Resample**(*target: float | Tuple[float, float, float] | str | Path | Image | None = 1*, *image_interpolation: str = 'linear'*, *label_interpolation: str = 'nearest'*, *pre_affine_name: str | None = None*, *scalars_only: bool = False*, ***kwargs*)

Bases: `SpatialTransform`

Resample image to a different physical space.

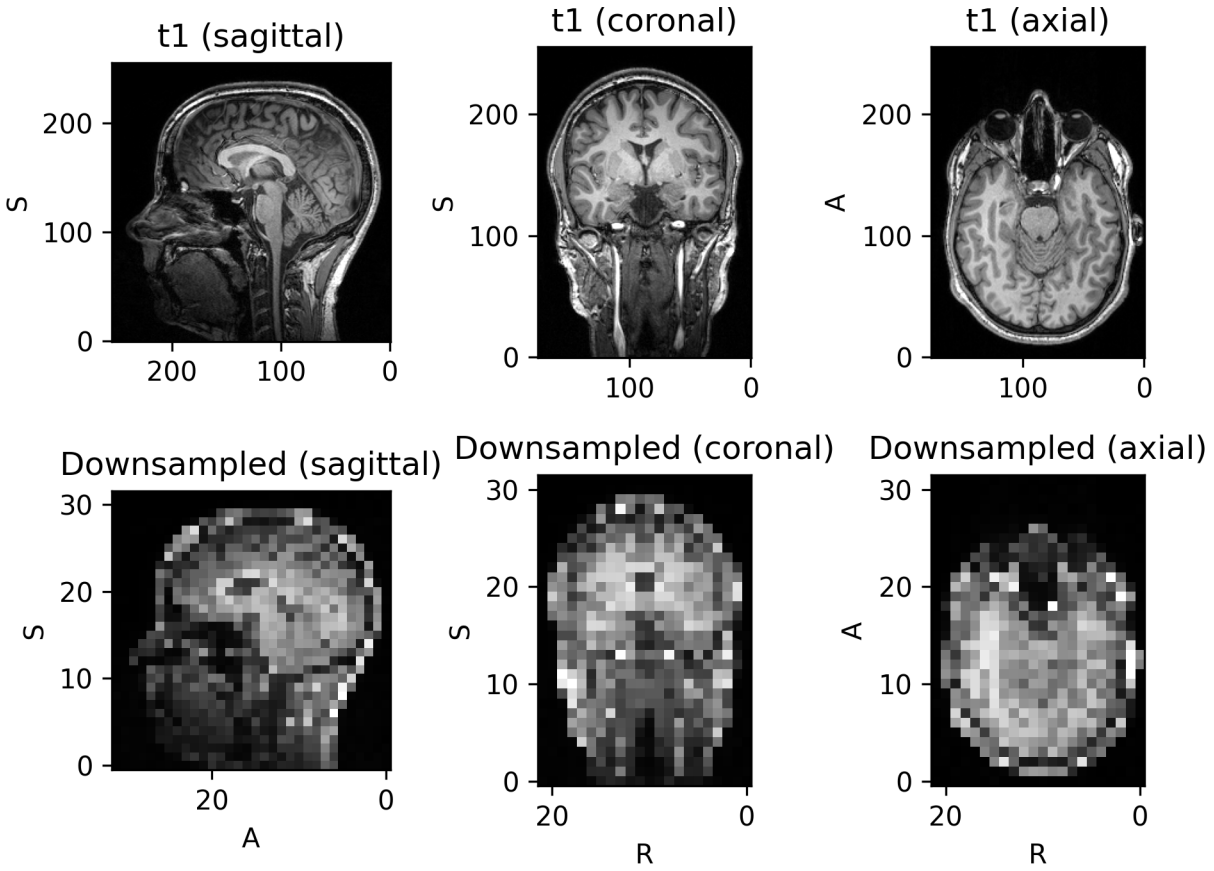
This is a powerful transform that can be used to change the image shape or spatial metadata, or to apply a spatial transformation.

Parameters

- **target** – Argument to define the output space. Can be one of:
  - Output spacing (s_w, s_h, s_d), in mm. If only one value s is specified, then $s_w = s_h = s_d = s$.
  - Path to an image that will be used as reference.
  - Instance of *Image*.
  - Name of an image key in the subject.
  - Tuple `(spatial_shape, affine)` defining the output space.
- **pre_affine_name** – Name of the *image key* (not subject key) storing an affine matrix that will be applied to the image header before resampling. If `None`, the image is resampled with an identity transform. See usage in the example below.
- **image_interpolation** – See *Interpolation*.
- **label_interpolation** – See *Interpolation*.
- **scalars_only** – Apply only to instances of *ScalarImage*. Used internally by *RandomAnisotropy*.
- ****kwargs** – See *Transform* for additional keyword arguments.

Example

```
>>> import torch
>>> import torchio as tio
>>> transform = tio.Resample(1)                        # resample all images to 1mm iso
>>> transform = tio.Resample((2, 2, 2))                # resample all images to 2mm iso
>>> transform = tio.Resample('t1')                     # resample all images to 't1'␣
↪image space
>>> # Example: using a precomputed transform to MNI space
>>> ref_path = tio.datasets.Colin27().t1.path  # this image is in the MNI space, so␣
↪we can use it as reference/target
>>> affine_matrix = tio.io.read_matrix('transform_to_mni.txt')  # from a NiftyReg␣
↪registration. Would also work with e.g. .tfm from SimpleITK
>>> image = tio.ScalarImage(tensor=torch.rand(1, 256, 256, 180), to_mni=affine_
↪matrix)  # 'to_mni' is an arbitrary name
>>> transform = tio.Resample(colin.t1.path, pre_affine_name='to_mni')  # nearest␣
↪neighbor interpolation is used for label maps
>>> transformed = transform(image)  # "image" is now in the MNI space
```



Resize

class torchio.transforms.**Resize**(*target_shape: int | Tuple[int, int, int]*, *image_interpolation: str = 'linear'*, *label_interpolation: str = 'nearest'*, ***kwargs*)

Bases: SpatialTransform

Resample images so the output shape matches the given target shape.

The field of view remains the same.

Warning: In most medical image applications, this transform should not be used as it will deform the physical object by scaling anistropically along the different dimensions. The solution to change an image size is typically applying *Resample* and *CropOrPad*.

Parameters

- **target_shape** – Tuple (W, H, D). If a single value N is provided, then $W = H = D = N$. The size of dimensions set to -1 will be kept.
- **image_interpolation** – See *Interpolation*.
- **label_interpolation** – See *Interpolation*.

EnsureShapeMultiple

class torchio.transforms.**EnsureShapeMultiple**(*target_multiple: int | Tuple[int, int, int]*, ***, *method: str = 'pad'*, ***kwargs*)

Bases: SpatialTransform

Ensure that all values in the image shape are divisible by n.

Some convolutional neural network architectures need that the size of the input across all spatial dimensions is a power of 2.

For example, the canonical 3D U-Net from Çiçek et al. includes three downsampling (pooling) and upsampling operations:

Pooling operations in PyTorch round down the output size:

```
>>> import torch
>>> x = torch.rand(3, 10, 20, 31)
>>> x_down = torch.nn.functional.max_pool3d(x, 2)
>>> x_down.shape
torch.Size([3, 5, 10, 15])
```

If we upsample this tensor, the original shape is lost:

```
>>> x_down_up = torch.nn.functional.interpolate(x_down, scale_factor=2)
>>> x_down_up.shape
torch.Size([3, 10, 20, 30])
>>> x.shape
torch.Size([3, 10, 20, 31])
```

If we try to concatenate `x_down` and `x_down_up` (to create skip connections), we will get an error. It is therefore good practice to ensure that the size of our images is such that concatenations will be safe.

Note: In these examples, it's assumed that all convolutions in the U-Net use padding so that the output size is the same as the input size.

The image above shows 3 downsampling operations, so the input size along all dimensions should be a multiple of $2^3 = 8$.

Example (assuming `pip install unet` has been run before):

```
>>> import torchio as tio
>>> import unet
>>> net = unet.UNet3D(padding=1)
>>> t1 = tio.datasets.Colin27().t1
>>> tensor_bad = t1.data.unsqueeze(0)
>>> tensor_bad.shape
torch.Size([1, 1, 181, 217, 181])
>>> net(tensor_bad).shape
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/fernando/miniconda3/envs/resseg/lib/python3.7/site-packages/torch/nn/
↪modules/module.py", line 727, in _call_impl
    result = self.forward(*input, **kwargs)
  File "/home/fernando/miniconda3/envs/resseg/lib/python3.7/site-packages/unet/unet.
↪py", line 122, in forward
    x = self.decoder(skip_connections, encoding)
  File "/home/fernando/miniconda3/envs/resseg/lib/python3.7/site-packages/torch/nn/
↪modules/module.py", line 727, in _call_impl
    result = self.forward(*input, **kwargs)
  File "/home/fernando/miniconda3/envs/resseg/lib/python3.7/site-packages/unet/
↪decoding.py", line 61, in forward
    x = decoding_block(skip_connection, x)
  File "/home/fernando/miniconda3/envs/resseg/lib/python3.7/site-packages/torch/nn/
↪modules/module.py", line 727, in _call_impl
    result = self.forward(*input, **kwargs)
  File "/home/fernando/miniconda3/envs/resseg/lib/python3.7/site-packages/unet/
↪decoding.py", line 131, in forward
    x = torch.cat((skip_connection, x), dim=CHANNELS_DIMENSION)
RuntimeError: Sizes of tensors must match except in dimension 1. Got 45 and 44 in␣
↪dimension 2 (The offending index is 1)
>>> num_poolings = 3
>>> fix_shape_unet = tio.EnsureShapeMultiple(2**num_poolings)
>>> t1_fixed = fix_shape_unet(t1)
>>> tensor_ok = t1_fixed.data.unsqueeze(0)
>>> tensor_ok.shape
torch.Size([1, 1, 184, 224, 184])  # as expected
```

Parameters

- **target_multiple** – Tuple (n_w, n_h, n_d), so that the size of the output along axis i is a multiple of n_i. If a single value n is provided, then $n_w = n_h = n_d = n$.
- **method** – Either `'crop'` or `'pad'`.
- ****kwargs** – See *Transform* for additional keyword arguments.

Example

```
>>> import torchio as tio
>>> image = tio.datasets.Colin27().t1
>>> image.shape
(1, 181, 217, 181)
>>> transform = tio.EnsureShapeMultiple(8, method='pad')
>>> transformed = transform(image)
>>> transformed.shape
(1, 184, 224, 184)
>>> transform = tio.EnsureShapeMultiple(8, method='crop')
>>> transformed = transform(image)
>>> transformed.shape
(1, 176, 216, 176)
>>> image_2d = image.data[..., :1]
>>> image_2d.shape
torch.Size([1, 181, 217, 1])
>>> transformed = transform(image_2d)
>>> transformed.shape
torch.Size([1, 176, 216, 1])
```

CopyAffine

class torchio.transforms.**CopyAffine**(*target: str*, ***kwargs*)

Bases: SpatialTransform

Copy the spatial metadata from a reference image in the subject.

Small unexpected differences in spatial metadata across different images of a subject can arise due to rounding errors while converting formats.

If the `shape` and `orientation` of the images are the same and their `affine` attributes are different but very similar, this transform can be used to avoid errors during safety checks in other transforms and samplers.

Parameters

target – Name of the image within the subject whose affine matrix will be used.

Example

```
>>> import torch
>>> import torchio as tio
>>> import numpy as np
>>> np.random.seed(0)
>>> affine = np.diag((*(np.random.rand(3) + 0.5), 1))
>>> t1 = tio.ScalarImage(tensor=torch.rand(1, 100, 100, 100), affine=affine)
>>> # Let's simulate a loss of precision
>>> # (caused for example by NIfTI storing spatial metadata in single precision)
>>> bad_affine = affine.astype(np.float16)
>>> t2 = tio.ScalarImage(tensor=torch.rand(1, 100, 100, 100), affine=bad_affine)
>>> subject = tio.Subject(t1=t1, t2=t2)
>>> resample = tio.Resample(0.5)
>>> resample(subject).shape  # error as images are in different spaces
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/fernando/git/torchio/torchio/data/subject.py", line 101, in shape
    self.check_consistent_attribute('shape')
  File "/Users/fernando/git/torchio/torchio/data/subject.py", line 229, in check_
↪consistent_attribute
    raise RuntimeError(message)
RuntimeError: More than one shape found in subject images:
{'t1': (1, 210, 244, 221), 't2': (1, 210, 243, 221)}
>>> transform = tio.CopyAffine('t1')
>>> fixed = transform(subject)
>>> resample(fixed).shape
(1, 210, 244, 221)
```

Warning: This transform should be used with caution. Modifying the spatial metadata of an image manually can lead to incorrect processing of the position of anatomical structures. For example, a machine learning algorithm might incorrectly predict that a lesion on the right lung is on the left lung.

Note: For more information, see some related discussions on GitHub:

- https://github.com/fepegar/torchio/issues/354
- https://github.com/fepegar/torchio/discussions/489
- https://github.com/fepegar/torchio/pull/584
- https://github.com/fepegar/torchio/issues/430
- https://github.com/fepegar/torchio/issues/382
- https://github.com/fepegar/torchio/pull/592

Crop

class torchio.transforms.**Crop**(*cropping: int | Tuple[int, int, int] | Tuple[int, int, int, int, int, int] | None*, ***kwargs*)

Bases: `BoundsTransform`

Crop an image.

Parameters

- **cropping** – Tuple $(w_{ini}, w_{fin}, h_{ini}, h_{fin}, d_{ini}, d_{fin})$ defining the number of values cropped from the edges of each axis. If the initial shape of the image is $W \times H \times D$, the final shape will be $(-w_{ini}+W-w_{fin}) \times (-h_{ini}+H-h_{fin}) \times (-d_{ini}+D-d_{fin})$. If only three values (w, h, d) are provided, then $w_{ini} = w_{fin} = w$, $h_{ini} = h_{fin} = h$ and $d_{ini} = d_{fin} = d$. If only one value n is provided, then $w_{ini} = w_{fin} = h_{ini} = h_{fin} = d_{ini} = d_{fin} = n$.
- ****kwargs** – See *Transform* for additional keyword arguments.

See also:

If you want to pass the output shape instead, please use *CropOrPad* instead.

Pad

class torchio.transforms.**Pad**(*padding: int | Tuple[int, int, int] | Tuple[int, int, int, int, int, int] | None*, *padding_mode: str | float = 0*, ***kwargs*)

Bases: BoundsTransform

Pad an image.

Parameters

- **padding** – Tuple $(w_{ini}, w_{fin}, h_{ini}, h_{fin}, d_{ini}, d_{fin})$ defining the number of values padded to the edges of each axis. If the initial shape of the image is $W \times H \times D$, the final shape will be $(w_{ini} + W + w_{fin}) \times (h_{ini} + H + h_{fin}) \times (d_{ini} + D + d_{fin})$. If only three values (w, h, d) are provided, then $w_{ini} = w_{fin} = w$, $h_{ini} = h_{fin} = h$ and $d_{ini} = d_{fin} = d$. If only one value n is provided, then $w_{ini} = w_{fin} = h_{ini} = h_{fin} = d_{ini} = d_{fin} = n$.
- **padding_mode** – See possible modes in NumPy docs. If it is a number, the mode will be set to `'constant'`.
- ****kwargs** – See *Transform* for additional keyword arguments.

See also:

If you want to pass the output shape instead, please use *CropOrPad* instead.

Label

RemapLabels

class torchio.transforms.**RemapLabels**(*remapping: Dict[int, int]*, *masking_method: str | Callable[[Tensor], Tensor] | int | Tuple[int, int, int] | Tuple[int, int, int, int, int, int] | None = None*, ***kwargs*)

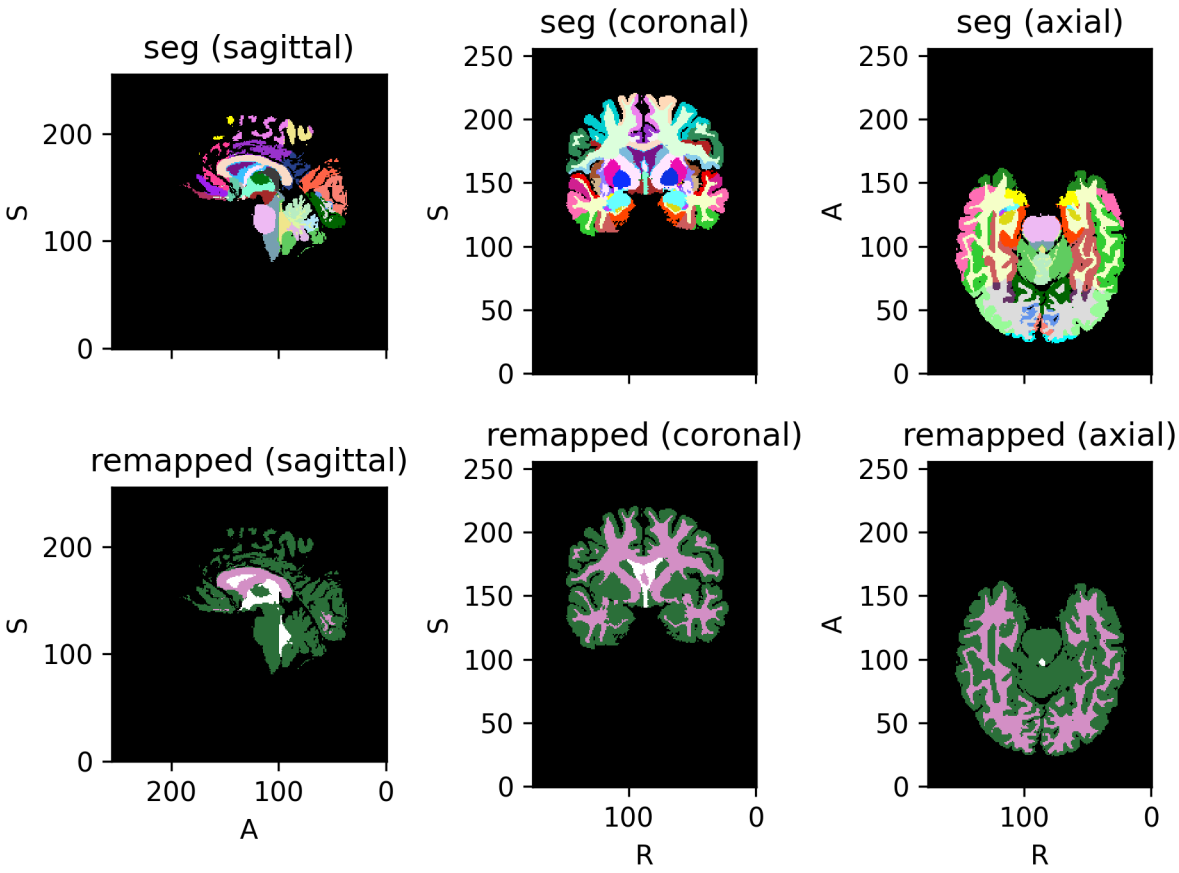
Bases: LabelTransform

Modify labels in a label map.

Masking can be used to split the label into two during the inverse transformation.

Parameters

- **remapping** – Dictionary that specifies how labels should be remapped. The keys are the old labels, and the corresponding values replace them.
- **masking_method** – Defines a mask for where the label remapping is applied. It can be one of:
  - `None`: the mask image is all ones, i.e. all values in the image are used.
  - A string: key to a *torchio.LabelMap* in the subject which is used as a mask, OR an anatomical label: `'Left'`, `'Right'`, `'Anterior'`, `'Posterior'`, `'Inferior'`, `'Superior'` which specifies a half of the mask volume to be ones.
  - A function: the mask image is computed as a function of the intensity image. The function must receive and return a 4D `torch.Tensor`.
- ****kwargs** – See *Transform* for additional keyword arguments.


seg (sagittal)
seg (coronal)
seg (axial)
remapped (sagittal)
remapped (coronal)
remapped (axial)
S
A
R
0
50
100
150
200
250

Example

```
>>> import torch
>>> import torchio as tio
>>> def get_image(*labels):
...     tensor = torch.as_tensor(labels).reshape(1, 1, 1, -1)
...     image = tio.LabelMap(tensor=tensor)
...     return image
...
>>> image = get_image(0, 1, 2, 3, 4)
>>> remapping = {1: 2, 2: 1, 3: 1, 4: 7}
>>> transform = tio.RemapLabels(remapping)
>>> transform(image).data
tensor([[[[0, 2, 1, 1, 7]]]])
```

Warning: The transform will not be correctly inverted if one of the values in `remapping` is also in the input image:

```
>>> tensor = torch.as_tensor([0, 1]).reshape(1, 1, 1, -1)
>>> subject = tio.Subject(label=tio.LabelMap(tensor=tensor))
>>> mapping = {3: 1}  # the value 1 is in the input image
>>> transform = tio.RemapLabels(mapping)
>>> transformed = transform(subject)
>>> back = transformed.apply_inverse_transform()
>>> original_label_set = set(subject.label.data.unique().tolist())
>>> back_label_set = set(back.label.data.unique().tolist())
>>> original_label_set
{0, 1}
>>> back_label_set
{0, 3}
```

Example

```
>>> import torchio as tio
>>> # Target label map has the following labels:
>>> # {
>>> #     'left_ventricle': 1, 'right_ventricle': 2,
>>> #     'left_caudate': 3,   'right_caudate': 4,
>>> #     'left_putamen': 5,   'right_putamen': 6,
>>> #     'left_thalamus': 7,  'right_thalamus': 8,
>>> # }
>>> transform = tio.RemapLabels({2:1, 4:3, 6:5, 8:7})
>>> # Merge right side labels with left side labels
>>> transformed = transform(subject)
>>> # Undesired behavior: The inverse transform will remap ALL left side labels to␣
↪right side labels
>>> # so the label map only has right side labels.
>>> inverse_transformed = transformed.apply_inverse_transform()
>>> # Here's the *right* way to do it with masking:
>>> transform = tio.RemapLabels({2:1, 4:3, 6:5, 8:7}, masking_method="Right")
```

(continues on next page)

(continued from previous page)

```
>>> # Remap the labels on the right side only (no difference yet).
>>> transformed = transform(subject)
>>> # Apply the inverse on the right side only. The labels are correctly split into␣
↪left/right.
>>> inverse_transformed = transformed.apply_inverse_transform()
```

RemoveLabels

class torchio.transforms.**RemoveLabels**(*labels: Sequence[int]*, *background_label: int = 0*, *masking_method: str | Callable[[Tensor], Tensor] | int | Tuple[int, int, int] | Tuple[int, int, int, int, int, int] | None = None*, ***kwargs*)

Bases: *RemapLabels*

Remove labels from a label map.

The removed labels are remapped to the background label.

This transformation is not invertible.

Parameters

- **labels** – A sequence of label integers that will be removed.
- **background_label** – integer that specifies which label is considered to be background (typically, `0`).
- **masking_method** – See *RemapLabels*.
- ****kwargs** – See *Transform* for additional keyword arguments.

