# Preprocessing

## Intensity

### RescaleIntensity

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

Bases: `torchio.transforms.preprocessing.intensity.normalization_transform.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: Optional[Union[str,
    Callable[[torch.Tensor], torch.Tensor]], int, Tuple[int, int, int], Tuple[int,
    int, int, int, int, int]]] = None, **kwargs)                       [source]
```

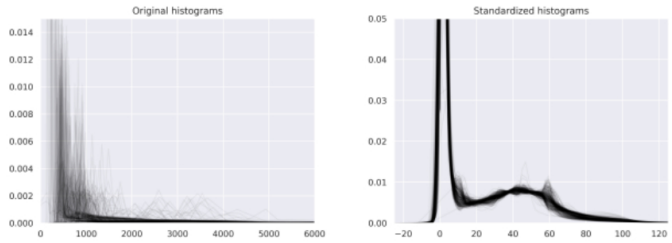
Bases: `torchio.transforms.preprocessing.intensity.normalization_transform.NormalizationTransform`

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: Union[str, os.PathLike,
    Dict[str, Union[str, os.PathLike, numpy.ndarray]]], masking_method:
    Optional[Union[str, Callable[[torch.Tensor], torch.Tensor]], int, Tuple[int, int,
    int], Tuple[int, int, int, int, int, int]]] = None, **kwargs)      [source]
```

Bases: `torchio.transforms.preprocessing.intensity.normalization_transform.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.

CONTENTS

**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[Union[str, os.PathLike]], cutoff:
    Optional[Tuple[float, float]] = None, mask_path: Optional[Union[str,
    os.PathLike, Sequence[Union[str, os.PathLike]]]] = None, masking_function:
    Optional[Callable] = None, output_path: Optional[Union[str, os.PathLike]] =
    None) → numpy.ndarray                                                [source]
```

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

## `Mask`

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

Bases: `torchio.transforms.intensity_transform.IntensityTransform`

Set voxels outside of mask to a constant value.

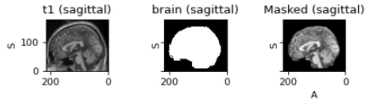
PARAMETERS

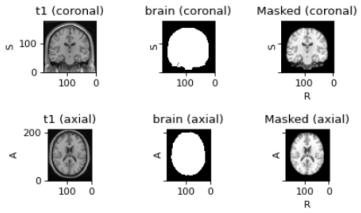
- **masking_method** – See `NormalizationTransform`.
- **outside_value** – Value to set for all voxels outside of the mask.
- **labels** – If a label map is used to generate the mask, sequence of labels to consider.
- ****kwargs** – See `Transform` for additional keyword arguments.

**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
```

(Source code, png, hires.png, pdf)






`Clamp`

```
class torchio.transforms.Clamp(out_min: Optional[float] = None, out_max:
    Optional[float] = None, **kwargs)                                    [source]
```

Bases: `torchio.transforms.intensity_transform.IntensityTransform`

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

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

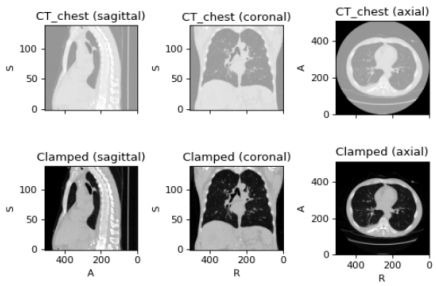
PARAMETERS

- **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)
```

(Source code, png, hires.png, pdf)




`NormalizationTransform`

```
class
    torchio.transforms.preprocessing.intensity.NormalizationTransform(masking_method:
    Optional[Union[str, Callable[[torch.Tensor], torch.Tensor], int, Tuple[int, int,
    int], Tuple[int, int, int, int, int, int]]] = None, **kwargs)        [source]
```

Bases: `torchio.transforms.intensity_transform.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 NormalizationTra
>>> 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: Optional[Union[int, Tuple[int, int,
```

```
int]]] = None, padding_mode: Union[str, float] = 0, mask_name: Optional[str] =
None, labels: Optional[Sequence[int]] = None, **kwargs)                    [source]
```

Bases: `torchio.transforms.spatial_transform.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

