# A quick look into data augmentations in nnUNetv2



## 0. Content



## 1. Introduction

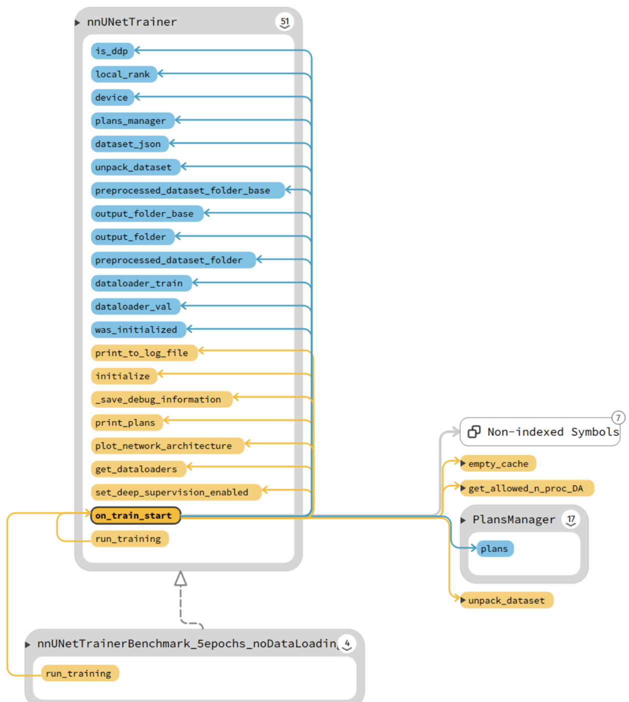
### a. General idea

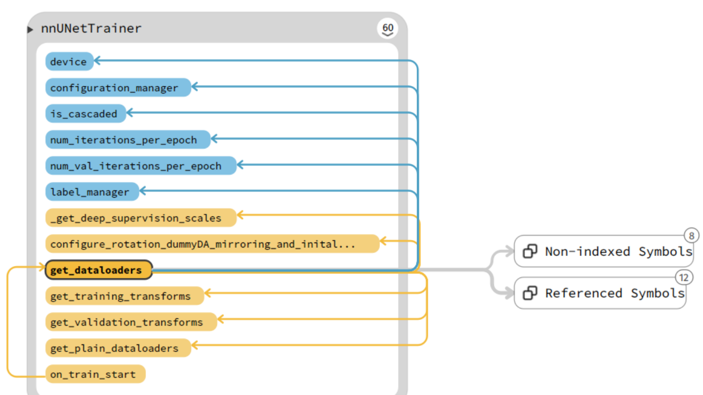
- As suggested by the author, data augmentation operations are mostly `fixed` (while nnUNet itself highlights adaptive data processing and training policies tailored to individual datasets).
- Based on commonly used `spatial transforms` and `color transforms`. Simple, conventional, but shown to be very effective in most cases (common anatomical structures and lesions + low-to-intermediate covariate shifts).
- Heavily relying on [batchgenerators](https://github.com/MIC-DKFZ/batchgenerators), a separate package written by DKFZ. Implemented mostly on CPU: numpy, scikit-image, etc. Further acceleration by their home-made multi-thread handlers (in practice people also use acceleration by pytorch dataloaders. while additional attention is needed when handling numpy random states (!) in that case).