SequentialLabels

class torchio.transforms.**SequentialLabels**(*masking_method: str | Callable[[Tensor], Tensor] | int | Tuple[int, int, int] | Tuple[int, int, int, int, int, int] | None = None*, ***kwargs*)

Bases: `LabelTransform`

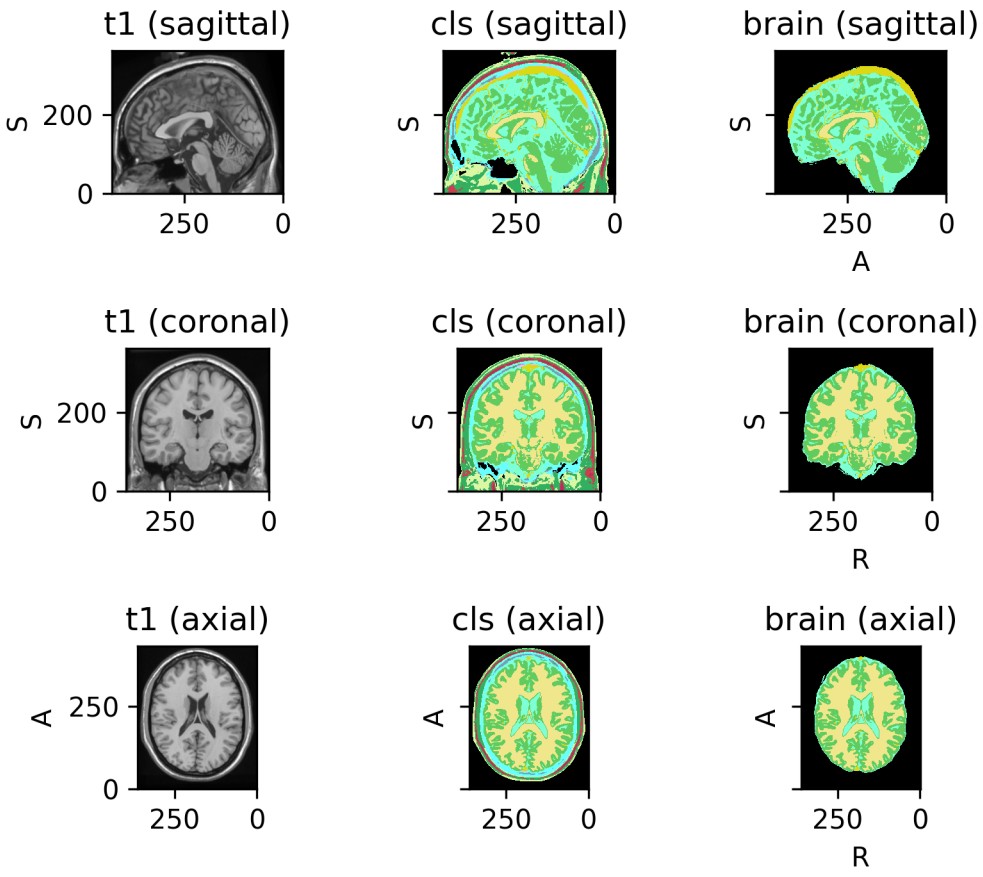
Remap labels in a label map so they become consecutive.

For example, if a label map has labels `(0, 3, 5)`, then this will apply a `RemapLabels` transform with `remapping={3: 1, 5: 2}`, and therefore the output image will have labels `(0, 1, 2)`.

Example

```
>>> import torch
>>> import torchio as tio
>>> def get_image(*labels):
...     tensor = torch.as_tensor(labels).reshape(1, 1, 1, -1)
...     image = tio.LabelMap(tensor=tensor)
...     return image
...
>>> img_with_bg = get_image(0, 5, 10)
>>> transform = tio.SequentialLabels()
```

(continues on next page)


t1 (sagittal)
cls (sagittal)
brain (sagittal)
S
200
0
250
0
A
t1 (coronal)
cls (coronal)
brain (coronal)
R
t1 (axial)
cls (axial)
brain (axial)

(continued from previous page)

```
>>> transform(img_with_bg).data
tensor([[[[0, 1, 2]]]])
>>> img_without_bg = get_image(7, 11, 99)
>>> transform(img_without_bg).data
tensor([[[[0, 1, 2]]]])
```

Note: This transformation is always fully invertible.

Warning: The background is typically represented with the label `0`. There will be zeros in the output image even if they are none in the input.

Parameters

- **masking_method** – See *RemapLabels*.
- ****kwargs** – See *Transform* for additional keyword arguments.

OneHot

class torchio.transforms.**OneHot**(*num_classes: int = -1*, ***kwargs*)

Bases: LabelTransform

Reencode label maps using one-hot encoding.

Parameters

- **num_classes** – See one_hot().
- ****kwargs** – See *Transform* for additional keyword arguments.

Contour

class torchio.transforms.**Contour**(*p: float = 1*, *copy: bool = True*, *include: Sequence[str] | None = None*, *exclude: Sequence[str] | None = None*, *keys: Sequence[str] | None = None*, *keep: Dict[str, str] | None = None*, *parse_input: bool = True*, *label_keys: Sequence[str] | None = None*)

Bases: LabelTransform

Keep only the borders of each connected component in a binary image.

Parameters

****kwargs** – See *Transform* for additional keyword arguments.

KeepLargestComponent

class torchio.transforms.**KeepLargestComponent**(*p: float = 1*, *copy: bool = True*, *include: Sequence[str] | None = None*, *exclude: Sequence[str] | None = None*, *keys: Sequence[str] | None = None*, *keep: Dict[str, str] | None = None*, *parse_input: bool = True*, *label_keys: Sequence[str] | None = None*)

Bases: LabelTransform

Keep only the largest connected component in a binary label map.

Parameters

****kwargs** – See *Transform* for additional keyword arguments.

Note: For now, this transform only works for binary images, i.e., label maps with a background and a foreground class. If you are interested in extending this transform, please open a new issue.

Augmentation

Augmentation transforms generate different results every time they are called.

Base class

RandomTransform

class torchio.transforms.augmentation.**RandomTransform**(***kwargs*)

Bases: *Transform*

Base class for stochastic augmentation transforms.

Parameters

****kwargs** – See *Transform* for additional keyword arguments.

Composition

Compose

class torchio.transforms.**Compose**(*transforms: Sequence[*Transform*]*, ***kwargs*)

Bases: *Transform*

Compose several transforms together.

Parameters

- **transforms** – Sequence of instances of *Transform*.
- ****kwargs** – See *Transform* for additional keyword arguments.

OneOf

class torchio.transforms.**OneOf**(*transforms: Dict[Transform, float] | Sequence[Transform]*, ***kwargs*)

Bases: *RandomTransform*

Apply only one of the given transforms.

Parameters

- **transforms** – Dictionary with instances of *Transform* as keys and probabilities as values. Probabilities are normalized so they sum to one. If a sequence is given, the same probability will be assigned to each transform.
- ****kwargs** – See *Transform* for additional keyword arguments.

Example

```
>>> import torchio as tio
>>> colin = tio.datasets.Colin27()
>>> transforms_dict = {
...     tio.RandomAffine(): 0.75,
...     tio.RandomElasticDeformation(): 0.25,
... }  # Using 3 and 1 as probabilities would have the same effect
>>> transform = tio.OneOf(transforms_dict)
>>> transformed = transform(colin)
```

Spatial

RandomFlip

class torchio.transforms.**RandomFlip**(*axes: int | Tuple[int, ...] = 0*, *flip_probability: float = 0.5*, ***kwargs*)

Bases: *RandomTransform*, SpatialTransform

Reverse the order of elements in an image along the given axes.

Parameters

- **axes** – Index or tuple of indices of the spatial dimensions along which the image might be flipped. If they are integers, they must be in `(0, 1, 2)`. Anatomical labels may also be used, such as `'Left'`, `'Right'`, `'Anterior'`, `'Posterior'`, `'Inferior'`, `'Superior'`, `'Height'` and `'Width'`, `'AP'` (antero-posterior), `'lr'` (lateral), `'w'` (width) or `'i'` (inferior). Only the first letter of the string will be used. If the image is 2D, `'Height'` and `'Width'` may be used.
- **flip_probability** – Probability that the image will be flipped. This is computed on a per-axis basis.
- ****kwargs** – See *Transform* for additional keyword arguments.

Example

```
>>> import torchio as tio
>>> fpg = tio.datasets.FPG()
>>> flip = tio.RandomFlip(axes=('LR',))  # flip along lateral axis only
```

Tip: It is handy to specify the axes as anatomical labels when the image orientation is not known.

RandomAffine

class torchio.transforms.**RandomAffine**(*scales: float | Tuple[float, float] | Tuple[float, float, float] | Tuple[float, float, float, float, float, float] = 0.1*, *degrees: float | Tuple[float, float] | Tuple[float, float, float] | Tuple[float, float, float, float, float, float] = 10*, *translation: float | Tuple[float, float] | Tuple[float, float, float] | Tuple[float, float, float, float, float, float] = 0*, *isotropic: bool = False*, *center: str = 'image'*, *default_pad_value: str | float = 'minimum'*, *image_interpolation: str = 'linear'*, *label_interpolation: str = 'nearest'*, *check_shape: bool = True*, ***kwargs*)

Bases: *RandomTransform*, SpatialTransform

Apply a random affine transformation and resample the image.

Parameters

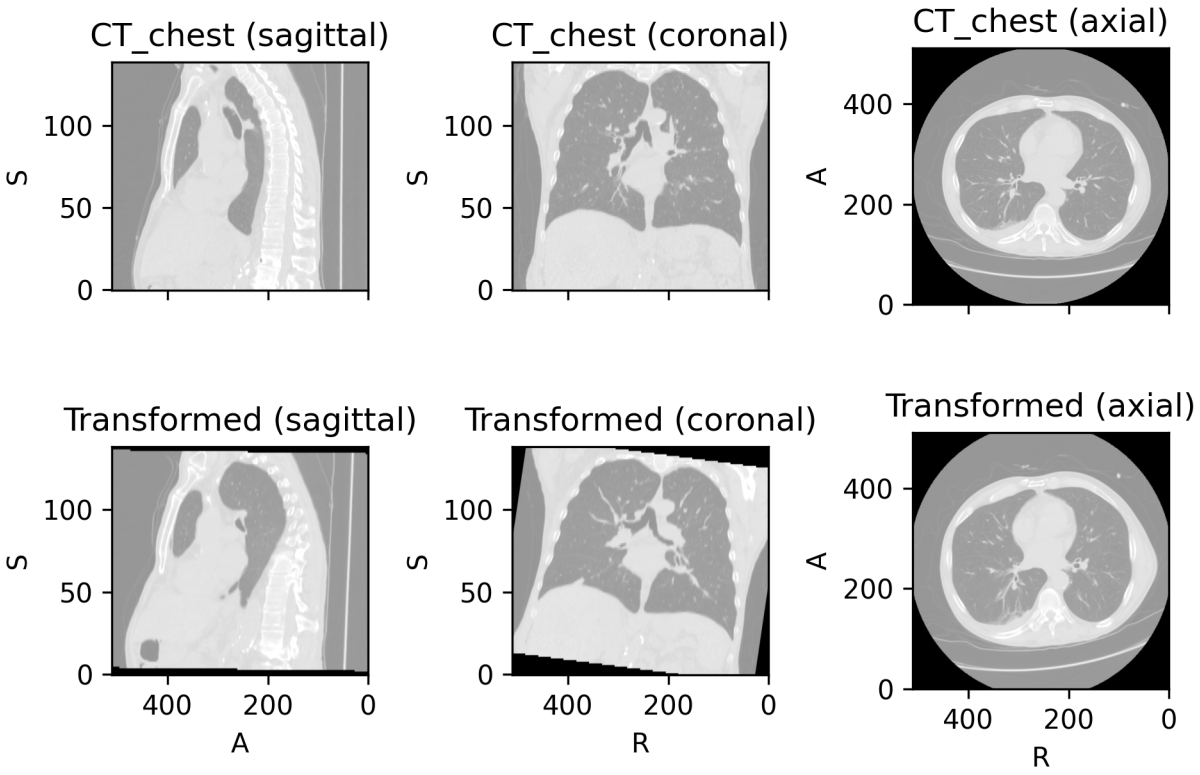
- **scales** – Tuple $(a_1, b_1, a_2, b_2, a_3, b_3)$ defining the scaling ranges. The scaling values along each dimension are (s_1, s_2, s_3), where $s_i \sim \mathcal{U}(a_i, b_i)$. If two values (a, b) are provided, then $s_i \sim \mathcal{U}(a, b)$. If only one value x is provided, then $s_i \sim \mathcal{U}(1 - x, 1 + x)$. If three values (x_1, x_2, x_3) are provided, then $s_i \sim \mathcal{U}(1 - x_i, 1 + x_i)$. For example, using `scales=(0.5, 0.5)` will zoom out the image, making the objects inside look twice as small while preserving the physical size and position of the image bounds.
- **degrees** – Tuple $(a_1, b_1, a_2, b_2, a_3, b_3)$ defining the rotation ranges in degrees. Rotation angles around each axis are $(\theta_1, \theta_2, \theta_3)$, where $\theta_i \sim \mathcal{U}(a_i, b_i)$. If two values (a, b) are provided, then $\theta_i \sim \mathcal{U}(a, b)$. If only one value x is provided, then $\theta_i \sim \mathcal{U}(-x, x)$. If three values (x_1, x_2, x_3) are provided, then $\theta_i \sim \mathcal{U}(-x_i, x_i)$.
- **translation** – Tuple $(a_1, b_1, a_2, b_2, a_3, b_3)$ defining the translation ranges in mm. Translation along each axis is (t_1, t_2, t_3), where $t_i \sim \mathcal{U}(a_i, b_i)$. If two values (a, b) are provided, then $t_i \sim \mathcal{U}(a, b)$. If only one value x is provided, then $t_i \sim \mathcal{U}(-x, x)$. If three values (x_1, x_2, x_3) are provided, then $t_i \sim \mathcal{U}(-x_i, x_i)$. For example, if the image is in RAS+ orientation (e.g., after applying `ToCanonical`) and the translation is $(10, 20, 30)$, the sample will move 10 mm to the right, 20 mm to the front, and 30 mm upwards. If the image was in, e.g., PIR+ orientation, the sample will move 10 mm to the back, 20 mm downwards, and 30 mm to the right.
- **isotropic** – If `True`, the scaling factor along all dimensions is the same, i.e. $s_1 = s_2 = s_3$.
- **center** – If `'image'`, rotations and scaling will be performed around the image center. If `'origin'`, rotations and scaling will be performed around the origin in world coordinates.
- **default_pad_value** – As the image is rotated, some values near the borders will be undefined. If `'minimum'`, the fill value will be the image minimum. If `'mean'`, the fill value

is the mean of the border values. If `'otsu'`, the fill value is the mean of the values at the border that lie under an Otsu threshold. If it is a number, that value will be used.

- **image_interpolation** – See *Interpolation*.
- **label_interpolation** – See *Interpolation*.
- **check_shape** – If `True` an error will be raised if the images are in different physical spaces. If `False`, `center` should probably not be `'image'` but `'center'`.
- ****kwargs** – See *Transform* for additional keyword arguments.

Example

```
>>> import torchio as tio
>>> image = tio.datasets.Colin27().t1
>>> transform = tio.RandomAffine(
...     scales=(0.9, 1.2),
...     degrees=15,
... )
>>> transformed = transform(image)
```



RandomElasticDeformation

class torchio.transforms.**RandomElasticDeformation**(*num_control_points: int | Tuple[int, int, int] = 7*, *max_displacement: float | Tuple[float, float, float] = 7.5*, *locked_borders: int = 2*, *image_interpolation: str = 'linear'*, *label_interpolation: str = 'nearest'*, ***kwargs*)

Bases: *RandomTransform*, SpatialTransform

Apply dense random elastic deformation.

A random displacement is assigned to a coarse grid of control points around and inside the image. The displacement at each voxel is interpolated from the coarse grid using cubic B-splines.

The 'Deformable Registration' topic on ScienceDirect contains useful articles explaining interpolation of displacement fields using cubic B-splines.

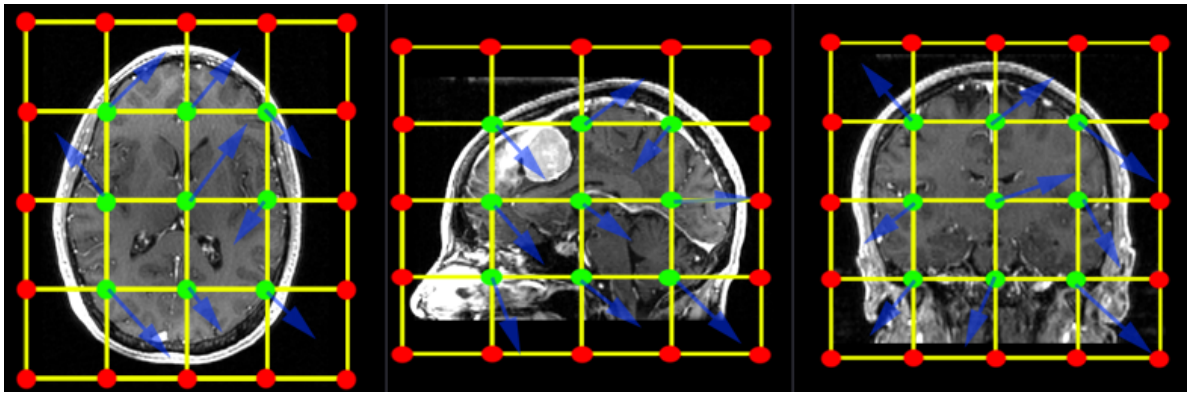
Warning: This transform is slow as it requires expensive computations. If your images are large you might want to use *RandomAffine* instead.

Parameters

- **num_control_points** – Number of control points along each dimension of the coarse grid (n_x, n_y, n_z). If a single value n is passed, then $n_x = n_y = n_z = n$. Smaller numbers generate smoother deformations. The minimum number of control points is `4` as this transform uses cubic B-splines to interpolate displacement.
- **max_displacement** – Maximum displacement along each dimension at each control point (D_x, D_y, D_z). The displacement along dimension i at each control point is $d_i \sim \mathcal{U}(0, D_i)$. If a single value D is passed, then $D_x = D_y = D_z = D$. Note that the total maximum displacement would actually be $D_{max} = \sqrt{D_x^2 + D_y^2 + D_z^2}$.
- **locked_borders** – If `0`, all displacement vectors are kept. If `1`, displacement of control points at the border of the coarse grid will be set to `0`. If `2`, displacement of control points at the border of the image (red dots in the image below) will also be set to `0`.
- **image_interpolation** – See *Interpolation*. Note that this is the interpolation used to compute voxel intensities when resampling using the dense displacement field. The value of the dense displacement at each voxel is always interpolated with cubic B-splines from the values at the control points of the coarse grid.
- **label_interpolation** – See *Interpolation*.
- ****kwargs** – See *Transform* for additional keyword arguments.

This gist can also be used to better understand the meaning of the parameters.

This is an example from the 3D Slicer registration FAQ.



To generate a similar grid of control points with TorchIO, the transform can be instantiated as follows:

```
>>> from torchio import RandomElasticDeformation
>>> transform = RandomElasticDeformation(
...     num_control_points=(7, 7, 7),  # or just 7
...     locked_borders=2,
... )
```

Note that control points outside the image bounds are not showed in the example image (they would also be red as we set `locked_borders` to 2).

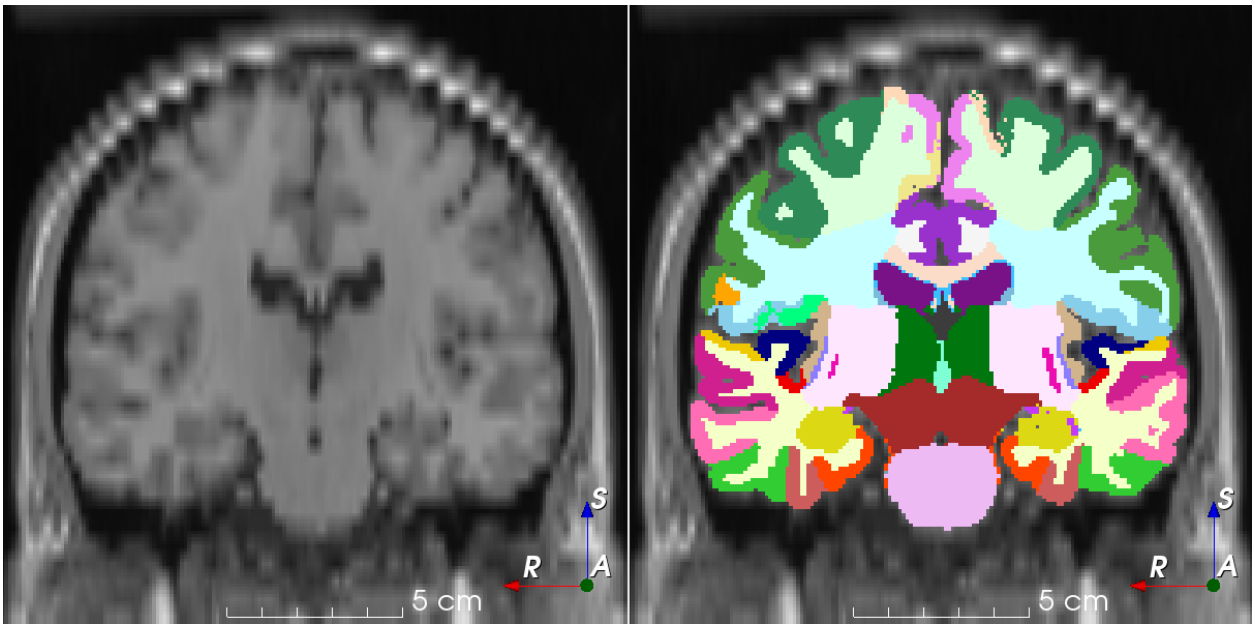
Warning: Image folding may occur if the maximum displacement is larger than half the coarse grid spacing. The grid spacing can be computed using the image bounds in physical space[1] and the number of control points:

```
>>> import numpy as np
>>> import torchio as tio
>>> image = tio.datasets.Slicer().MRHead.as_sitk()
>>> image.GetSize()  # in voxels
(256, 256, 130)
>>> image.GetSpacing()  # in mm
(1.0, 1.0, 1.2999954223632812)
>>> bounds = np.array(image.GetSize()) * np.array(image.GetSpacing())
>>> bounds  # mm
array([256.        , 256.        , 168.99940491])
>>> num_control_points = np.array((7, 7, 6))
>>> grid_spacing = bounds / (num_control_points - 2)
>>> grid_spacing
array([51.2       , 51.2       , 42.24985123])
>>> potential_folding = grid_spacing / 2
>>> potential_folding  # mm
array([25.6       , 25.6       , 21.12492561])
```

Using a `max_displacement` larger than the computed `potential_folding` will raise a `RuntimeWarning`.

[1] Technically, 2ϵ should be added to the image bounds, where $\epsilon = 2^{-3}$ according to ITK source code.

`RandomAnisotropy`



class torchio.transforms.**RandomAnisotropy**(*axes: int | Tuple[int, ...] = (0, 1, 2)*, *downsampling: float | Tuple[float, float] = (1.5, 5)*, *image_interpolation: str = 'linear'*, *scalars_only: bool = True*, ***kwargs*)

Bases: *RandomTransform*

Downsample an image along an axis and upsample to initial space.

