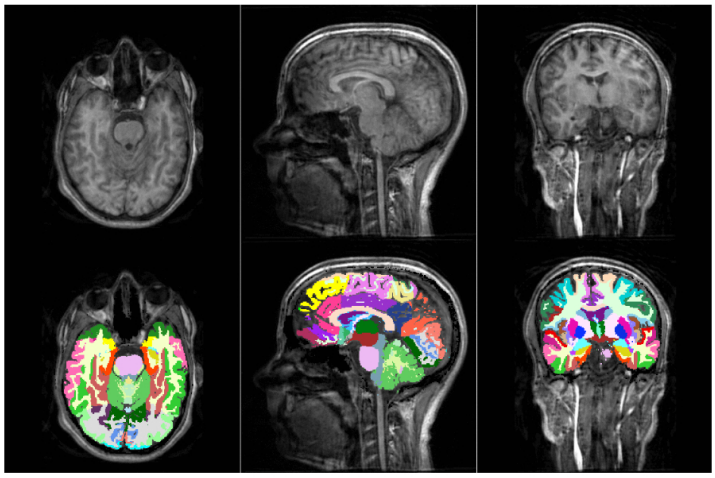
# Transforms



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:
    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]
```

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 images that will be kept in the subject and their new names.
- **parse_input** – If `True`, the input will be converted to an instance of `Subject`. This is used internally by some special transforms like `Compose`.

```
__call__(data: Union[torchio.data.subject.Subject, torchio.data.image.Image,
    torch.Tensor, numpy.ndarray, SimpleITK.SimpleITK.Image, dict,
    nibabel.nifti1.Nifti1Image]) → Union[torchio.data.subject.Subject,
    torchio.data.image.Image, torch.Tensor, numpy.ndarray,
    SimpleITK.SimpleITK.Image, dict, nibabel.nifti1.Nifti1Image]          [source]
```

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

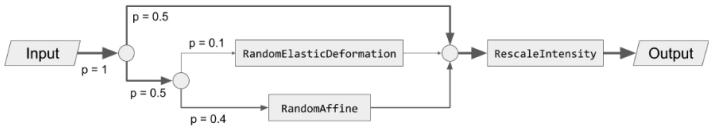
CONTENTS

`dict`. The output type is the same as the input type.

## Composability

Images 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)),
... ])
```

## 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)
```

## 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='
...     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`.

## 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) [source]

Bases: `enum.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.

## Transforms API

‹ Previous
Inference

Next
Preprocessing ›