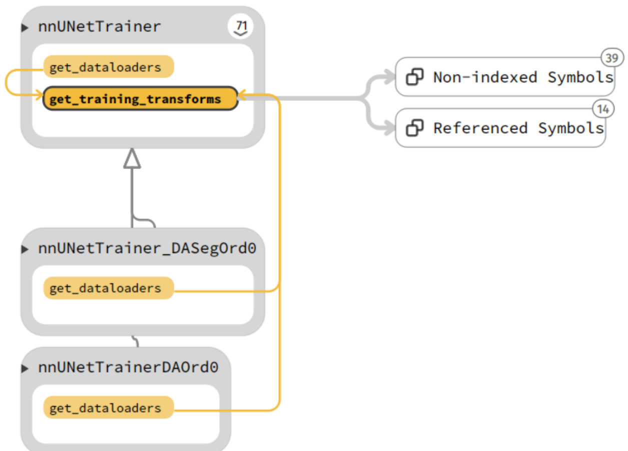
## b. Where and how it is called

- `/nnunetv2/training/nnUNetTrainer/nnUNetTrainer.py:`

`run_training` → `on_training_start` → `get_dataloaders` → `get_training_transforms`






```python
566     def get_dataloaders(self):
567         # we use the patch size to determine whether we need 2D or 3D dataloaders. We also use it to determine whether
568         # we need to use dummy 2D augmentation (in case of 3D training) and what our initial patch size should be
569         patch_size = self.configuration_manager.patch_size
570         dim = len(patch_size)
571 
572         # needed for deep supervision: how much do we need to downscale the segmentation targets for the different
573         # outputs?
574         deep_supervision_scales = self._get_deep_supervision_scales()
575 
576         rotation_for_DA, do_dummy_2d_data_aug, initial_patch_size, mirror_axes = \
577             self.configure_rotation_dummyDA_mirroring_and_inital_patch_size()
578 
579         # training pipeline
580         tr_transforms = self.get_training_transforms(
581             patch_size, rotation_for_DA, deep_supervision_scales, mirror_axes, do_dummy_2d_data_aug,
582             order_resampling_data=3, order_resampling_seg=1,
583             use_mask_for_norm=self.configuration_manager.use_mask_for_norm,
584             is_cascaded=self.is_cascaded, foreground_labels=self.label_manager.foreground_labels,
585             regions=self.label_manager.foreground_regions if self.label_manager.has_regions else None,
586             ignore_label=self.label_manager.ignore_label)
587 
588         # validation pipeline
589         val_transforms = self.get_validation_transforms(deep_supervision_scales,
590                                                         is_cascaded=self.is_cascaded,
591                                                         foreground_labels=self.label_manager.foreground_labels,
592                                                         regions=self.label_manager.foreground_regions if
593                                                         self.label_manager.has_regions else None,
594                                                         ignore_label=self.label_manager.ignore_label)
595 
596         dl_tr, dl_val = self.get_plain_dataloaders(initial_patch_size, dim)
597 
598         allowed_num_processes = get_allowed_n_proc_DA()
599         if allowed_num_processes == 0:
600             mt_gen_train = SingleThreadedAugmenter(dl_tr, tr_transforms)
601             mt_gen_val = SingleThreadedAugmenter(dl_val, val_transforms)
602         else:
603             mt_gen_train = LimitedLenWrapper(self.num_iterations_per_epoch, data_loader=dl_tr, transform=tr_transforms,
604                                              num_processes=allowed_num_processes, num_cached=6, seeds=None,
605                                              pin_memory=self.device.type == 'cuda', wait_time=0.02)
606             mt_gen_val = LimitedLenWrapper(self.num_val_iterations_per_epoch, data_loader=dl_val,
607                                            transform=val_transforms, num_processes=max(1, allowed_num_processes // 2),
608                                            num_cached=3, seeds=None, pin_memory=self.device.type == 'cuda',
609                                            wait_time=0.02)
610         return mt_gen_train, mt_gen_val
```

### Summary

- In `/nnunetv2/training/nnUNetTrainer/nnUNetTrainer.py`

    1. Line 576: Deciding rotation policy (adaptive) + computing actual patch size after possible rotations.
    2. Line 580:

Spatial transforms (almost fixed).

Intensity (color) transforms (almost fixed).

Additional steps for cascaded training (low-res + full_res).

Will be introduced in details in the below section.

---

## 2. Key components

### a. Deciding rotation policies and possible patch sizes (obtained after rotations)

**Where it resides**:

`nnUNetTrainer.py`:

```
    def get_dataloaders(self):
        # we use the patch size to determine whether we need 2D or 3D dataloaders. We also use it to determine whether
        # we need to use dummy 2D augmentation (in case of 3D training) and what our initial patch size should be
        patch_size = self.configuration_manager.patch_size
        dim = len(patch_size)

        # needed for deep supervision: how much do we need to downscale the segmentation targets for the different
        # outputs?
        deep_supervision_scales = self._get_deep_supervision_scales()

        rotation_for_DA, do_dummy_2d_data_aug, initial_patch_size, mirror_axes = \
            self.configure_rotation_dummyDA_mirroring_and_inital_patch_size()
```