This transform simulates an image that has been acquired using anisotropic spacing and resampled back to its original spacing.

Similar to the work by Billot et al.: Partial Volume Segmentation of Brain MRI Scans of any Resolution and Contrast.

Parameters

- **axes** – Axis or tuple of axes along which the image will be downsampled.
- **downsampling** – Downsampling factor $m1$. If a tuple (a, b) is provided then $m \sim \mathcal{U}(a, b)$.
- **image_interpolation** – Image interpolation used to upsample the image back to its initial spacing. Downsampling is performed using nearest neighbor interpolation. See *Interpolation* for supported interpolation types.
- **scalars_only** – Apply only to instances of *torchio.ScalarImage*. This is useful when the segmentation quality needs to be kept, as in Billot et al..
- ****kwargs** – See *Transform* for additional keyword arguments.

Example

```
>>> import torchio as tio
>>> transform = tio.RandomAnisotropy(axes=1, downsampling=2)
>>> transform = tio.RandomAnisotropy(
...     axes=(0, 1, 2),
...     downsampling=(2, 5),
... )   # Multiply spacing of one of the 3 axes by a factor randomly chosen in [2,␣
↪5]
>>> colin = tio.datasets.Colin27()
>>> transformed = transform(colin)
```

Intensity

RandomMotion

class torchio.transforms.**RandomMotion**(*degrees: float | Tuple[float, float] = 10*, *translation: float | Tuple[float, float] = 10*, *num_transforms: int = 2*, *image_interpolation: str = 'linear'*, ***kwargs*)

Bases: *RandomTransform*, IntensityTransform, FourierTransform

Add random MRI motion artifact.

Magnetic resonance images suffer from motion artifacts when the subject moves during image acquisition. This transform follows Shaw et al., 2019 to simulate motion artifacts for data augmentation.

Parameters

- **degrees** – Tuple (a, b) defining the rotation range in degrees of the simulated movements. The rotation angles around each axis are $(\theta_1, \theta_2, \theta_3)$, where $\theta_i \sim \mathcal{U}(a, b)$. If only one value d is provided, $\theta_i \sim \mathcal{U}(-d, d)$. Larger values generate more distorted images.
- **translation** – Tuple (a, b) defining the translation in mm of the simulated movements. The translations along each axis are (t_1, t_2, t_3), where $t_i \sim \mathcal{U}(a, b)$. If only one value t is provided, $t_i \sim \mathcal{U}(-t, t)$. Larger values generate more distorted images.
- **num_transforms** – Number of simulated movements. Larger values generate more distorted images.
- **image_interpolation** – See *Interpolation*.
- ****kwargs** – See *Transform* for additional keyword arguments.

Warning: Large numbers of movements lead to longer execution times for 3D images.

RandomGhosting

class torchio.transforms.**RandomGhosting**(*num_ghosts: int | Tuple[int, int] = (4, 10)*, *axes: int | Tuple[int, ...] = (0, 1, 2)*, *intensity: float | Tuple[float, float] = (0.5, 1)*, *restore: float = 0.02*, ***kwargs*)

Bases: *RandomTransform*, IntensityTransform

Add random MRI ghosting artifact.

Discrete "ghost" artifacts may occur along the phase-encode direction whenever the position or signal intensity of imaged structures within the field-of-view vary or move in a regular (periodic) fashion. Pulsatile flow of blood or CSF, cardiac motion, and respiratory motion are the most important patient-related causes of ghost artifacts in clinical MR imaging (from mriquestions.com).

Parameters

- **num_ghosts** – Number of 'ghosts' n in the image. If `num_ghosts` is a tuple (a, b), then $n \sim \mathcal{U}(a, b) \cap \mathbb{N}$. If only one value d is provided, $n \sim \mathcal{U}(0, d) \cap \mathbb{N}$.
- **axes** – Axis along which the ghosts will be created. If `axes` is a tuple, the axis will be randomly chosen from the passed values. Anatomical labels may also be used (see `RandomFlip`).
- **intensity** – Positive number representing the artifact strength s with respect to the maximum of the k-space. If `0`, the ghosts will not be visible. If a tuple (a, b) is provided then $s \sim \mathcal{U}(a, b)$. If only one value d is provided, $s \sim \mathcal{U}(0, d)$.
- **restore** – Number between `0` and `1` indicating how much of the k-space center should be restored after removing the planes that generate the artifact.
- ****kwargs** – See *Transform* for additional keyword arguments.

Note: The execution time of this transform does not depend on the number of ghosts.

RandomSpike

class torchio.transforms.**RandomSpike**(*num_spikes: int | Tuple[int, int] = 1*, *intensity: float | Tuple[float, float] = (1, 3)*, ***kwargs*)

Bases: *RandomTransform*, IntensityTransform, FourierTransform

Add random MRI spike artifacts.

Also known as Herringbone artifact, crisscross artifact or corduroy artifact, it creates stripes in different directions in image space due to spikes in k-space.

Parameters

- **num_spikes** – Number of spikes n present in k-space. If a tuple (a, b) is provided, then $n \sim \mathcal{U}(a, b) \cap \mathbb{N}$. If only one value d is provided, $n \sim \mathcal{U}(0, d) \cap \mathbb{N}$. Larger values generate more distorted images.
- **intensity** – Ratio r between the spike intensity and the maximum of the spectrum. If a tuple (a, b) is provided, then $r \sim \mathcal{U}(a, b)$. If only one value d is provided, $r \sim \mathcal{U}(-d, d)$. Larger values generate more distorted images.

- ****kwargs** – See *Transform* for additional keyword arguments.

Note: The execution time of this transform does not depend on the number of spikes.

RandomBiasField

class torchio.transforms.**RandomBiasField**(*coefficients: float | Tuple[float, float] = 0.5*, *order: int = 3*, ***kwargs*)

Bases: *RandomTransform*, IntensityTransform

Add random MRI bias field artifact.

MRI magnetic field inhomogeneity creates intensity variations of very low frequency across the whole image.

The bias field is modeled as a linear combination of polynomial basis functions, as in K. Van Leemput et al., 1999, *Automated model-based tissue classification of MR images of the brain*.

It was implemented in NiftyNet by Carole Sudre and used in Sudre et al., 2017, Longitudinal segmentation of age-related white matter hyperintensities.

Parameters

- **coefficients** – Maximum magnitude n of polynomial coefficients. If a tuple (a, b) is specified, then $n \sim \mathcal{U}(a, b)$.
- **order** – Order of the basis polynomial functions.
- ****kwargs** – See *Transform* for additional keyword arguments.

RandomBlur

class torchio.transforms.**RandomBlur**(*std: float | Tuple[float, float] = (0, 2)*, ***kwargs*)

Bases: *RandomTransform*, IntensityTransform

Blur an image using a random-sized Gaussian filter.

Parameters

- **std** – Tuple $(a_1, b_1, a_2, b_2, a_3, b_3)$ representing the ranges (in mm) of the standard deviations $(\sigma_1, \sigma_2, \sigma_3)$ of the Gaussian kernels used to blur the image along each axis, where $\sigma_i \sim \mathcal{U}(a_i, b_i)$. If two values (a, b) are provided, then $\sigma_i \sim \mathcal{U}(a, b)$. If only one value x is provided, then $\sigma_i \sim \mathcal{U}(0, x)$. If three values (x_1, x_2, x_3) are provided, then $\sigma_i \sim \mathcal{U}(0, x_i)$.
- ****kwargs** – See *Transform* for additional keyword arguments.

RandomNoise

class torchio.transforms.**RandomNoise**(*mean: float | Tuple[float, float] = 0*, *std: float | Tuple[float, float] = (0, 0.25)*, ***kwargs*)

Bases: *RandomTransform*, IntensityTransform

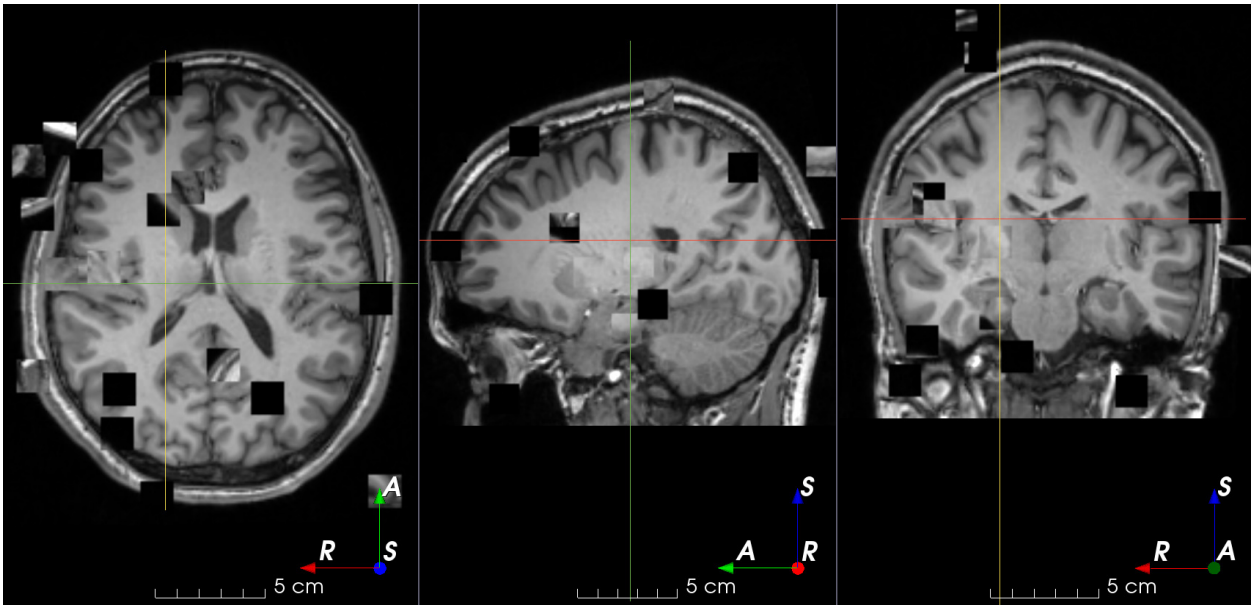
Add Gaussian noise with random parameters.

Add noise sampled from a normal distribution with random parameters.

Parameters

- **mean** – Mean μ of the Gaussian distribution from which the noise is sampled. If two values (a, b) are provided, then $\mu \sim \mathcal{U}(a, b)$. If only one value d is provided, $\mu \sim \mathcal{U}(-d, d)$.
- **std** – Standard deviation σ of the Gaussian distribution from which the noise is sampled. If two values (a, b) are provided, then $\sigma \sim \mathcal{U}(a, b)$. If only one value d is provided, $\sigma \sim \mathcal{U}(0, d)$.
- ****kwargs** – See *Transform* for additional keyword arguments.

RandomSwap



class torchio.transforms.**RandomSwap**(*patch_size: int | Tuple[int, int, int] = 15*, *num_iterations: int = 100*, ***kwargs*)

Bases: *RandomTransform*, IntensityTransform

Randomly swap patches within an image.

This is typically used in context restoration for self-supervised learning.

Parameters

- **patch_size** – Tuple of integers (w, h, d) to swap patches of size $w \times h \times d$. If a single number n is provided, $w = h = d = n$.

- **num_iterations** – Number of times that two patches will be swapped.
- ****kwargs** – See *Transform* for additional keyword arguments.

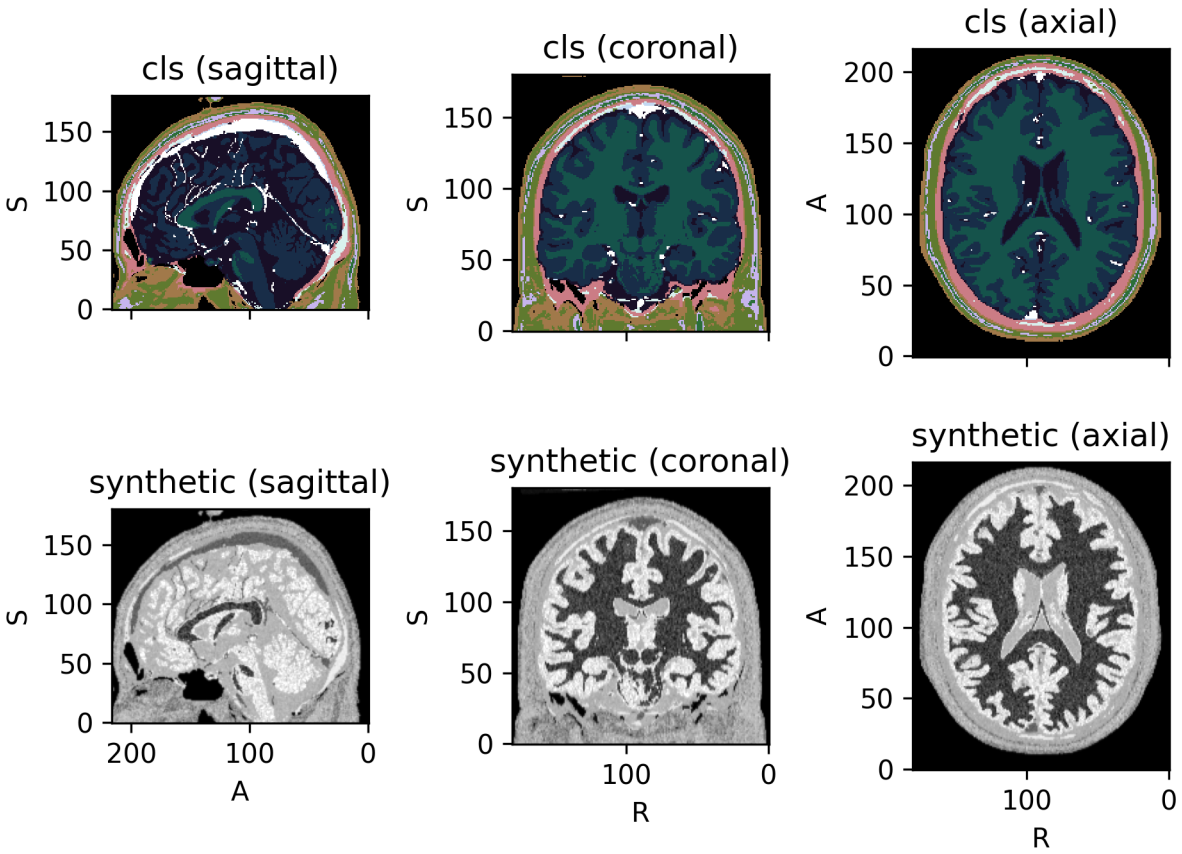
RandomLabelsToImage

class torchio.transforms.**RandomLabelsToImage**(*label_key: str | None = None*, *used_labels: Sequence[int] | None = None*, *image_key: str = 'image_from_labels'*, *mean: Sequence[float | Tuple[float, float]] | None = None*, *std: Sequence[float | Tuple[float, float]] | None = None*, *default_mean: float | Tuple[float, float] = (0.1, 0.9)*, *default_std: float | Tuple[float, float] = (0.01, 0.1)*, *discretize: bool = False*, *ignore_background: bool = False*, ***kwargs*)

Bases: *RandomTransform*, IntensityTransform

Randomly generate an image from a segmentation.

Based on the work by Billot et al.: A Learning Strategy for Contrast-agnostic MRI Segmentation and Partial Volume Segmentation of Brain MRI Scans of any Resolution and Contrast.



Parameters

- **label_key** – String designating the label map in the subject that will be used to generate the new image.

- **used_labels** – Sequence of integers designating the labels used to generate the new image. If categorical encoding is used, `label_channels` refers to the values of the categorical encoding. If one hot encoding or partial-volume label maps are used, `label_channels` refers to the channels of the label maps. Default uses all labels. Missing voxels will be filled with zero or with voxels from an already existing volume, see `image_key`.
- **image_key** – String designating the key to which the new volume will be saved. If this key corresponds to an already existing volume, missing voxels will be filled with the corresponding values in the original volume.
- **mean** – Sequence of means for each label. For each value v, if a tuple (a, b) is provided then $v \sim \mathcal{U}(a, b)$. If `None`, `default_mean` range will be used for every label. If not `None` and `label_channels` is not `None`, `mean` and `label_channels` must have the same length.
- **std** – Sequence of standard deviations for each label. For each value v, if a tuple (a, b) is provided then $v \sim \mathcal{U}(a, b)$. If `None`, `default_std` range will be used for every label. If not `None` and `label_channels` is not `None`, `std` and `label_channels` must have the same length.
- **default_mean** – Default mean range.
- **default_std** – Default standard deviation range.
- **discretize** – If `True`, partial-volume label maps will be discretized. Does not have any effects if not using partial-volume label maps. Discretization is done taking the class of the highest value per voxel in the different partial-volume label maps using `torch.argmax()` on the channel dimension (i.e. 0).
- **ignore_background** – If `True`, input voxels labeled as `0` will not be modified.
- ****kwargs** – See *Transform* for additional keyword arguments.

Tip: It is recommended to blur the new images in order to simulate partial volume effects at the borders of the synthetic structures. See *RandomBlur*.

Example

```
>>> import torchio as tio
>>> subject = tio.datasets.ICBM2009CNonlinearSymmetric()
>>> # Using the default parameters
>>> transform = tio.RandomLabelsToImage(label_key='tissues')
>>> # Using custom mean and std
>>> transform = tio.RandomLabelsToImage(
...     label_key='tissues', mean=[0.33, 0.66, 1.], std=[0, 0, 0]
... )
>>> # Discretizing the partial volume maps and blurring the result
>>> simulation_transform = tio.RandomLabelsToImage(
...     label_key='tissues', mean=[0.33, 0.66, 1.], std=[0, 0, 0], discretize=True
... )
>>> blurring_transform = tio.RandomBlur(std=0.3)
>>> transform = tio.Compose([simulation_transform, blurring_transform])
>>> transformed = transform(subject)  # subject has a new key 'image_from_labels'␣
↪with the simulated image
>>> # Filling holes of the simulated image with the original T1 image
```

(continues on next page)

(continued from previous page)

```
>>> rescale_transform = tio.RescaleIntensity(
...     out_min_max=(0, 1), percentiles=(1, 99))   # Rescale intensity before␣
↪filling holes
>>> simulation_transform = tio.RandomLabelsToImage(
...     label_key='tissues',
...     image_key='t1',
...     used_labels=[0, 1]
... )
>>> transform = tio.Compose([rescale_transform, simulation_transform])
>>> transformed = transform(subject)  # subject's key 't1' has been replaced with the␣
↪simulated image
```

See also:

RemapLabels.

RandomGamma

class torchio.transforms.**RandomGamma**(*log_gamma: float | Tuple[float, float] = (-0.3, 0.3)*, ***kwargs*)

Bases: *RandomTransform*, IntensityTransform

Randomly change contrast of an image by raising its values to the power γ.

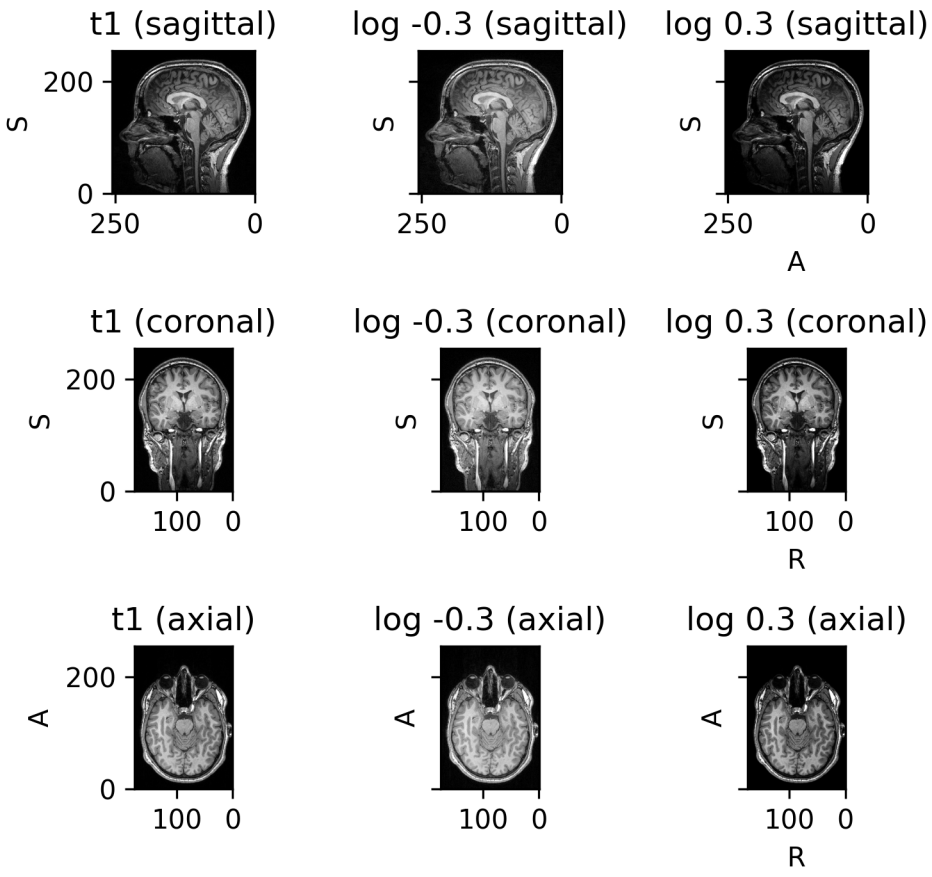
Parameters

- **log_gamma** – Tuple (a, b) to compute the exponent $\gamma = e^{\beta}$, where $\beta \sim \mathcal{U}(a, b)$. If a single value d is provided, then $\beta \sim \mathcal{U}(-d, d)$. Negative and positive values for this argument perform gamma compression and expansion, respectively. See the Gamma correction Wikipedia entry for more information.
- ****kwargs** – See *Transform* for additional keyword arguments.

Note: Fractional exponentiation of negative values is generally not well-defined for non-complex numbers. If negative values are found in the input image I, the applied transform is $\text{sign}(I)|I|^{\gamma}$, instead of the usual I^{γ}. The *RescaleIntensity* transform may be used to ensure that all values are positive. This is generally not problematic, but it is recommended to visualize results on images with negative values. More information can be found on this StackExchange question.

Example

```
>>> import torchio as tio
>>> subject = tio.datasets.FPG()
>>> transform = tio.RandomGamma(log_gamma=(-0.3, 0.3))  # gamma between 0.74 and 1.
↪34
>>> transformed = transform(subject)
```


t1 (sagittal)
log -0.3 (sagittal)
log 0.3 (sagittal)
S
200
0
250
0
A
t1 (coronal)
log -0.3 (coronal)
log 0.3 (coronal)
S
200
0
100
0
R
t1 (axial)
log -0.3 (axial)
log 0.3 (axial)
A
200
0
100
0
R

Others

Lambda

class torchio.transforms.**Lambda**(*function: Callable[[Tensor], Tensor]*, *types_to_apply: Sequence[str] | None = None*, ***kwargs*)

Bases: *Transform*

Applies a user-defined function as transform.