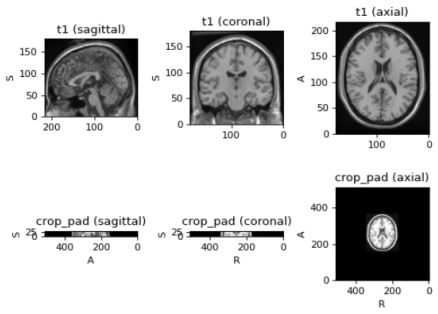
- **target_shape** – Tuple $(W, H, D)$. If a single value $N$ is provided, then $W = H = D = N$. If `None`, the shape will be computed from the `mask_name` (and the `labels`, if `labels` is not `None`).
- **padding_mode** – Same as `padding_mode` in `Pad`.
- **mask_name** – If `None`, the centers of 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.

(Source code, png, hires.png, pdf)




```
static _get_six_bounds_parameters(parameters: numpy.ndarray) → Tuple[int, int,
    int, int, int, int]                                                    [source]
```

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 $n_{ini} = \left\lceil \frac{n}{2} \right\rceil$ and $n_{fin} = \left\lfloor \frac{n}{2} \right\rfloor$.

**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:
    Optional[Sequence[str]] = None, exclude: Optional[Sequence[str]] = None, keys:
    Optional[Sequence[str]] = None, keep: Optional[Dict[str, str]] = None,
    parse_input: bool = True)                                              [source]
```

Bases: `torchio.transforms.spatial_transform.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:

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: Optional[Union[float, Tuple[float, float,
    float], str, pathlib.Path, torchio.data.image.Image]] = 1, image_interpolation:
    str = 'linear', label_interpolation: str = 'nearest', pre_affine_name:
    Optional[str] = None, scalars_only: bool = False, **kwargs)                [source]
```

Bases: `torchio.transforms.spatial_transform.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.

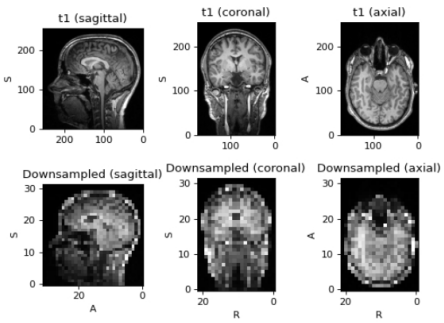
PARAMETERS

- **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 s
>>> # Example: using a precomputed transform to MNI space
>>> ref_path = tio.datasets.Colin27().t1.path  # this image is in the MNI space, so we ca
>>> affine_matrix = tio.io.read_matrix('transform_to_mni.txt')  # from a NiftyReg regist
>>> image = tio.ScalarImage(tensor=torch.rand(1, 256, 256, 180), to_mni=affine_matrix)  #
>>> transform = tio.Resample(colin.t1.path, pre_affine_name='to_mni')  # nearest neighbo
>>> transformed = transform(image)  # "image" is now in the MNI space
```

(Source code, png, hires.png, pdf)




## `Resize`

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

Bases: `torchio.transforms.spatial_transform.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

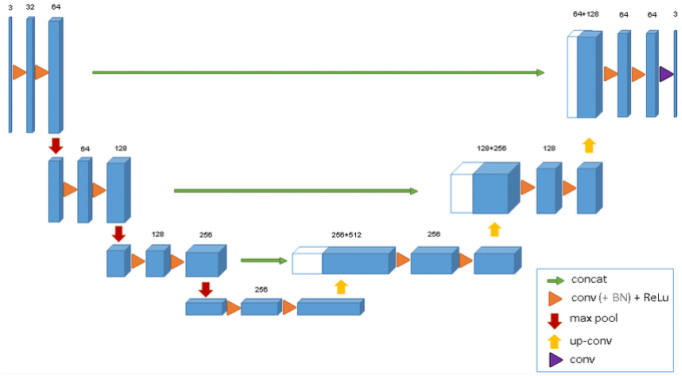
```
class torchio.transforms.EnsureShapeMultiple(target_multiple: Union[int, Tuple[int,
    int, int]], *, method: str = 'pad', **kwargs)                              [source]
```

Bases: `torchio.transforms.spatial_transform.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/module
    result = self.forward(*input, **kwargs)
  File "/home/fernando/miniconda3/envs/resseg/lib/python3.7/site-packages/unet/unet.py",
    x = self.decoder(skip_connections, encoding)
  File "/home/fernando/miniconda3/envs/resseg/lib/python3.7/site-packages/torch/nn/module
    result = self.forward(*input, **kwargs)
  File "/home/fernando/miniconda3/envs/resseg/lib/python3.7/site-packages/unet/decoding.p
    x = decoding_block(skip_connection, x)
  File "/home/fernando/miniconda3/envs/resseg/lib/python3.7/site-packages/torch/nn/module
    result = self.forward(*input, **kwargs)
  File "/home/fernando/miniconda3/envs/resseg/lib/python3.7/site-packages/unet/decoding.p
    x = torch.cat((skip_connection, x), dim=CHANNELS_DIMENSION)
RuntimeError: Sizes of tensors must match except in dimension 1. Got 45 and 44 in dimensi
>>> 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.

- **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)  [source]
```