This calls the following:


```
    def get_dataloaders(self):
        # we use the patch size to determine whether we need 2D or 3D dataloaders. We also use it to determine whether
        # we need to use dummy 2D augmentation (in case of 3D training) and what our initial patch size should be
        patch_size = self.configuration_manager.patch_size
        dim = len(patch_size)

        # needed for deep supervision: how much do we need to downscale the segmentation targets for the different
        # outputs?
        deep_supervision_scales = self._get_deep_supervision_scales()

        rotation_for_DA, do_dummy_2d_data_aug, initial_patch_size, mirror_axes = \
            self.configure_rotation_dummyDA_mirroring_and_inital_patch_size()
```


The detailed implementation:

```python
354     def configure_rotation_dummyDA_mirroring_and_inital_patch_size(self):
355         """
356         This function is stupid and certainly one of the weakest spots of this implementation. Not entirely sure how we can f
357         """
358         patch_size = self.configuration_manager.patch_size
359         dim = len(patch_size)
360         # todo rotation should be defined dynamically based on patch size (more isotropic patch sizes = more rotation)
361         if dim == 2:
362             do_dummy_2d_data_aug = False
363             # todo revisit this parametrization
364             if max(patch_size) / min(patch_size) > 1.5:
365                 rotation_for_DA = {
366                     'x': (-15. / 360 * 2. * np.pi, 15. / 360 * 2. * np.pi),
367                     'y': (0, 0),
368                     'z': (0, 0)
369                 }
370             else:
371                 rotation_for_DA = {
372                     'x': (-180. / 360 * 2. * np.pi, 180. / 360 * 2. * np.pi),
373                     'y': (0, 0),
374                     'z': (0, 0)
375                 }
376             mirror_axes = (0, 1)
377         elif dim == 3:
378             # todo this is not ideal. We could also have patch_size (64, 16, 128) in which case a full 180deg 2d rot would be
379             # order of the axes is determined by spacing, not image size
380             do_dummy_2d_data_aug = (max(patch_size) / patch_size[0]) > ANISO_THRESHOLD
381             if do_dummy_2d_data_aug:
382                 # why do we rotate 180 deg here all the time? We should also restrict it
383                 rotation_for_DA = {
384                     'x': (-180. / 360 * 2. * np.pi, 180. / 360 * 2. * np.pi),
385                     'y': (0, 0),
386                     'z': (0, 0)
387                 }
388             else:
389                 rotation_for_DA = {
390                     'x': (-30. / 360 * 2. * np.pi, 30. / 360 * 2. * np.pi),
391                     'y': (-30. / 360 * 2. * np.pi, 30. / 360 * 2. * np.pi),
392                     'z': (-30. / 360 * 2. * np.pi, 30. / 360 * 2. * np.pi),
393                 }
394             mirror_axes = (0, 1, 2)
395         else:
396             raise RuntimeError()
397
398         # todo this function is stupid. It doesn't even use the correct scale range (we keep things as they were in the
399         #  old nnunet for now)
400         initial_patch_size = get_patch_size(patch_size[-dim:],
401                                             *rotation_for_DA.values(),
402                                             (0.85, 1.25))
403         if do_dummy_2d_data_aug:
404             initial_patch_size[0] = patch_size[0]
405
406         self.print_to_log_file(f'do_dummy_2d_data_aug: {do_dummy_2d_data_aug}')
407         self.inference_allowed_mirroring_axes = mirror_axes
408
409         return rotation_for_DA, do_dummy_2d_data_aug, initial_patch_size, mirror_axes
```

**Summary:**

- `/nnunetv2/training/nnUNetTrainer/nnUNetTrainer.py` Line 358 - Line 399:

Deciding the rotation policy based on the **anisotropy** (how patch size along different dimensions may differ) of pre-configured patch size along different dimensions.

This is controlled by `do_dummy_2D_data_aug`: if the patch size has high anisotropy, then restrict the range of the rotation angle:

> For example, a 90-degree rotation for a rectangular patch with a high aspect ratio may not make too much sense: much of the expanded patch after rotation will be empty. Therefore their range of rotation along the dimensions with high anistropy would be restrained to +- 15 degree for 2D or +- 30 degrees for 3D.
>
> (As suggested by the author, order of the axes in `patch_size` is determined by spacing: smaller ones first.)

- `/nnunetv2/training/nnUNetTrainer/nnUNetTrainer.py` Line 400 - 403:

Computing the **largest actual patch size (final_shape)** you can get after possible rotations.