Parameters

- **function** – Callable that receives and returns a 4D `torch.Tensor`.
- **types_to_apply** – List of strings corresponding to the image types to which this transform should be applied. If `None`, the transform will be applied to all images in the subject.
- ****kwargs** – See *Transform* for additional keyword arguments.

Example

```
>>> import torchio as tio
>>> invert_intensity = tio.Lambda(lambda x: -x, types_to_apply=[tio.INTENSITY])
>>> invert_mask = tio.Lambda(lambda x: 1 - x, types_to_apply=[tio.LABEL])
>>> def double(x):
...     return 2 * x
>>> double_transform = tio.Lambda(double)
```

1.5 Medical image datasets

TorchIO offers tools to easily download publicly available datasets from different institutions and modalities.

The interface is similar to `torchvision.datasets`.

If you use any of them, please visit the corresponding website (linked in each description) and make sure you comply with any data usage agreement and you acknowledge the corresponding authors' publications.

If you would like to add a dataset here, please open a discussion on the GitHub repository:

1.5.1 IXI

The Information eXtraction from Images (IXI) dataset contains "nearly 600 MR images from normal, healthy subjects", including "T1, T2 and PD-weighted images, MRA images and Diffusion-weighted images (15 directions)".

Note: This data is made available under the Creative Commons CC BY-SA 3.0 license. If you use it please acknowledge the source of the IXI data, e.g. the IXI website.

IXI

class torchio.datasets.ixi.**IXI**(*root: str | Path, transform: Transform | None = None, download: bool = False, modalities: Sequence[str] = ('T1', 'T2'), **kwargs*)

Bases: *SubjectsDataset*

Full IXI dataset.

Parameters

- **root** – Root directory to which the dataset will be downloaded.
- **transform** – An instance of *Transform*.
- **download** – If set to `True`, will download the data into `root`.
- **modalities** – List of modalities to be downloaded. They must be in `('T1', 'T2', 'PD', 'MRA', 'DTI')`.

Warning: The size of this dataset is multiple GB. If you set `download` to `True`, it will take some time to be downloaded if it is not already present.

Example

```
>>> import torchio as tio
>>> transforms = [
...     tio.ToCanonical(),  # to RAS
...     tio.Resample((1, 1, 1)),  # to 1 mm iso
... ]
>>> ixi_dataset = tio.datasets.IXI(
...     'path/to/ixi_root/',
...     modalities=('T1', 'T2'),
...     transform=tio.Compose(transforms),
...     download=True,
... )
>>> print('Number of subjects in dataset:', len(ixi_dataset))  # 577
>>> sample_subject = ixi_dataset[0]
>>> print('Keys in subject:', tuple(sample_subject.keys()))  # ('T1', 'T2')
>>> print('Shape of T1 data:', sample_subject['T1'].shape)  # [1, 180, 268, 268]
>>> print('Shape of T2 data:', sample_subject['T2'].shape)  # [1, 241, 257, 188]
```

IXITiny

class torchio.datasets.ixi.**IXITiny**(*root: str | Path, transform: Transform | None = None, download: bool = False, **kwargs*)

Bases: *SubjectsDataset*

This is the dataset used in the main notebook. It is a tiny version of IXI, containing 566 T_1-weighted brain MR images and their corresponding brain segmentations, all with size $83 \times 44 \times 55$.

It can be used as a medical image MNIST.

Parameters

- **root** – Root directory to which the dataset will be downloaded.

- **transform** – An instance of *Transform*.
- **download** – If set to `True`, will download the data into `root`.

1.5.2 EPISURG

EPISURG

class torchio.datasets.episurg.**EPISURG**(*root: str | Path, transform:* Transform | *None = None, download: bool = False, **kwargs*)

Bases: *SubjectsDataset*

EPISURG is a clinical dataset of T_1-weighted MRI from 430 epileptic patients who underwent resective brain surgery at the National Hospital of Neurology and Neurosurgery (Queen Square, London, United Kingdom) between 1990 and 2018.

The dataset comprises 430 postoperative MRI. The corresponding preoperative MRI is present for 268 subjects.

Three human raters segmented the resection cavity on partially overlapping subsets of EPISURG.

If you use this dataset for your research, you agree with the *Data use agreement* presented at the EPISURG entry on the UCL Research Data Repository and you must cite the corresponding publications.

Parameters

- **root** – Root directory to which the dataset will be downloaded.
- **transform** – An instance of *Transform*.
- **download** – If set to `True`, will download the data into `root`.

Warning: The size of this dataset is multiple GB. If you set `download` to `True`, it will take some time to be downloaded if it is not already present.

get_labeled() → *SubjectsDataset*

Get dataset from subjects with manual annotations.

get_paired() → *SubjectsDataset*

Get dataset from subjects with pre- and post-op MRI.

get_unlabeled() → *SubjectsDataset*

Get dataset from subjects without manual annotations.

1.5.3 Kaggle datasets

RSNAMICCAI

class torchio.datasets.rsna_miccai.**RSNAMICCAI**(*root_dir: str | Path, train: bool = True, ignore_empty: bool = True, modalities: Sequence[str] = ('T1w', 'T1wCE', 'T2w', 'FLAIR'), **kwargs*)

Bases: *SubjectsDataset*

RSNA-MICCAI Brain Tumor Radiogenomic Classification challenge dataset.

This is a helper class for the dataset used in the RSNA-MICCAI Brain Tumor Radiogenomic Classification challenge hosted on kaggle. The dataset must be downloaded before instantiating this class (as opposed to, e.g., `torchio.datasets.IXI`).

This kaggle kernel includes a usage example including preprocessing of all the scans.

If you reference or use the dataset in any form, include the following citation:

U.Baid, et al., "The RSNA-ASNR-MICCAI BraTS 2021 Benchmark on Brain Tumor Segmentation and Radiogenomic Classification", arXiv:2107.02314, 2021.

Parameters

- **root_dir** – Directory containing the dataset (`train` directory, `test` directory, etc.).
- **train** – If `True`, the `train` set will be used. Otherwise the `test` set will be used.
- **ignore_empty** – If `True`, the three subjects flagged as "presenting issues" (empty images) by the challenge organizers will be ignored. The subject IDs are `00109`, `00123` and `00709`.

Example

```
>>> import torchio as tio
>>> from subprocess import call
>>> call('kaggle competitions download -c rsna-miccai-brain-tumor-radiogenomic-
↪classification'.split())
>>> root_dir = 'rsna-miccai-brain-tumor-radiogenomic-classification'
>>> train_set = tio.datasets.RSNAMICCAI(root_dir, train=True)
>>> test_set = tio.datasets.RSNAMICCAI(root_dir, train=False)
>>> len(train_set), len(test_set)
(582, 87)
```

RSNACervicalSpineFracture

class torchio.datasets.rsna_spine_fracture.**RSNACervicalSpineFracture**(*root_dir: str | Path*, *add_segmentations: bool = False*, *add_bounding_boxes: bool = False*, ***kwargs*)

Bases: *SubjectsDataset*

RSNA 2022 Cervical Spine Fracture Detection dataset.

This is a helper class for the dataset used in the RSNA 2022 Cervical Spine Fracture Detection hosted on kaggle. The dataset must be downloaded before instantiating this class.

1.5.4 MNI

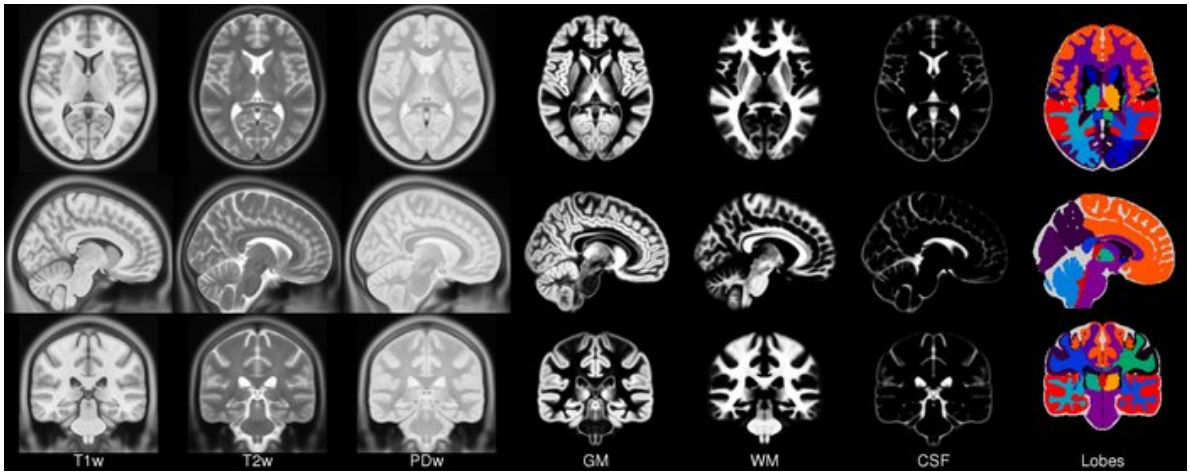
ICBM2009CNonlinearSymmetric

class torchio.datasets.mni.**ICBM2009CNonlinearSymmetric**(*load_4d_tissues: bool = True*)

Bases: SubjectMNI

ICBM template.

More information can be found in the website.



Parameters

load_4d_tissues – If True, the tissue probability maps will be loaded together into a 4D image. Otherwise, they will be loaded into independent images.

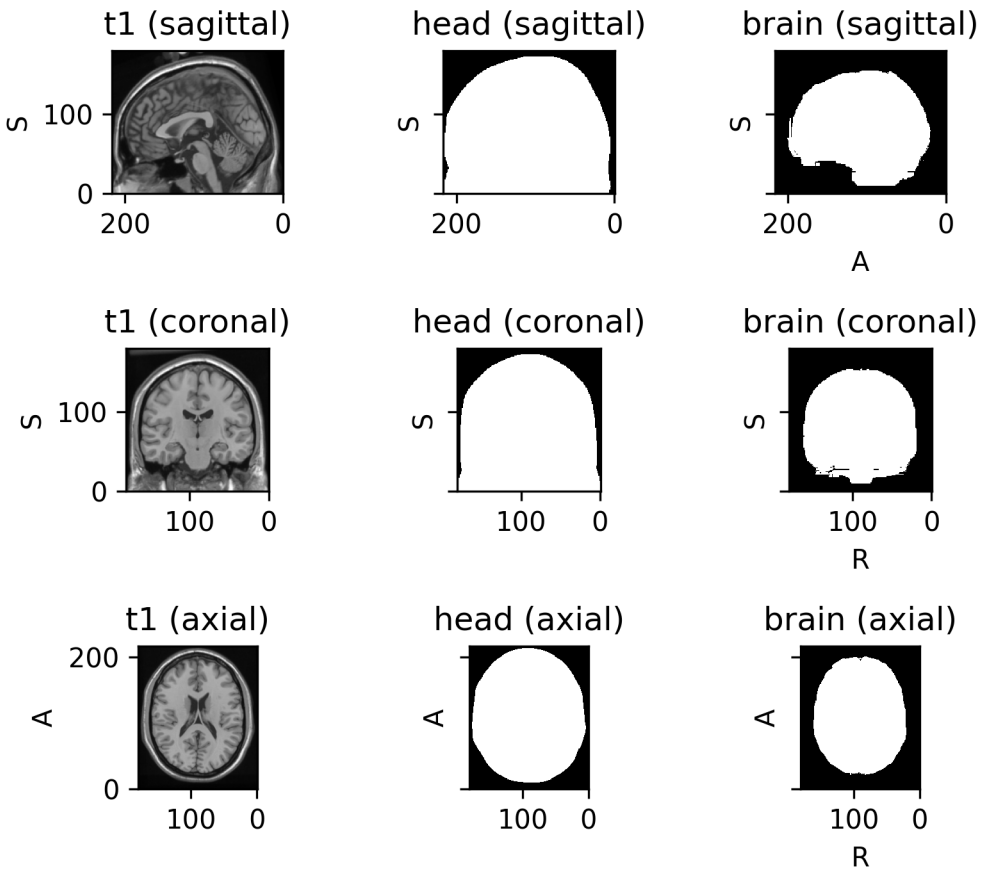
Example

```
>>> import torchio as tio
>>> icbm = tio.datasets.ICBM2009CNonlinearSymmetric()
>>> icbm
ICBM2009CNonlinearSymmetric(Keys: ('t1', 'eyes', 'face', 'brain', 't2', 'pd',
↪'tissues'); images: 7)
>>> icbm = tio.datasets.ICBM2009CNonlinearSymmetric(load_4d_tissues=False)
>>> icbm
ICBM2009CNonlinearSymmetric(Keys: ('t1', 'eyes', 'face', 'brain', 't2', 'pd', 'gm',
↪'wm', 'csf'); images: 9)
```

Colin27

class torchio.datasets.mni.**Colin27**(*version=1998*)

Bases: SubjectMNI


t1 (sagittal)
head (sagittal)
brain (sagittal)
S
100
0
200
0
200
0
200
0
A
t1 (coronal)
head (coronal)
brain (coronal)
S
100
0
100
0
100
0
100
0
R
t1 (axial)
head (axial)
brain (axial)
A
200
0
100
0
100
0
100
0
R

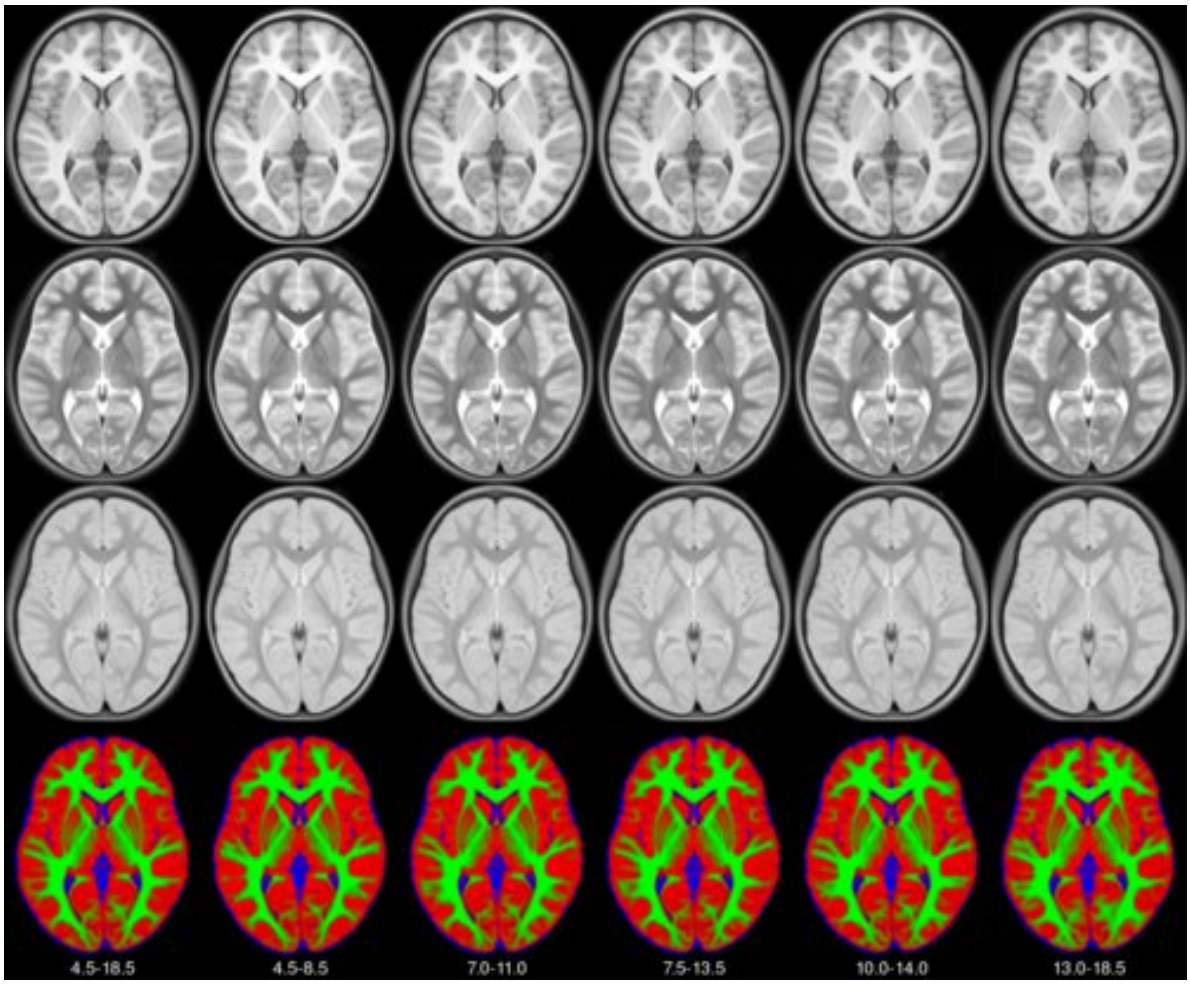
Pediatric

class torchio.datasets.mni.**Pediatric**(*years*, *symmetric=False*)

Bases: SubjectMNI

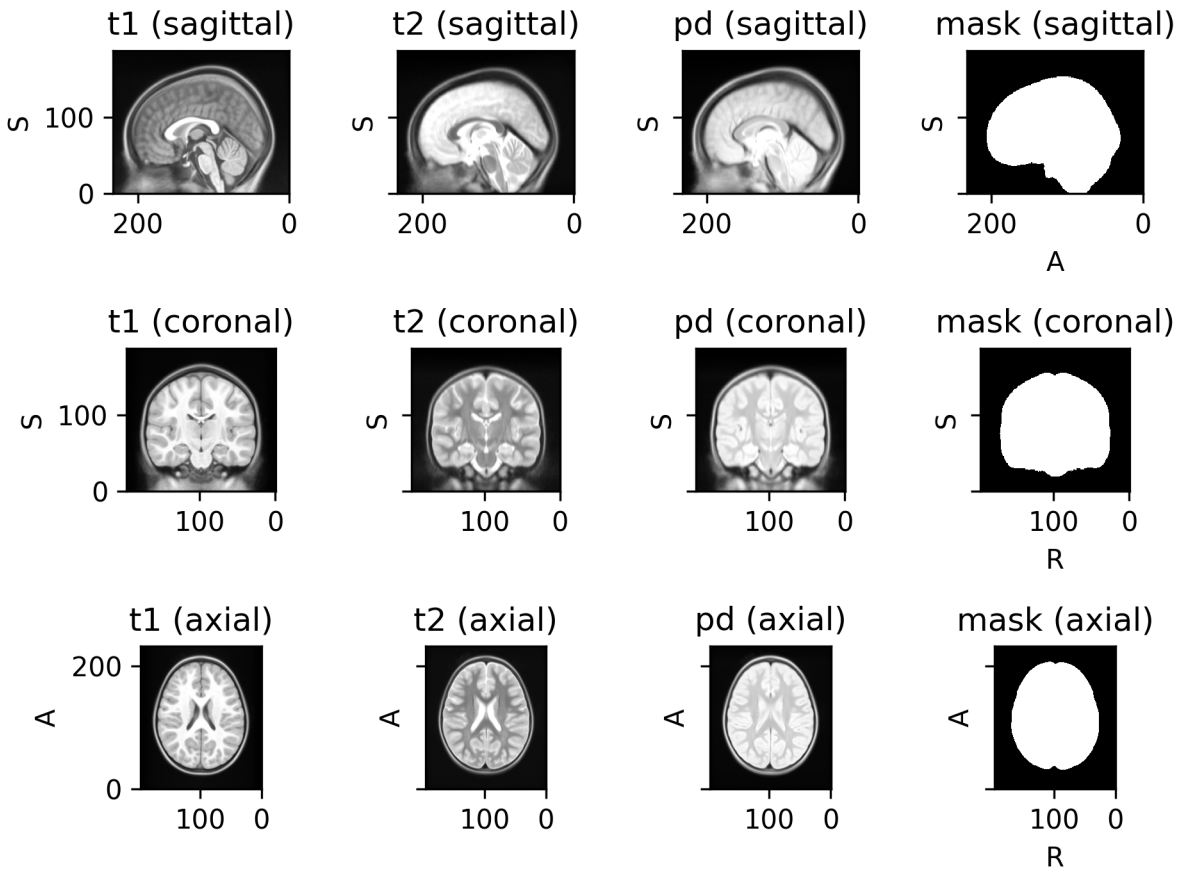
MNI pediatric atlases.

See the MNI website for more information.



Parameters

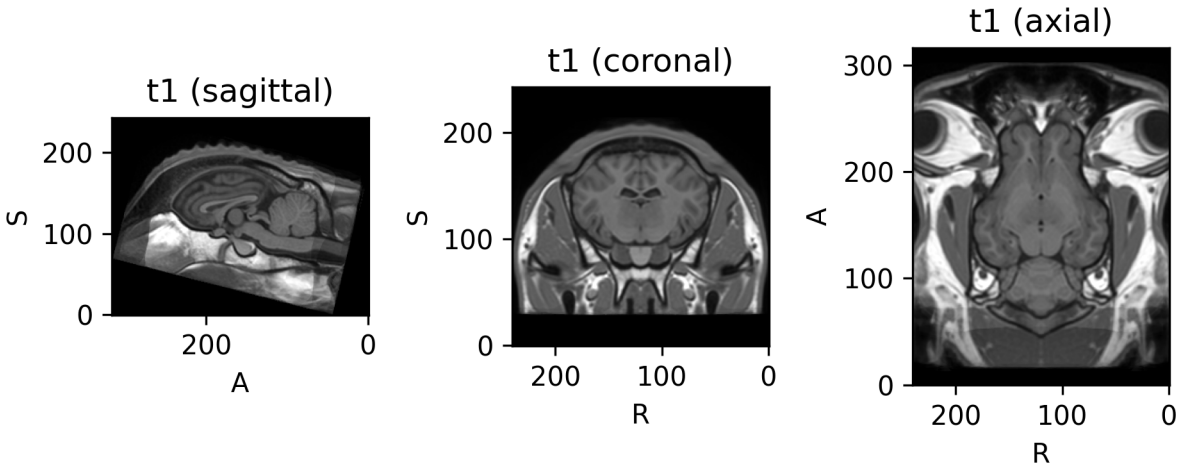
- **years** – Tuple of 2 ages. Possible values are: (4.5, 18.5), (4.5, 8.5), (7, 11), (7.5, 13.5), (10, 14) and (13, 18.5).
- **symmetric** – If True, the left-right symmetric templates will be used. Else, the asymmetric (natural) templates will be used.


t1 (sagittal)
t2 (sagittal)
pd (sagittal)
mask (sagittal)
S
100
0
200
0
A
t1 (coronal)
t2 (coronal)
pd (coronal)
mask (coronal)
100
R
t1 (axial)
t2 (axial)
pd (axial)
mask (axial)
A
200

Sheep

class torchio.datasets.mni.**Sheep**

Bases: SubjectMNI



BITE3

class torchio.datasets.bite.**BITE3**(*root: str | Path*, *transform:* Transform | *None* = *None*, *download: bool* = *False*, ***kwargs*)

Bases: BITE

Pre- and post-resection MR images in BITE.