Bases: `torchio.transforms.spatial_transform.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
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_consiste
    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: Optional[Union[int, Tuple[int, int, int],
    Tuple[int, int, int, int, int, int]]], **kwargs)  [source]
```

Bases: `torchio.transforms.preprocessing.spatial.bounds_transform.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.

**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: Optional[Union[int, Tuple[int, int, int],
    Tuple[int, int, int, int, int, int]]], padding_mode: Union[str, float] = 0,
    **kwargs)                                                              [source]
```

Bases: `torchio.transforms.preprocessing.spatial.bounds_transform.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:
    Optional[Union[str, Callable[[torch.Tensor], torch.Tensor]], int, Tuple[int, int,
    int], Tuple[int, int, int, int, int, int]]] = None, **kwargs)          [source]
```

Bases: `torchio.transforms.preprocessing.label.label_transform.LabelTransform`

Remap the integer ids of labels in a LabelMap.

This transformation may not be invertible if two labels are combined by the remapping. A masking method can be used to correctly split the label into two during the inverse transformation (see example).

PARAMETERS

- **remapping** – Dictionary that specifies how labels should be remapped. The keys are the old label ids, 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 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
>>> # 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
>>> # 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")
>>> # 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
>>> inverse_transformed = transformed.apply_inverse_transform()
```

`RemoveLabels`

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

Bases: `torchio.transforms.preprocessing.label.remap_labels.RemapLabels`

Remove labels from a label map by remapping them to the background label.

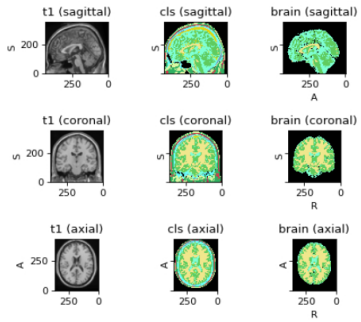
This transformation is not invertible.

PARAMETERS

- **labels** – A sequence of label integers that will be removed.
- **background_label** – integer that specifies which label is considered to be background (generally 0).

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

(Source code, png, hires.png, pdf)




## SequentialLabels

```
class torchio.transforms.SequentialLabels(masking_method: Optional[Union[str,
    Callable[[torch.Tensor], torch.Tensor], int, Tuple[int, int, int], Tuple[int,
    int, int, int, int, int]]] = None, **kwargs)                          [source]
```

Bases: `torchio.transforms.preprocessing.label.label_transform.LabelTransform`

Remap the integer IDs of labels in a LabelMap to be sequential.

For example, if a label map has 6 labels with IDs (3, 5, 9, 15, 16, 23), then this will apply a `RemapLabels` transform with `remapping={3: 1, 5: 2, 9: 3, 15: 4, 16: 5, 23: 6}`. This transformation is always fully invertible.

PARAMETERS

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

## OneHot

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

Bases: `torchio.transforms.preprocessing.label.label_transform.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:
    Optional[Sequence[str]] = None, exclude: Optional[Sequence[str]] = None, keys:
    Optional[Sequence[str]] = None, keep: Optional[Dict[str, str]] = None,
    parse_input: bool = True)                                             [source]
```

Bases: `torchio.transforms.preprocessing.label.label_transform.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: Optional[Sequence[str]] = None, exclude: Optional[Sequence[str]] = None,
    keys: Optional[Sequence[str]] = None, keep: Optional[Dict[str, str]] = None,
    parse_input: bool = True)                                             [source]
```

Bases: `torchio.transforms.preprocessing.label.label_transform.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.

Previous Transforms

Next Augmentation