The goal of BITE is to share in vivo medical images of patients wtith brain tumors to facilitate the development and validation of new image processing algorithms.

Please check the BITE website for more information and acknowledgments instructions.

Parameters

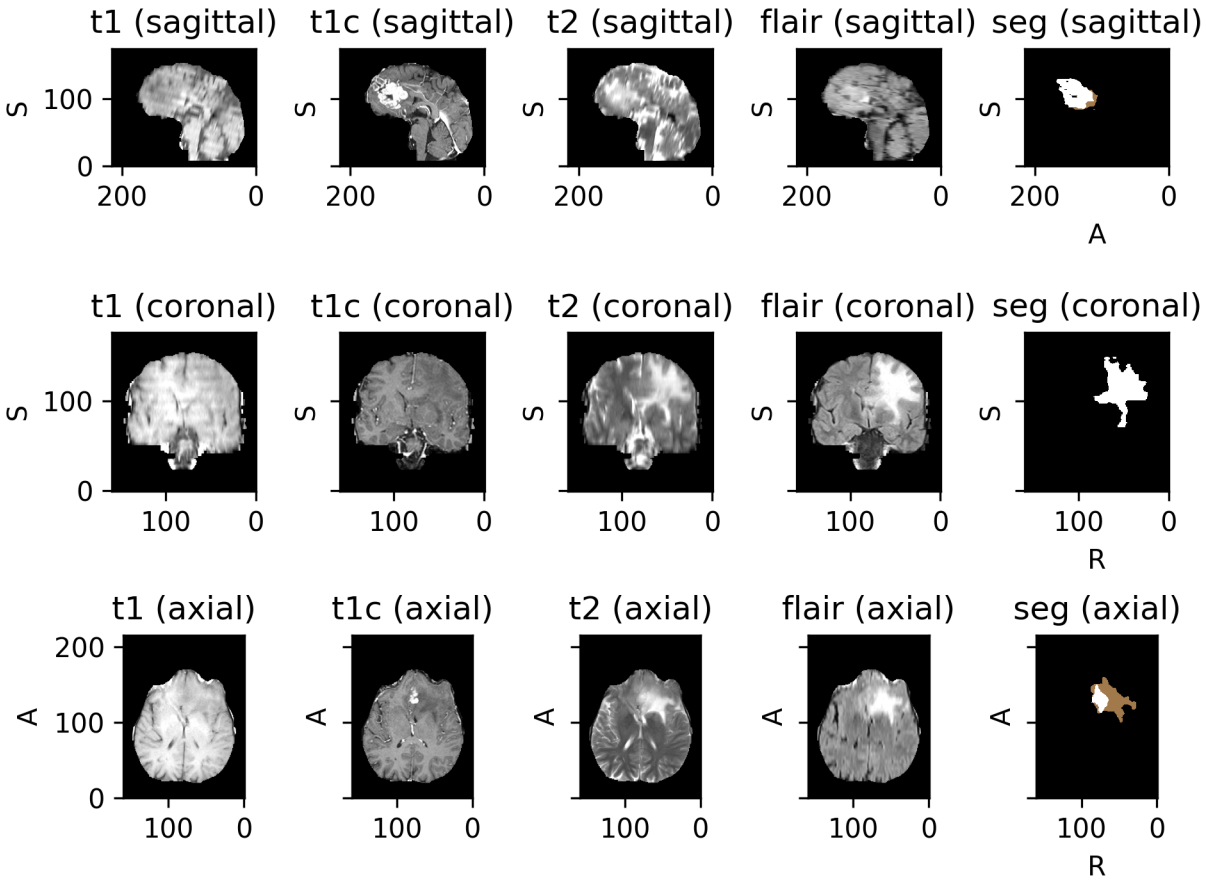
- **root** – Root directory to which the dataset will be downloaded.
- **transform** – An instance of *Transform*.
- **download** – If set to True, will download the data into root.

1.5.5 ITK-SNAP

BrainTumor

class torchio.datasets.itk_snap.**BrainTumor**

Bases: SubjectITKSNAP



T1T2

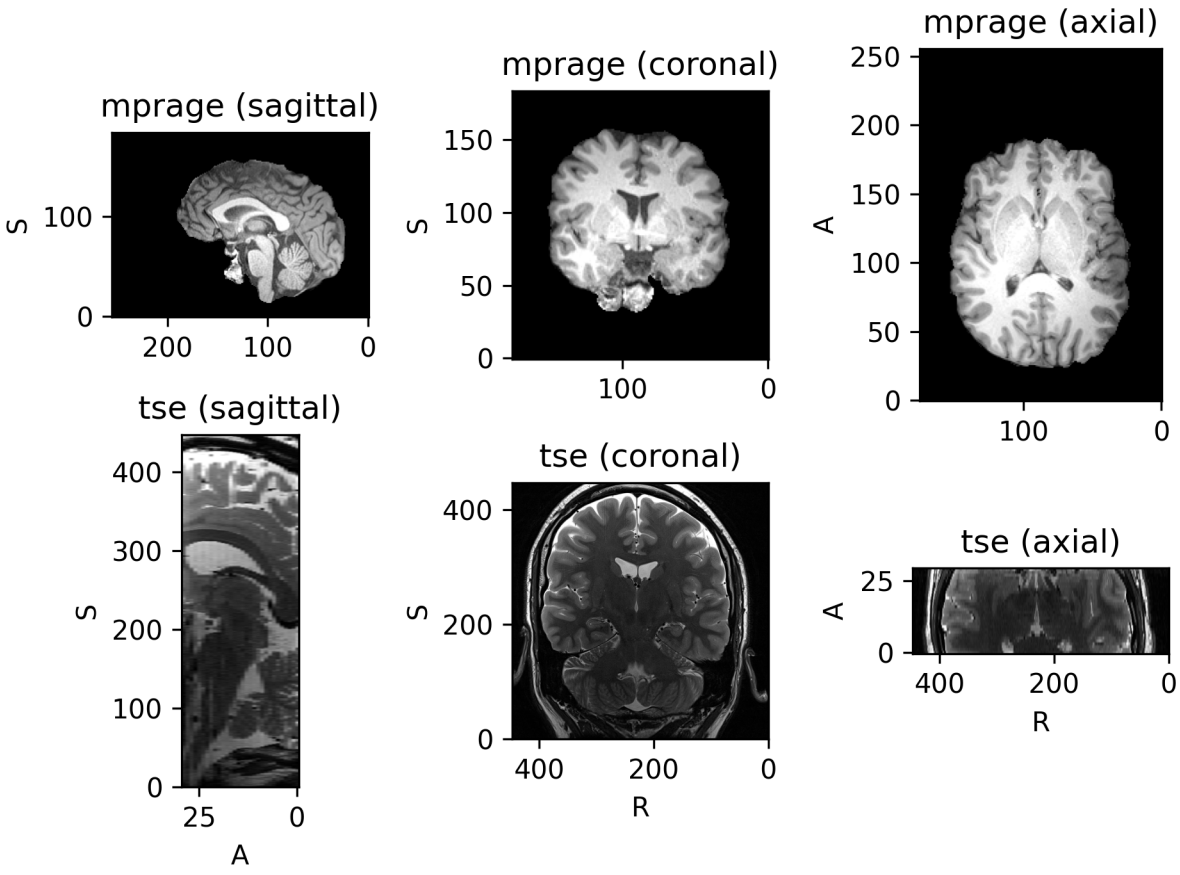
class torchio.datasets.itk_snap.**T1T2**

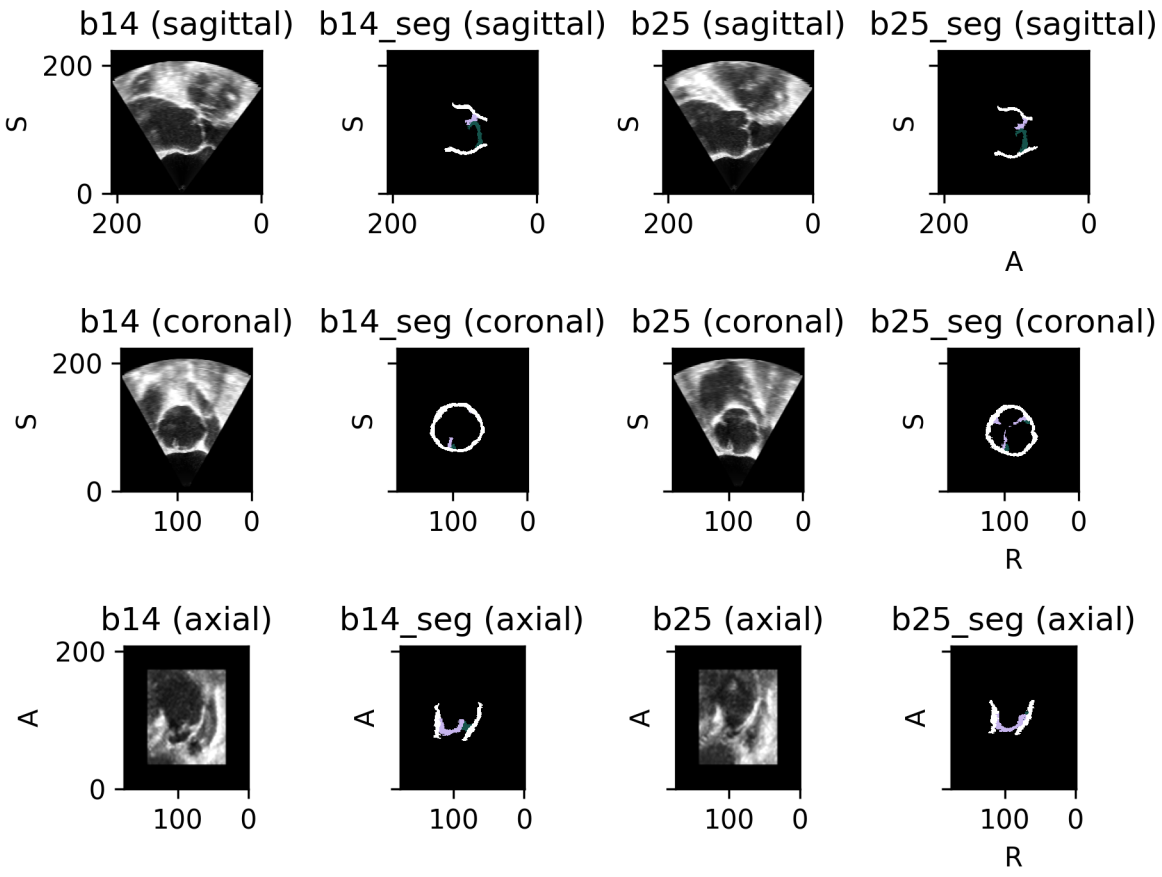
Bases: SubjectITKSNAP

AorticValve

class torchio.datasets.itk_snap.**AorticValve**

Bases: SubjectITKSNAP


mprage (sagittal)
S
100
0
200
100
0
mprage (coronal)
S
150
100
50
0
100
0
mprage (axial)
A
250
200
150
100
50
0
100
0
tse (sagittal)
S
400
300
200
100
0
25
0
A
tse (coronal)
S
400
200
0
400
200
0
R
tse (axial)
A
25
0
400
200
0
R


b14 (sagittal)
b14_seg (sagittal)
b25 (sagittal)
b25_seg (sagittal)
S
A
b14 (coronal)
b14_seg (coronal)
b25 (coronal)
b25_seg (coronal)
S
R
b14 (axial)
b14_seg (axial)
b25 (axial)
b25_seg (axial)
A
R
200
0
100

1.5.6 3D Slicer

Slicer

class torchio.datasets.slicer.**Slicer**(*name='MRHead'*)

Bases: _RawSubjectCopySubject

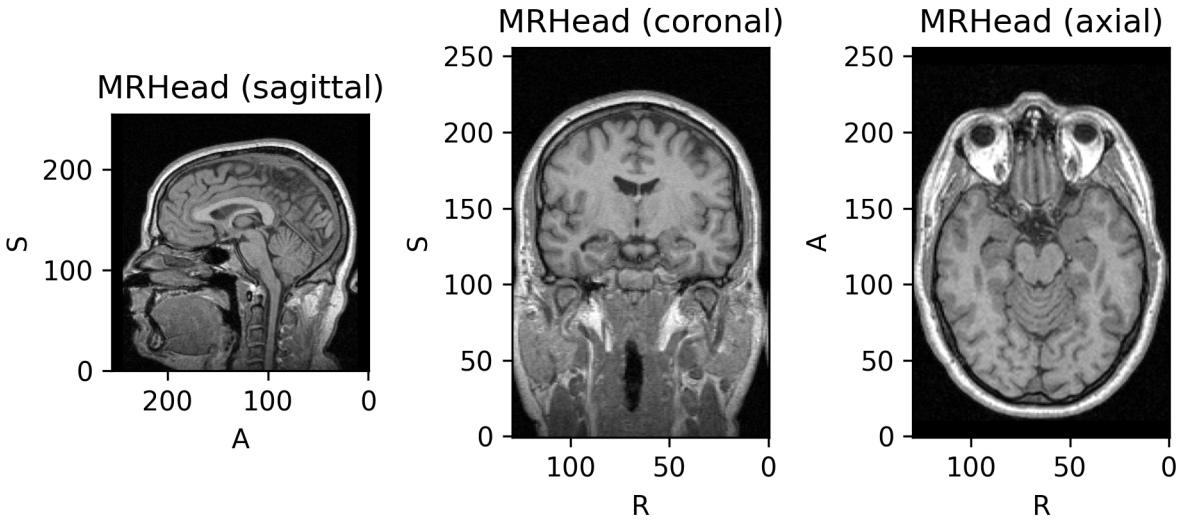
Sample data provided by 3D Slicer.

See the Slicer wiki for more information.

For information about licensing and permissions, check the Sample Data module.

Parameters

name – One of the keys in torchio.datasets.slicer.URLS_DICT.



1.5.7 FPG

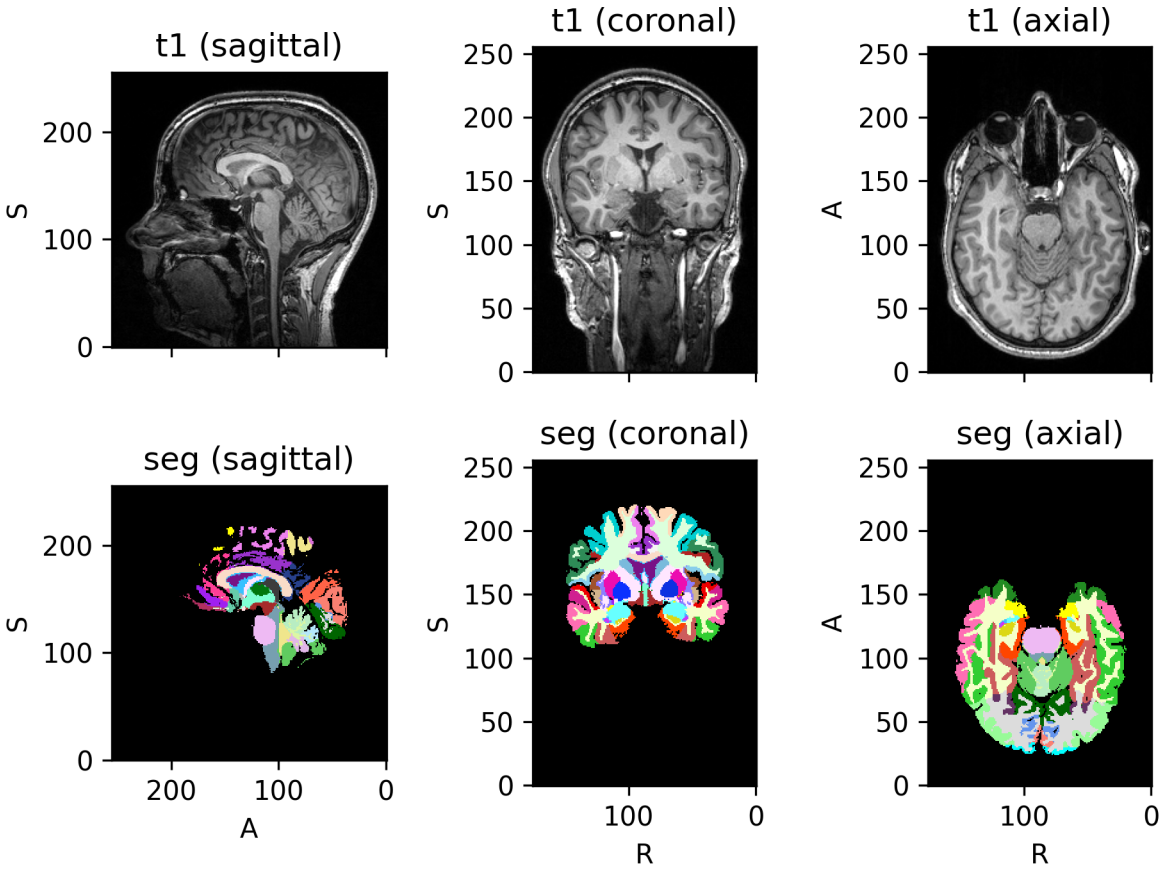
class torchio.datasets.fpg.**FPG**(*load_all: bool = False*)

Bases: _RawSubjectCopySubject

3T T_1-weighted brain MRI and corresponding parcellation.

Parameters

load_all – If True, three more images will be loaded: a T_2-weighted MRI, a diffusion MRI and a functional MRI.



1.5.8 MedMNIST

class torchio.datasets.medmnist.**OrganMNIST3D**(*split*, ***kwargs*)

Bases: MedMNIST

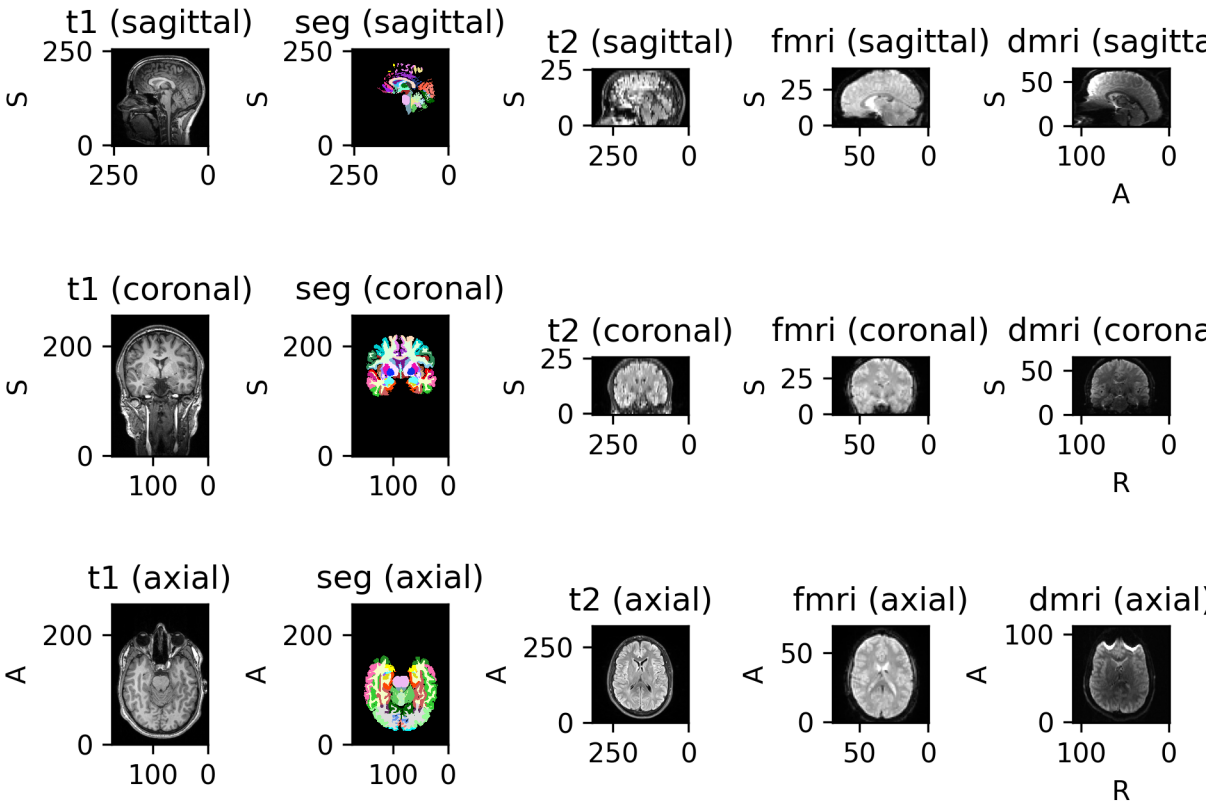
3D MedMNIST v2 datasets.

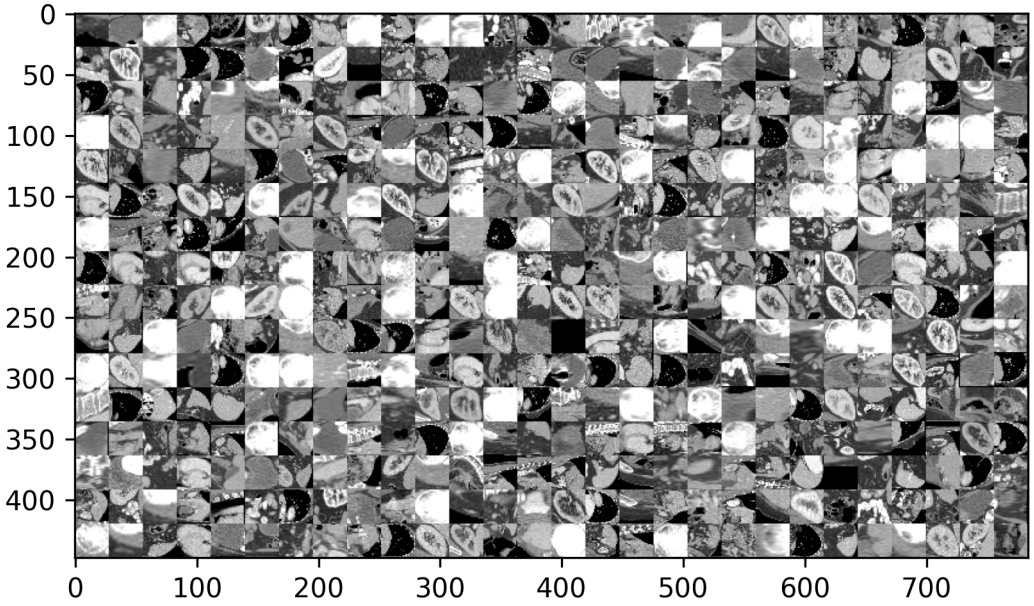
Datasets from MedMNIST v2: A Large-Scale Lightweight Benchmark for 2D and 3D Biomedical Image Classification.

Please check the MedMNIST website for more information, inclusing the license.

Parameters

split – Dataset split. Should be 'train', 'val' or 'test'.


t1 (sagittal)
seg (sagittal)
t2 (sagittal)
fmri (sagittal)
dmri (sagitta
t1 (coronal)
seg (coronal)
t2 (coronal)
fmri (coronal)
dmri (corona
t1 (axial)
seg (axial)
t2 (axial)
fmri (axial)
dmri (axial)
S
A
R


0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600
700

class torchio.datasets.medmnist.**NoduleMNIST3D**(*split*, ***kwargs*)

Bases: MedMNIST

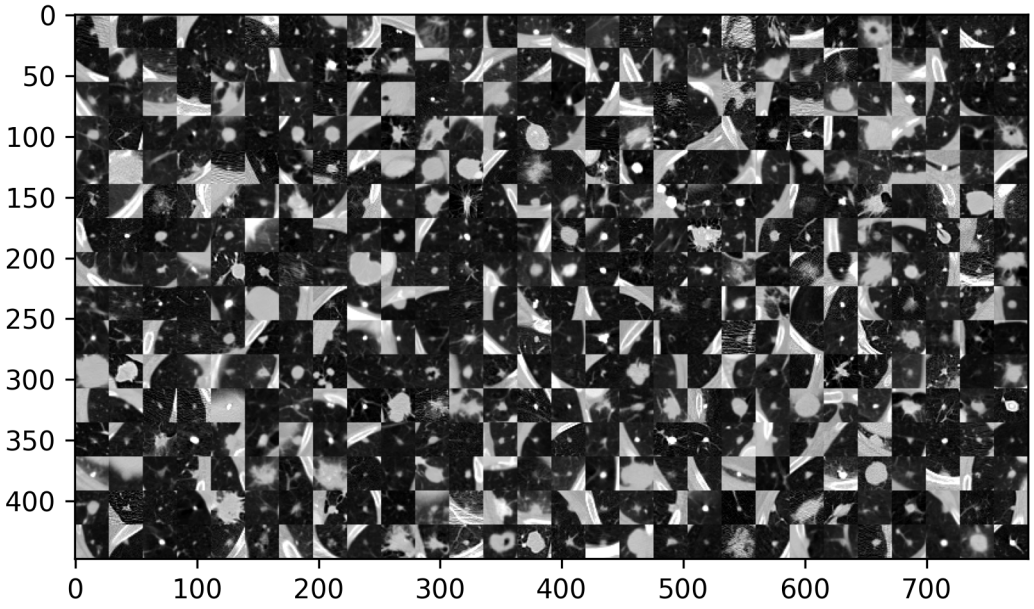
3D MedMNIST v2 datasets.

Datasets from MedMNIST v2: A Large-Scale Lightweight Benchmark for 2D and 3D Biomedical Image Classification.

Please check the MedMNIST website for more information, inclusing the license.

Parameters

split – Dataset split. Should be 'train', 'val' or 'test'.



class torchio.datasets.medmnist.**AdrenalMNIST3D**(*split*, ***kwargs*)

Bases: MedMNIST

3D MedMNIST v2 datasets.

Datasets from MedMNIST v2: A Large-Scale Lightweight Benchmark for 2D and 3D Biomedical Image Classification.

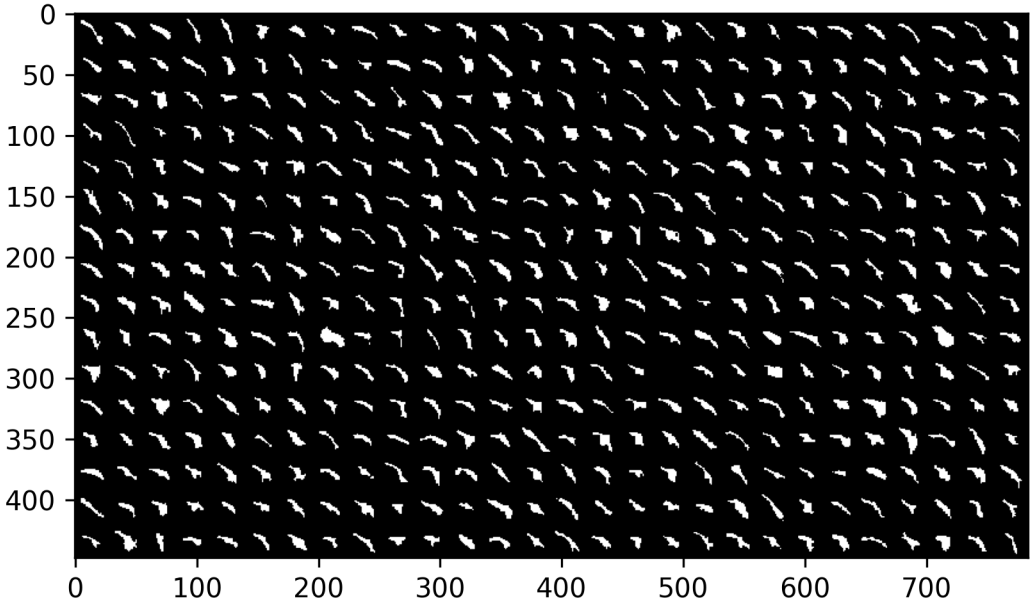
Please check the MedMNIST website for more information, inclusing the license.

Parameters

split – Dataset split. Should be 'train', 'val' or 'test'.

class torchio.datasets.medmnist.**FractureMNIST3D**(*split*, ***kwargs*)

Bases: MedMNIST

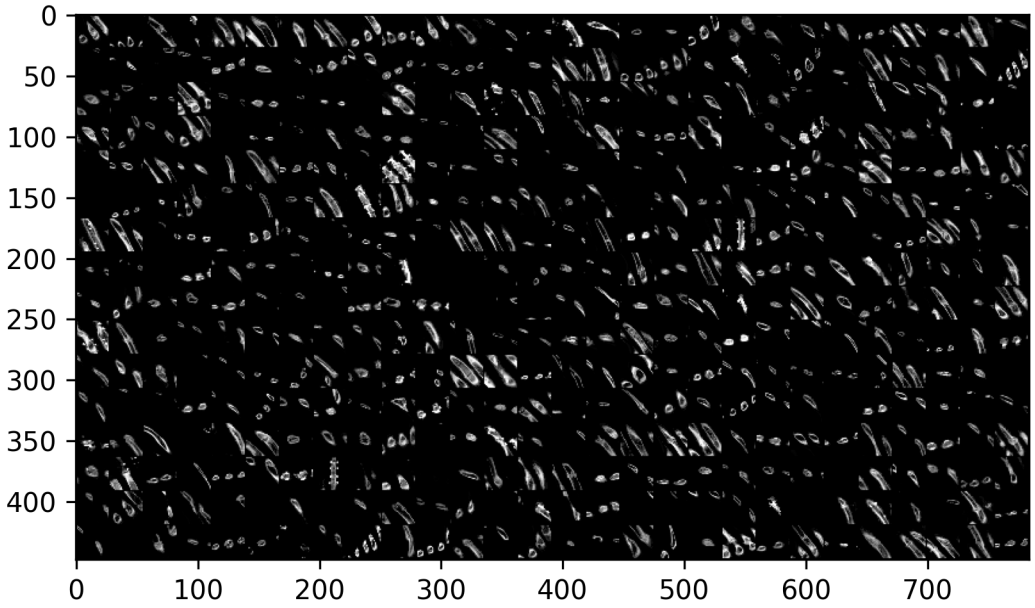

0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600
700

3D MedMNIST v2 datasets.

Datasets from MedMNIST v2: A Large-Scale Lightweight Benchmark for 2D and 3D Biomedical Image Classification.

Please check the MedMNIST website for more information, inclusing the license.

Parameters
split – Dataset split. Should be `'train'`, `'val'` or `'test'`.



class torchio.datasets.medmnist.**VesselMNIST3D**(*split*, ***kwargs*)

Bases: `MedMNIST`

3D MedMNIST v2 datasets.

Datasets from MedMNIST v2: A Large-Scale Lightweight Benchmark for 2D and 3D Biomedical Image Classification.

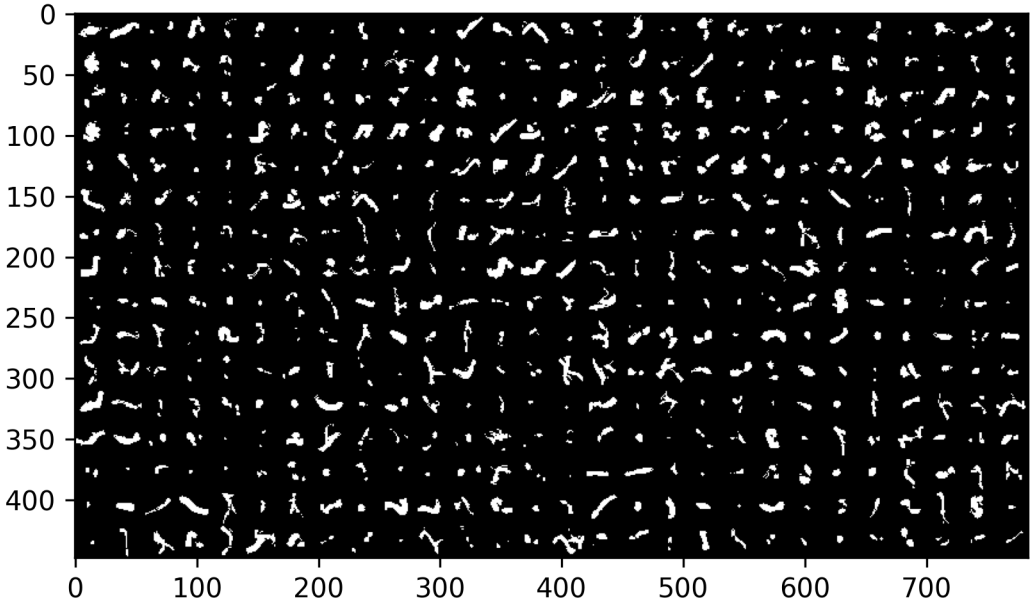
Please check the MedMNIST website for more information, inclusing the license.

Parameters
split – Dataset split. Should be `'train'`, `'val'` or `'test'`.

class torchio.datasets.medmnist.**SynapseMNIST3D**(*split*, ***kwargs*)

Bases: `MedMNIST`

3D MedMNIST v2 datasets.

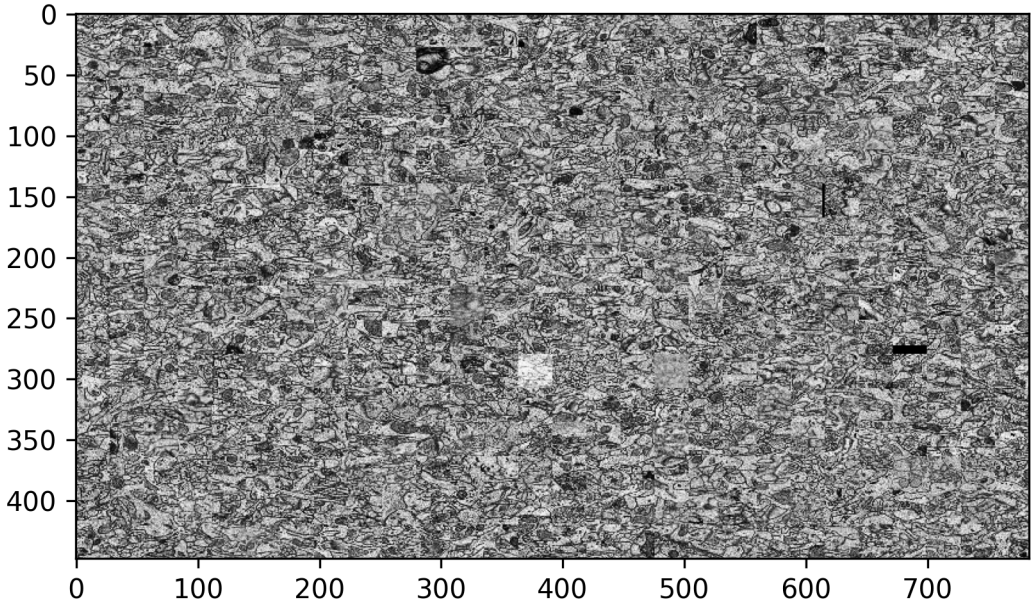

0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600
700

Datasets from MedMNIST v2: A Large-Scale Lightweight Benchmark for 2D and 3D Biomedical Image Classification.

Please check the MedMNIST website for more information, inclusing the license.

Parameters

split – Dataset split. Should be `'train'`, `'val'` or `'test'`.



1.6 Additional interfaces

TorchIO features can be accessed through the standard Python interface, the *Command-line tools* or the *3D Slicer GUI*.

1.6.1 Command-line tools

`tiotr`

A transform can be quickly applied to an image file using the command-line tool `tiotr`, which is automatically installed by `pip` during installation of TorchIO:

```
$ tiotr input.nii RandomAffine output.nii.gz --kwargs "degrees=(0,0,10) scales=0.1" --
↪seed 42
```

For more information, run `tiotr --help`.

`tiohd`

To print some image metadata, `tiohd` can be used. Adding the `--plot` argument will plot the image using Matplotlib:

```
$ tiohd ~/.cache/torchio/mni_colin27_1998_nifti/colin27_t1_tal_lin.nii
ScalarImage(shape: (1, 181, 217, 181); spacing: (1.00, 1.00, 1.00); orientation: RAS+;␣
↪dtype: torch.FloatTensor; memory: 27.1 MiB)
```

For more information, run `tiohd --help`.

1.6.2 3D Slicer GUI

3D Slicer is an open-source software platform for medical image informatics, image processing, and three-dimensional visualization.

TorchIO provides a 3D Slicer extension for quick experimentation and visualization of the package features without any coding.

The TorchIO extension can be easily installed using the Extensions Manager.

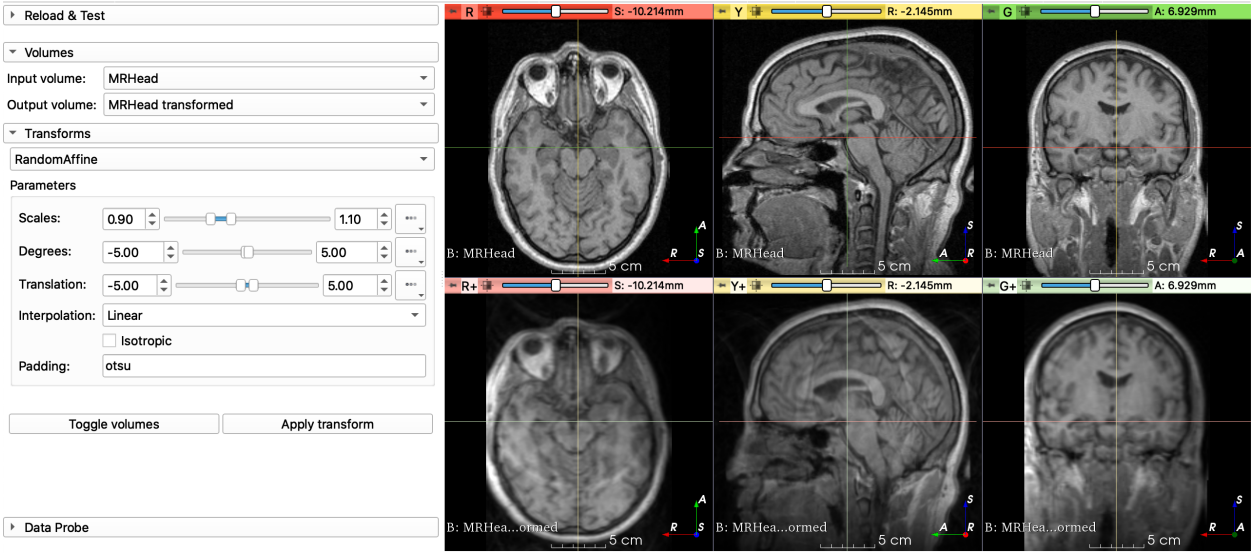
The code and installation instructions are available on GitHub.

Note: The Preview version (built nightly) is recommended. You can download and install Slicer from their download website or, if you are on macOS, using Homebrew:

```
brew tap homebrew/cask-versions && brew cask install slicer-preview
```

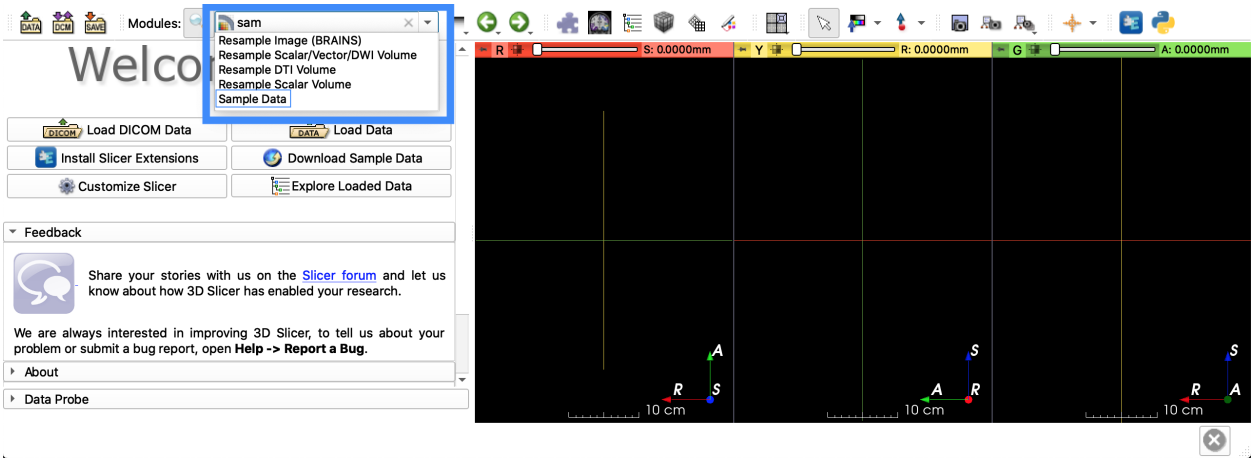
TorchIO Transforms

This module can be used to quickly visualize the effect of each transform parameter. That way, users can have an intuitive feeling of what the output of a transform looks like without any coding at all.

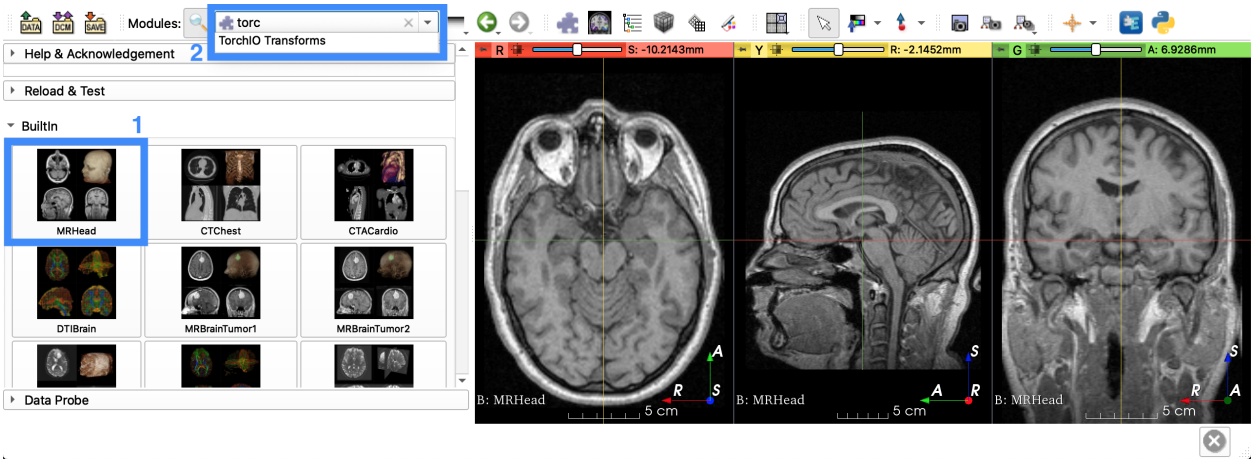


Usage example

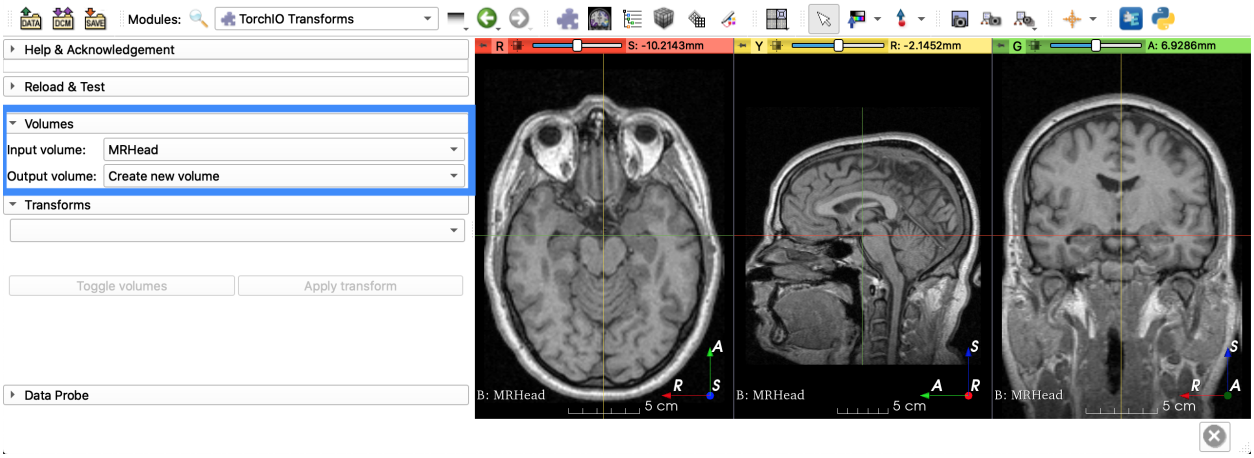
Go to the `Sample Data` module to get an image we can use:



Click on an image to download, for example MRHead[1], and go to the `TorchIO Transforms` module:

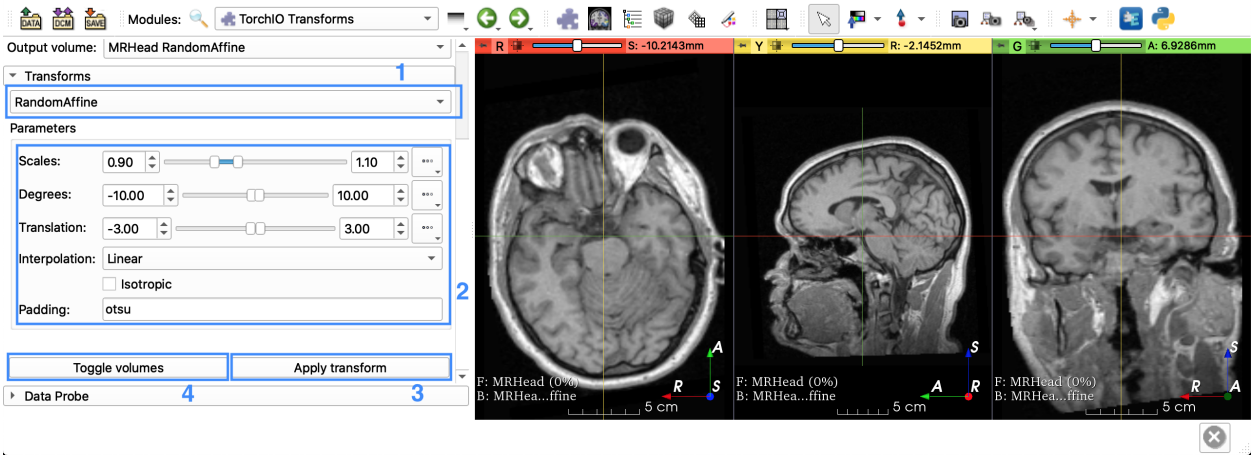


Select the input and output volume nodes:



[1] All the data in `Sample Data` can be downloaded and used in the TorchIO Python library using the *torchio.datasets.slicer.Slicer* class.

Modify the transform parameters and click on `Apply transform`. Hovering the mouse over the transforms will show tooltips extracted from the TorchIO documentation.



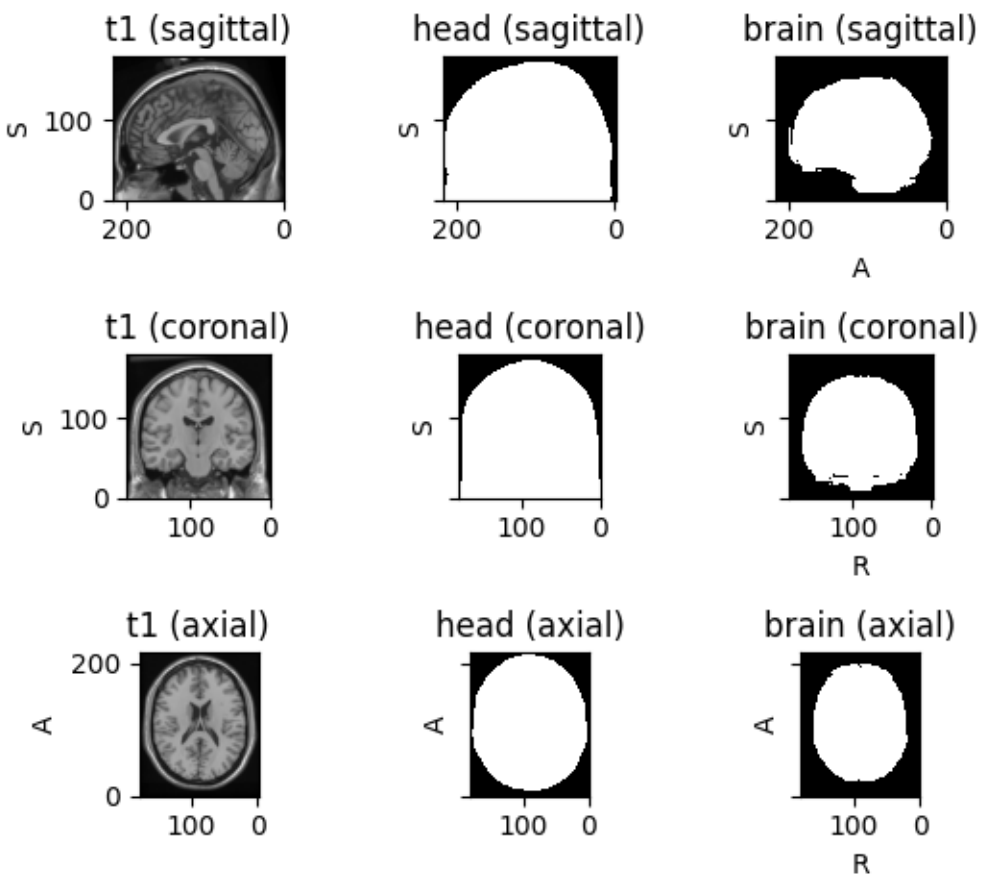
You can click on the `Toggle volumes` button to switch between input and output volumes.

1.7 Examples gallery

Below is a gallery of examples covering several features of TorchIO.

1.7.1 Plot a subject

Use *`torchio.Subject.plot()`* to plot the images within a subject.



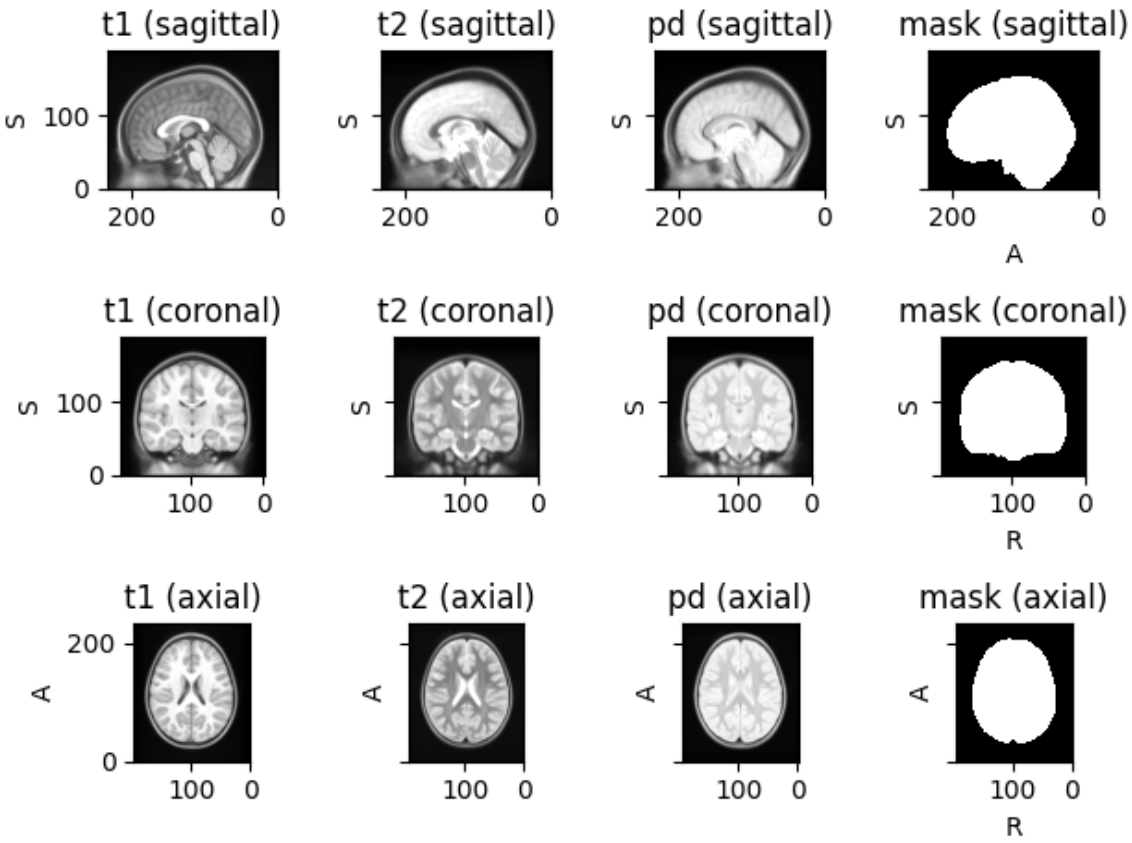

```
Downloading http://packages.bic.mni.mcgill.ca/mni-models/colin27/mni_colin27_1998_nifti.
↪zip to /home/docs/.cache/torchio/mni_colin27_1998_nifti/mni_colin27_1998_nifti.zip

0it [00:00, ?it/s]
  0%|          | 0/24250681 [00:00<?, ?it/s]
  1%|          | 163840/24250681 [00:00<00:15, 1539683.83it/s]
  3%|         | 712704/24250681 [00:00<00:06, 3642258.78it/s]
  6%|         | 1376256/24250681 [00:00<00:04, 4805977.22it/s]
  9%|         | 2195456/24250681 [00:00<00:03, 5896221.33it/s]
 13%|        | 3211264/24250681 [00:00<00:02, 7115232.39it/s]
 18%|        | 4415488/24250681 [00:01<00:02, 8591535.50it/s]
 24%|       | 5865472/24250681 [00:01<00:01, 10216039.72it/s]
 32%|      | 7651328/24250681 [00:01<00:01, 12184111.79it/s]
 40%|      | 9781248/24250681 [00:01<00:00, 14682527.86it/s]
 51%|     | 12304384/24250681 [00:01<00:00, 17790913.37it/s]
 62%|   | 14958592/24250681 [00:01<00:00, 20382877.98it/s]
 76%|  | 18432000/24250681 [00:01<00:00, 24646859.49it/s]
 94%|| 22749184/24250681 [00:01<00:00, 30154078.74it/s]
24256512it [00:01, 13738630.03it/s]
```

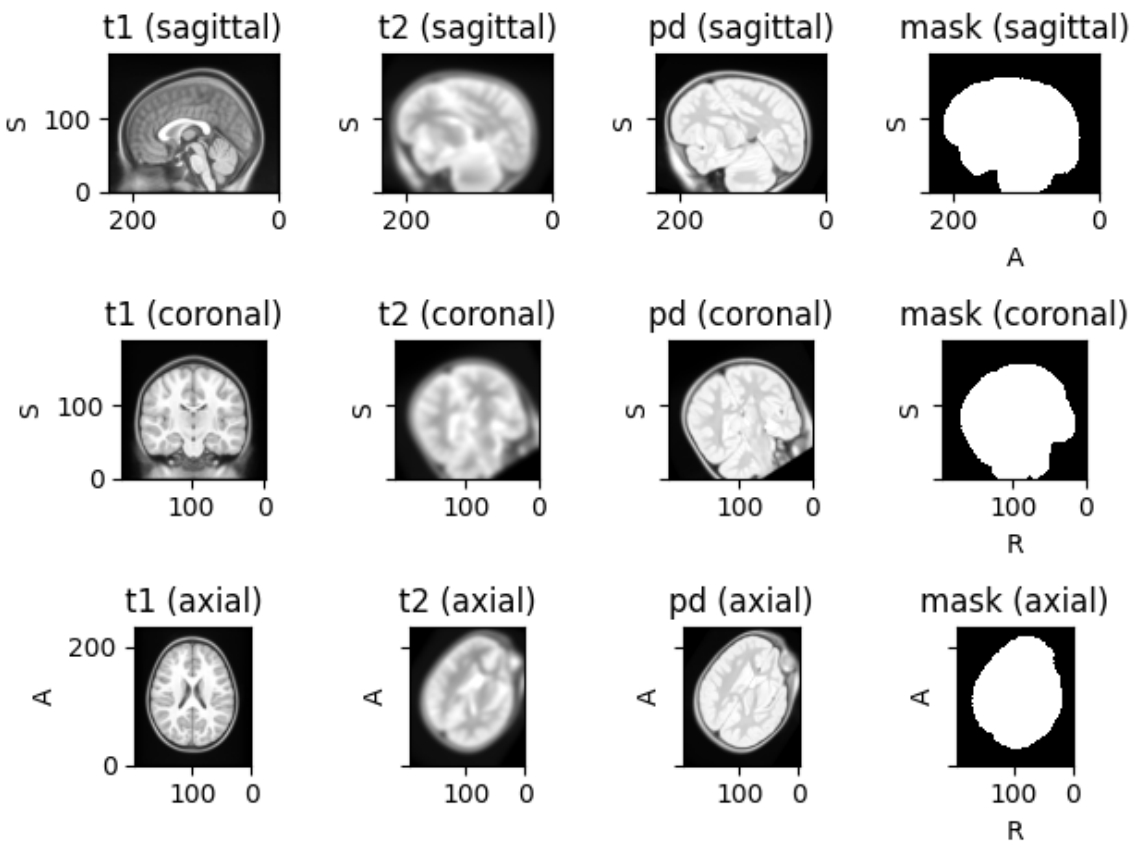
```
import torchio as tio

tio.datasets.Colin27().plot()
```

Total running time of the script: (0 minutes 6.183 seconds)

1.7.2 Exclude images from transform

In this example we show how the kwargs `include` and `exclude` can be used to apply a transform to only some of the images within a subject.






```
Downloading http://www.bic.mni.mcgill.ca/~vfonov/nihpd/obj1/nihpd_asym_04.5-08.5_nifti.
↪zip to /home/docs/.cache/torchio/nihpd_asym_04.5-08.5_nifti/nihpd_asym_04.5-08.5_nifti.
↪zip

0it [00:00, ?it/s]
  0%|          | 0/58169474 [00:00<?, ?it/s]
  0%|          | 114688/58169474 [00:00<03:20, 290160.58it/s]
  1%|          | 614400/58169474 [00:01<00:38, 1509254.09it/s]
  2%|         | 1220608/58169474 [00:01<00:21, 2688924.74it/s]
  3%|         | 1957888/58169474 [00:01<00:14, 3896739.59it/s]
  5%|         | 2850816/58169474 [00:01<00:10, 5188667.11it/s]
  7%|         | 3932160/58169474 [00:01<00:08, 6626122.61it/s]
  9%|         | 5242880/58169474 [00:01<00:06, 8266121.10it/s]
 12%|        | 6823936/58169474 [00:01<00:05, 10187306.59it/s]
 15%|        | 8626176/58169474 [00:01<00:03, 12401569.83it/s]
 18%|        | 10592256/58169474 [00:01<00:03, 14447167.58it/s]
 23%|       | 13197312/58169474 [00:02<00:02, 17786933.80it/s]
 28%|       | 16293888/58169474 [00:02<00:01, 21648870.53it/s]
 34%|      | 19546112/58169474 [00:02<00:01, 24844556.75it/s]
 41%|      | 23969792/58169474 [00:02<00:01, 30591929.30it/s]
 50%|     | 28917760/58169474 [00:02<00:00, 36097734.48it/s]
 60%|    | 34717696/58169474 [00:02<00:00, 42606308.14it/s]
 70%|   | 40878080/58169474 [00:02<00:00, 48069215.29it/s]
 83%| | 48381952/58169474 [00:02<00:00, 56102046.19it/s]
```

(continues on next page)

(continued from previous page)

```
 96%|| 55795712/58169474 [00:02<00:00, 61383152.31it/s]
58171392it [00:02, 20328685.52it/s]
```

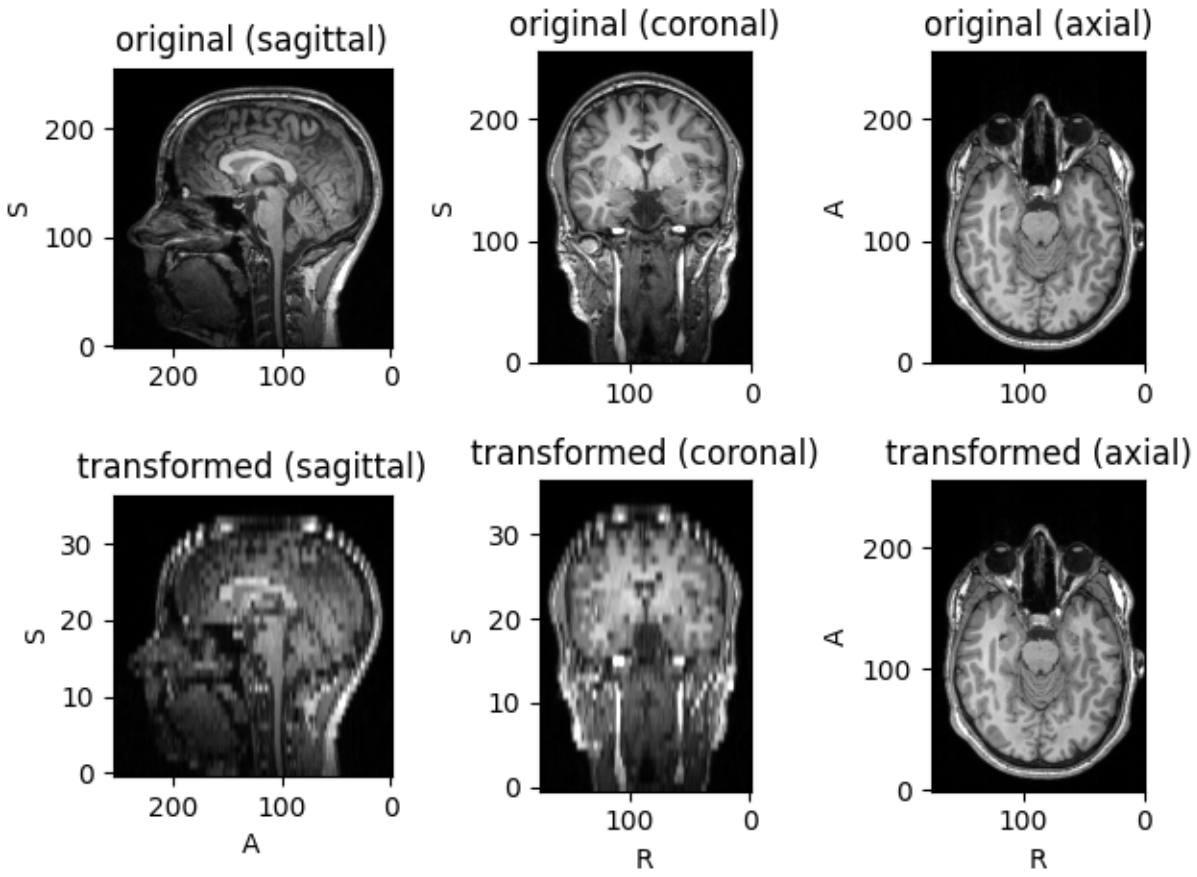
```
import torch
import torchio as tio


torch.manual_seed(0)

subject = tio.datasets.Pediatric(years=(4.5, 8.5))
subject.plot()
transform = tio.Compose(
    [
        tio.RandomAffine(degrees=(20, 30), exclude=['t1']),
        tio.RandomBlur(std=(3, 4), include=['t2']),
    ]
)
transformed = transform(subject)
transformed.plot()
```

Total running time of the script: (0 minutes 16.239 seconds)

1.7.3 Resample only one axis

In this example, we create a custom preprocessing transfom that changes the image spacing across one axis only.

Inspired by this discussion.




```
Downloading https://github.com/fepegar/torchio-data/raw/main/data/fernando/t1.nii.gz to /
↪home/docs/.cache/torchio/fpg/t1.nii.gz

0it [00:00, ?it/s]
  0%|          | 0/10860389 [00:00<?, ?it/s]
10862592it [00:00, 15195007.06it/s]
Downloading https://github.com/fepegar/torchio-data/raw/main/data/fernando/t1_seg_gif.
↪nii.gz to /home/docs/.cache/torchio/fpg/t1_seg_gif.nii.gz

0it [00:00, ?it/s]
  0%|          | 0/544126 [00:00<?, ?it/s]
548864it [00:00, 1174268.88it/s]
Downloading https://github.com/fepegar/torchio-data/raw/main/data/fernando/t1_to_mni.tfm␣
↪to /home/docs/.cache/torchio/fpg/t1_to_mni.tfm

0it [00:00, ?it/s]
  0%|          | 0/329 [00:00<?, ?it/s]
8192it [00:00, 26958.54it/s]
Downloading https://github.com/fepegar/torchio-data/raw/main/data/fernando/t1_to_mni_
↪affine.h5 to /home/docs/.cache/torchio/fpg/t1_to_mni_affine.h5

0it [00:00, ?it/s]
  0%|          | 0/8392 [00:00<?, ?it/s]
16384it [00:00, 49090.78it/s]
```

```
import torch
import torchio as tio


class ResampleZ:
    def __init__(self, spacing_z):
        self.spacing_z = spacing_z

    def __call__(self, subject):
        # We'll assume all images in the subject have the same spacing
        sx, sy, _ = subject.spacing
        resample = tio.Resample((sx, sy, self.spacing_z))
        resampled = resample(subject)
        return resampled


torch.manual_seed(42)
image = tio.datasets.FPG().t1
transforms = tio.ToCanonical(), ResampleZ(spacing_z=7)
transform = tio.Compose(transforms)
transformed = transform(image)
subject = tio.Subject(original=image, transformed=transformed)
subject.plot()
```

Total running time of the script: (0 minutes 3.614 seconds)

1.7.4 Sample slices from volumes

In this example, volumes are padded, scaled, rotated and sometimes flipped. Then, 2D slices are extracted.

```
import matplotlib.pyplot as plt
import torch
import torchio as tio

torch.manual_seed(0)
max_queue_length = 16
patches_per_volume = 2

subject = tio.datasets.Colin27()
subject.remove_image('head')

subjects = 50 * [subject]
max_side = max(subject.shape)
transform = tio.Compose(
    (
        tio.CropOrPad(max_side),
        tio.RandomFlip(),
        tio.RandomAffine(degrees=360),
    )
```

(continues on next page)

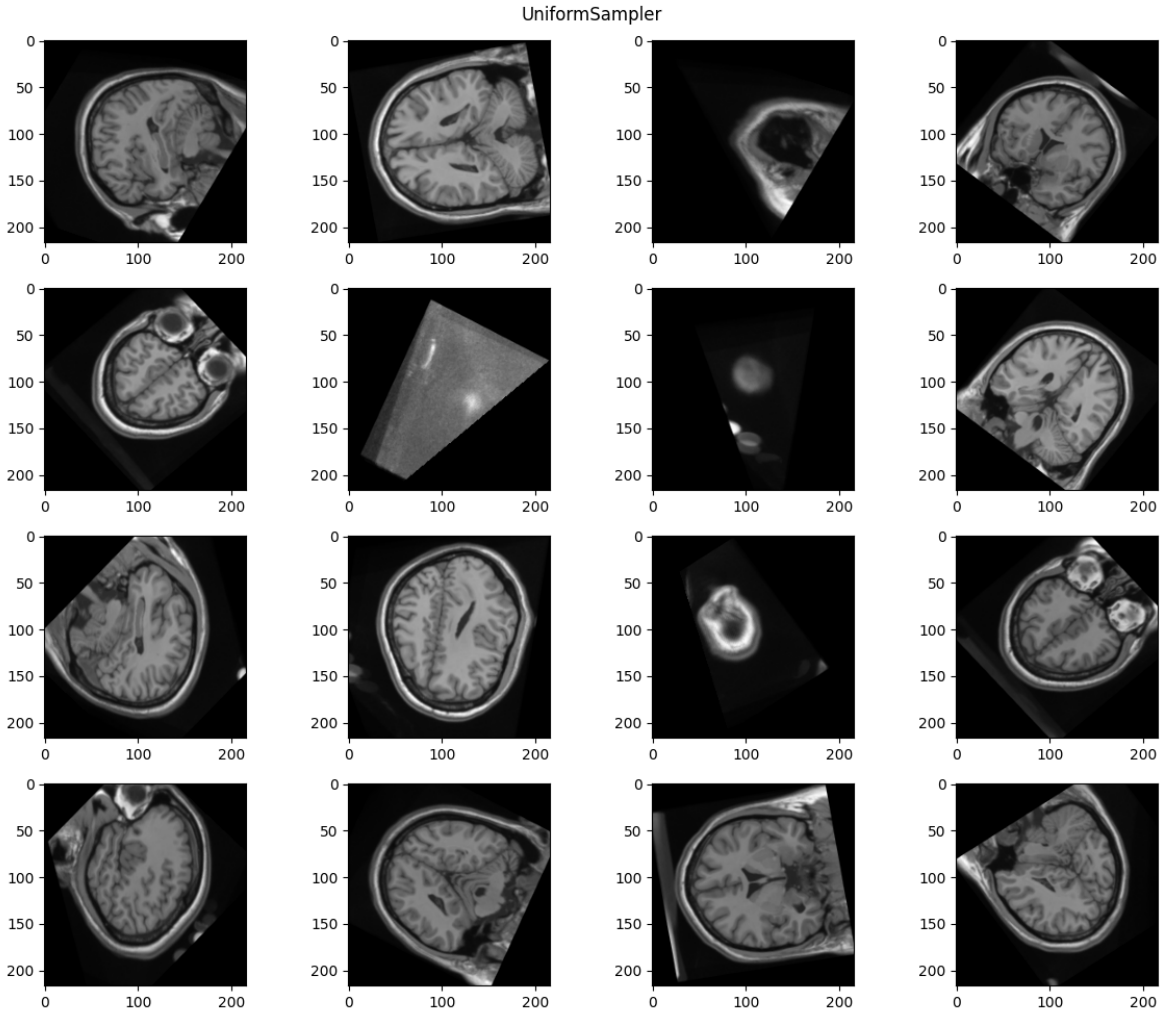
(continued from previous page)

```
)
dataset = tio.SubjectsDataset(subjects, transform=transform)
patch_size = (max_side, max_side, 1)  # 2D slices


def plot_batch(sampler):
    queue = tio.Queue(dataset, max_queue_length, patches_per_volume, sampler)
    loader = torch.utils.data.DataLoader(queue, batch_size=16)
    batch = tio.utils.get_first_item(loader)

    fig, axes = plt.subplots(4, 4, figsize=(12, 10))
    for ax, im in zip(axes.flatten(), batch['t1']['data']):
        ax.imshow(im.squeeze(), cmap='gray')
    plt.suptitle(sampler.__class__.__name__)
    plt.tight_layout()
```

Uniform sampler

When a `torchio.UniformSampler` is used, some of the patches don't contain much useful information:

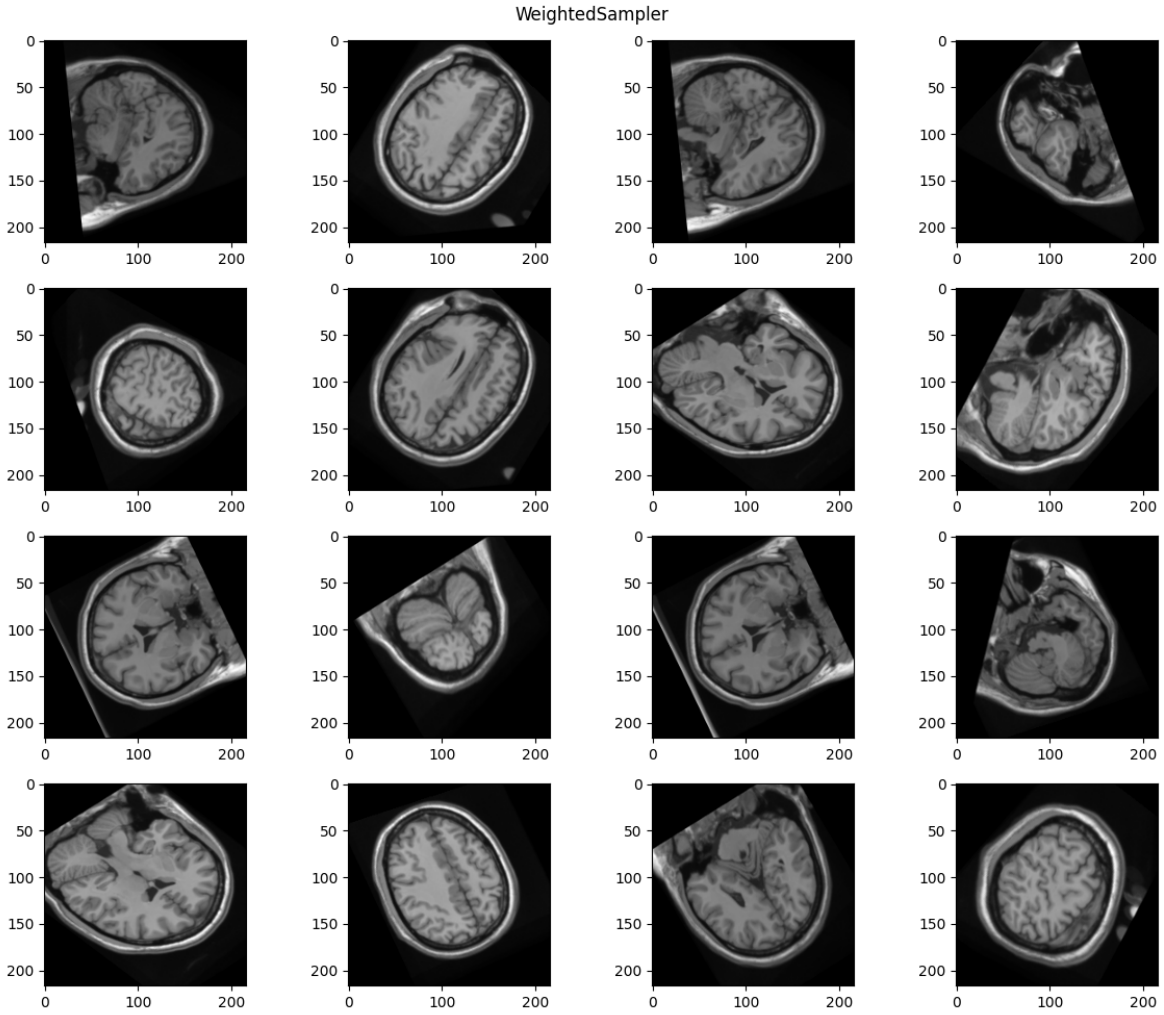
```
sampler = tio.UniformSampler(patch_size)
plot_batch(sampler)
```



Weighted sampler

We can use the `brain` image contained in the subject as a probability map for a `torchio.WeightedSampler`. That way, we ensure that the center of all patches correspond to brain tissue.

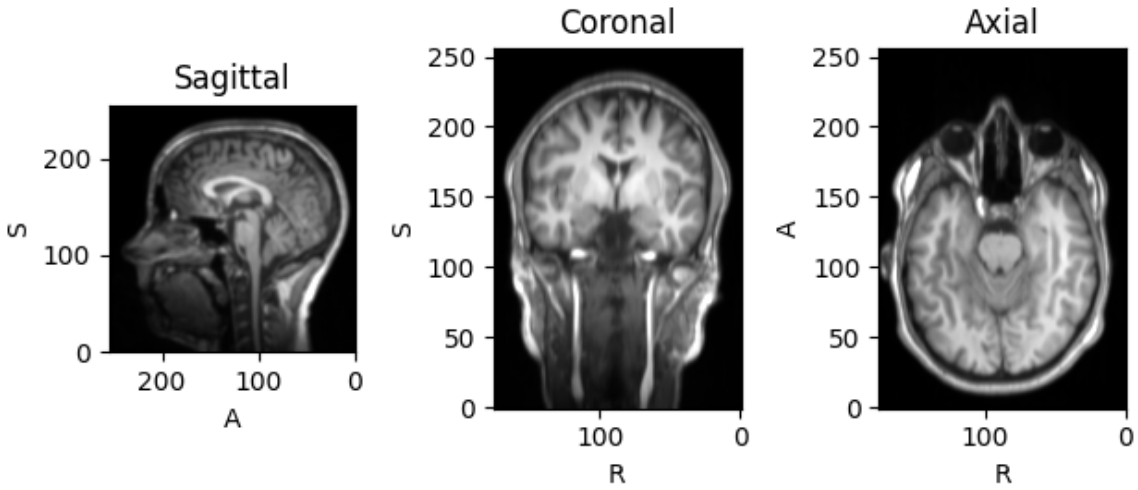
```
sampler = tio.WeightedSampler(patch_size, probability_map='brain')
plot_batch(sampler)

plt.show()
```

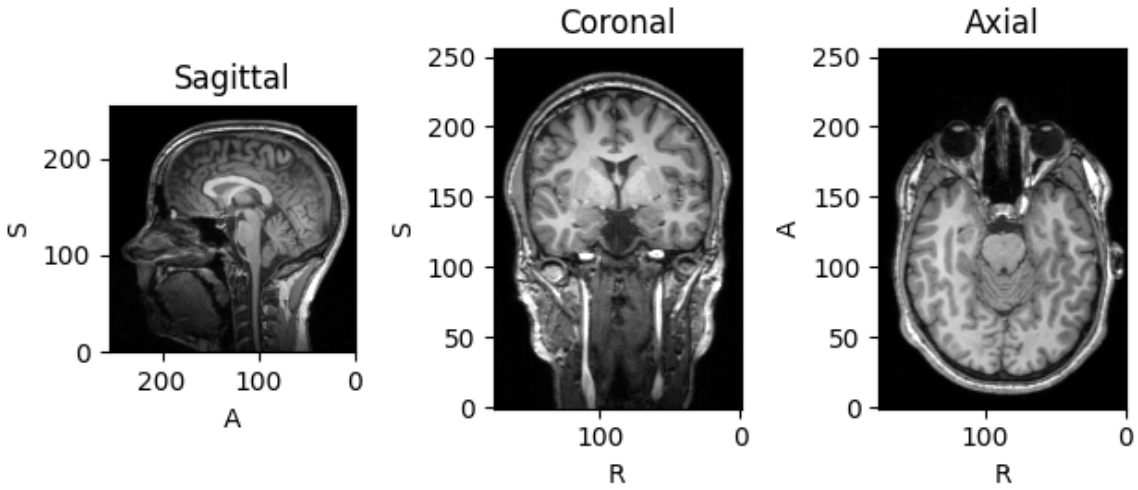


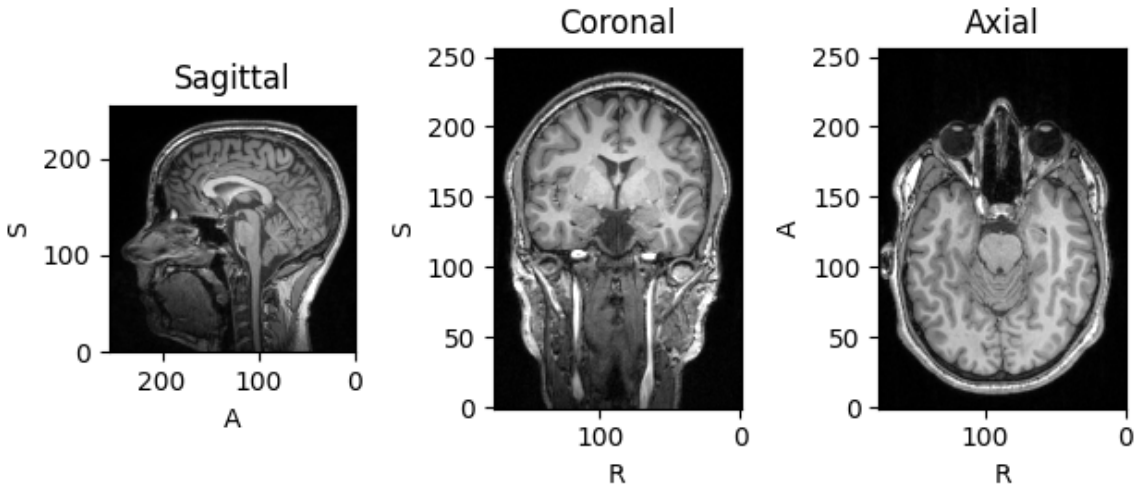
Total running time of the script: (0 minutes 23.530 seconds)

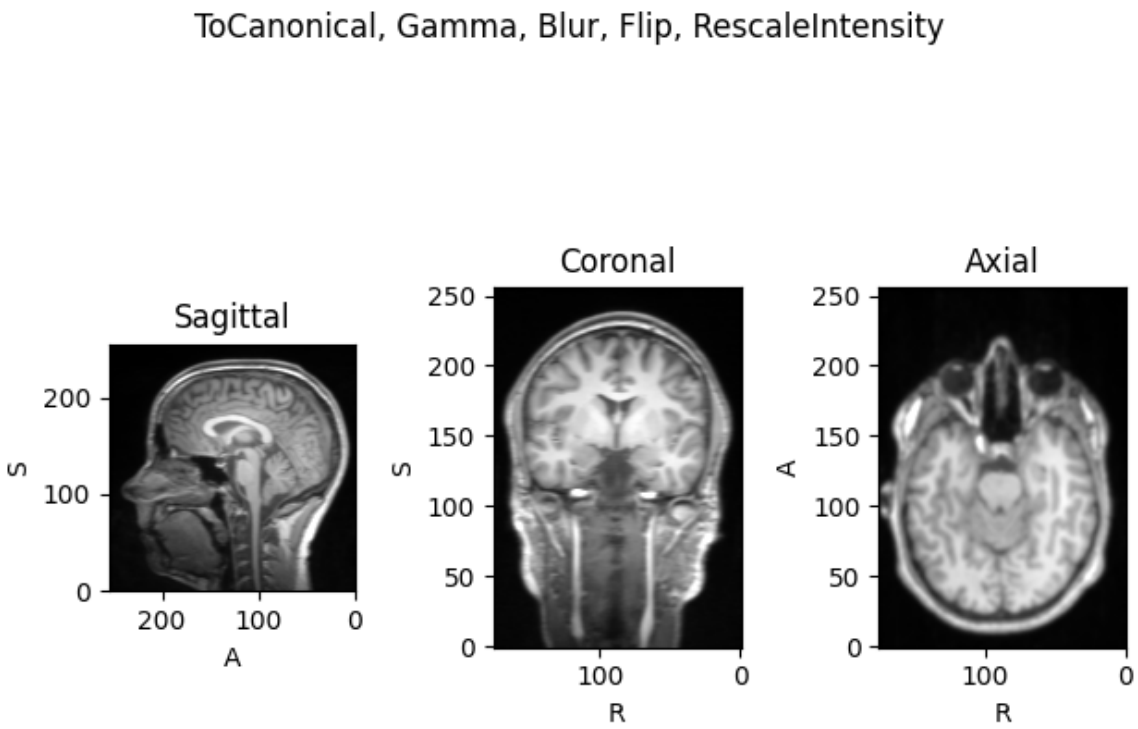
1.7.5 Trace applied transforms

Sometimes we would like to see which transform was applied to a certain batch during training. This can be done in TorchIO using `torchio.utils.history_collate()` for the data loader. The transforms history can be saved during training to check what was applied.


ToCanonical, Gamma, Blur, Flip, RescaleIntensity
Sagittal
Coronal
Axial
S
A
R
200
100
0
250
150
50


ToCanonical, Blur, RescaleIntensity
Sagittal
Coronal
Axial
S
A
R
250
200
150
100
50
0


ToCanonical, Gamma, Flip, RescaleIntensity
Sagittal
Coronal
Axial
S
A
S
R
A
R




```
Applied transforms:
[ToCanonical(),
 Gamma(gamma={'t1': [0.8018917031404817]}),
 RescaleIntensity(out_min_max=(-1, 1), percentiles=(0, 100), masking_method=None, in_min_
↪max=None)]

Composed transform to reproduce history:
Compose([ToCanonical(), Gamma(gamma={'t1': [0.8018917031404817]}), RescaleIntensity(out_
↪min_max=(-1, 1), percentiles=(0, 100), masking_method=None, in_min_max=None)])

Composed transform to invert applied transforms when possible:
/home/docs/checkouts/readthedocs.org/user_builds/torchio/checkouts/latest/src/torchio/
↪data/subject.py:197: RuntimeWarning: Skipping ToCanonical as it is not invertible
  inverse_transform = history_transform.inverse(warn=warn)
/home/docs/checkouts/readthedocs.org/user_builds/torchio/checkouts/latest/src/torchio/
↪data/subject.py:197: RuntimeWarning: Skipping RescaleIntensity as it is not invertible
  inverse_transform = history_transform.inverse(warn=warn)
Compose([Gamma(gamma={'t1': [0.8018917031404817]}, invert=True)])

Transforms applied to subjects in batch:
[[ToCanonical(),
  Gamma(gamma={'t1': [1.1259200934274378]}),
  Blur(std={'t1': tensor([0.5645, 1.3632, 1.8304])}),
  Flip(axes=(0,)),
```

(continues on next page)

(continued from previous page)

```
 RescaleIntensity(out_min_max=(-1, 1), percentiles=(0, 100), masking_method=None, in_
↪min_max=None)],
[ToCanonical(),
 Blur(std={'t1': tensor([0.5397, 0.3014, 0.0634])}),
 RescaleIntensity(out_min_max=(-1, 1), percentiles=(0, 100), masking_method=None, in_
↪min_max=None)],
[ToCanonical(),
 Gamma(gamma={'t1': [0.8567072622705179]}),
 Flip(axes=(0,)),
 RescaleIntensity(out_min_max=(-1, 1), percentiles=(0, 100), masking_method=None, in_
↪min_max=None)],
[ToCanonical(),
 Gamma(gamma={'t1': [0.7924771084926655]}),
 Blur(std={'t1': tensor([1.4525, 1.4022, 0.4076])}),
 Flip(axes=(0,)),
 RescaleIntensity(out_min_max=(-1, 1), percentiles=(0, 100), masking_method=None, in_
↪min_max=None)]]
```

```python
import pprint

import matplotlib.pyplot as plt
import torch
import torchio as tio

torch.manual_seed(0)

batch_size = 4
subject = tio.datasets.FPG()
subject.remove_image('seg')
subjects = 4 * [subject]

transform = tio.Compose(
    (
        tio.ToCanonical(),
        tio.RandomGamma(p=0.75),
        tio.RandomBlur(p=0.5),
        tio.RandomFlip(),
        tio.RescaleIntensity(out_min_max=(-1, 1)),
    )
)

dataset = tio.SubjectsDataset(subjects, transform=transform)

transformed = dataset[0]
print('Applied transforms:')  # noqa: T201
pprint.pprint(transformed.history)  # noqa: T203
print('\nComposed transform to reproduce history:')  # noqa: T201
```

(continues on next page)

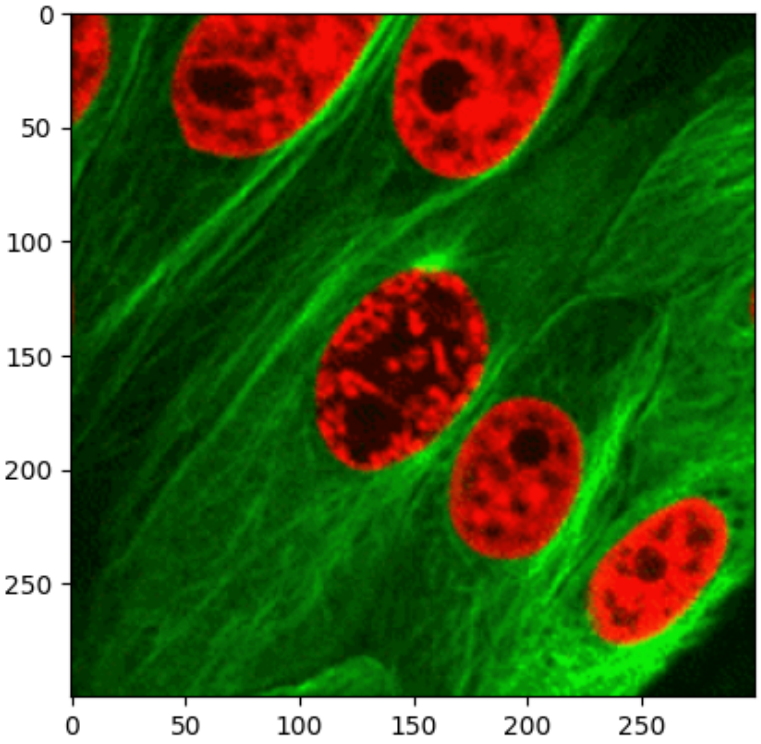
(continued from previous page)

```
print(transformed.get_composed_history())  # noqa: T201
print(
    '\nComposed transform to invert applied transforms when possible:'
)  # noqa: T201, B950
print(transformed.get_inverse_transform(ignore_intensity=False))  # noqa: T201

loader = torch.utils.data.DataLoader(
    dataset,
    batch_size=batch_size,
    collate_fn=tio.utils.history_collate,
)

batch = tio.utils.get_first_item(loader)
print('\nTransforms applied to subjects in batch:')  # noqa: T201
pprint.pprint(batch[tio.HISTORY])  # noqa: T203

for i in range(batch_size):
    tensor = batch['t1'][tio.DATA][i]
    affine = batch['t1'][tio.AFFINE][i]
    image = tio.ScalarImage(tensor=tensor, affine=affine)
    image.plot(show=False)
    history = batch[tio.HISTORY][i]
    title = ', '.join(t.name for t in history)
    plt.suptitle(title)
    plt.tight_layout()

plt.show()
```

Total running time of the script: (0 minutes 6.628 seconds)

1.7.6 Transform video

In this example, we use `torchio.Resample((2, 2, 1))` to divide the spatial size of the clip (height and width) by two and `RandomAffine(degrees=(0, 0, 20))` to rotate a maximum of 20 degrees around the time axis.



-
-
-

```
import matplotlib.animation as animation
import matplotlib.pyplot as plt
import numpy as np
import torch
import torchio as tio
from PIL import Image


def read_clip(path, undersample=4):
    """Read a GIF a return an array of shape (C, W, H, T)."""
    gif = Image.open(path)
    frames = []
    for i in range(gif.n_frames):
        gif.seek(i)
        frames.append(np.array(gif.convert('RGB')))
    frames = frames[::undersample]
    array = np.stack(frames).transpose(3, 1, 2, 0)
    delay = gif.info['duration']
    return array, delay
```

(continues on next page)

(continued from previous page)

```
def plot_gif(image):
    def _update_frame(num):
        frame = get_frame(image, num)
        im.set_data(frame)
        return

    def get_frame(image, i):
        return image.data[..., i].permute(1, 2, 0).byte()

    plt.rcParams['animation.embed_limit'] = 25
    fig, ax = plt.subplots()
    im = ax.imshow(get_frame(image, 0))
    return animation.FuncAnimation(
        fig,
        _update_frame,
        repeat_delay=image['delay'],
        frames=image.shape[-1],
    )


# Source: https://thehigherlearning.wordpress.com/2014/06/25/watching-a-cell-divide-
↪under-an-electron-microscope-is-mesmerizing-gif/  # noqa: B950
array, delay = read_clip('nBTu3oi.gif')
plt.imshow(array[..., 0].transpose(1, 2, 0))
plt.plot()
image = tio.ScalarImage(tensor=array, delay=delay)
original_animation = plot_gif(image)

transform = tio.Compose(
    (
        tio.Resample((2, 2, 1)),
        tio.RandomAffine(degrees=(0, 0, 20)),
    )
)

torch.manual_seed(0)
transformed = transform(image)
transformed_animation = plot_gif(transformed)
```

Total running time of the script: (0 minutes 13.527 seconds)

CHAPTER

TWO

SEE ALSO

PyTorch implementations of 2D and 3D network architectures:

- HighResNet
- U-Net

PYTHON MODULE INDEX

t

INDEX

Symbols

A

B

C

D

E

F

G

H

T

U

V

W

Z