This calls `/nnunetv2/training/data_augmentation/compute_initial_patch_size.py`:

```python
def get_patch_size(final_patch_size, rot_x, rot_y, rot_z, scale_range):
    if isinstance(rot_x, (tuple, list)):
        rot_x = max(np.abs(rot_x))
    if isinstance(rot_y, (tuple, list)):
        rot_y = max(np.abs(rot_y))
    if isinstance(rot_z, (tuple, list)):
        rot_z = max(np.abs(rot_z))
    rot_x = min(90 / 360 * 2. * np.pi, rot_x)
    rot_y = min(90 / 360 * 2. * np.pi, rot_y)
    rot_z = min(90 / 360 * 2. * np.pi, rot_z)
    from batchgenerators.augmentations.utils import rotate_coords_3d, rotate_coords_2d
    coords = np.array(final_patch_size)
    final_shape = np.copy(coords)
    if len(coords) == 3:
        final_shape = np.max(np.vstack((np.abs(rotate_coords_3d(coords, rot_x, 0, 0)), final_shape)), 0)
        final_shape = np.max(np.vstack((np.abs(rotate_coords_3d(coords, 0, rot_y, 0)), final_shape)), 0)
        final_shape = np.max(np.vstack((np.abs(rotate_coords_3d(coords, 0, 0, rot_z)), final_shape)), 0)
    elif len(coords) == 2:
        final_shape = np.max(np.vstack((np.abs(rotate_coords_2d(coords, rot_x)), final_shape)), 0)
    final_shape /= min(scale_range)
    return final_shape.astype(int)
```

- Line 660 & 680: compress the channel dimension if only b,c, x, y, z → b,c*x,y,z

## b. Spatial transforms

### Overview

Most of transforms are implemented in [batchgenerators](https://github.com/MIC-DKFZ/batchgenerators).

In batchgenerators, callables are defined in `batchgenerators/batchgenerators/transforms/`

Actual implementations of transform functions are in `batchgenerators/batchgenerators/augmentations/`

Mostly rigid transforms. Elastic transforms are `disabled` by default (why?). Need to manually enable them. In `nnUNetv1`, DA configurations are detached from the trainer so you can directly change them in the configuration file to accommodate your own changes. In `v2` you probably need to derive a variant trainer class (as those in `nnunetv2/training/nnUNetTrainer/variants/`) to do so.

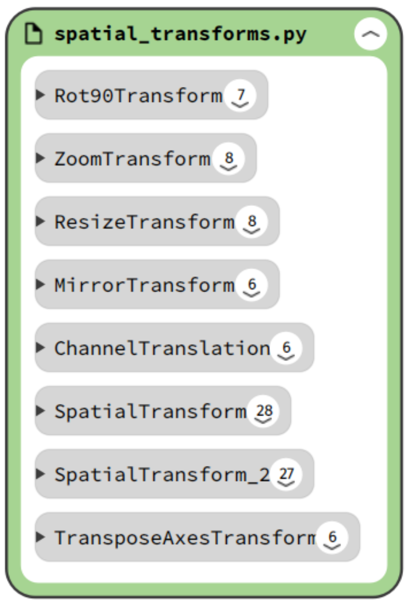
### Where it resides

- `/nnunetv2/training/nnUNetTrainer/nnUNetTrainer.py`

```
666        tr_transforms.append(SpatialTransform(
667            patch_size_spatial, patch_center_dist_from_border=None,
668            do_elastic_deform=False, alpha=(0, 0), sigma=(0, 0),
669            do_rotation=True, angle_x=rotation_for_DA['x'], angle_y=rotation_for_DA['y'], angle_z=rotation_for_DA['z'],
670            p_rot_per_axis=1,  # todo experiment with this
671            do_scale=True, scale=(0.7, 1.4),
672            border_mode_data="constant", border_cval_data=0, order_data=order_resampling_data,
673            border_mode_seg="constant", border_cval_seg=border_val_seg, order_seg=order_resampling_seg,
674            random_crop=False,  # random cropping is part of our dataloaders
675            p_el_per_sample=0, p_scale_per_sample=0.2, p_rot_per_sample=0.2,
676            independent_scale_for_each_axis=False  # todo experiment with this
677        ))
678
679        if do_dummy_2d_data_aug:
680            tr_transforms.append(Convert2DTo3DTransform())
```

This calls augmentation functions in batchgenerators

`/batchgenerators/transforms/spatial_transforms.py` → `/batchgenerators/augmentations/spatial_transformations.py`




**Key facts**:

- Use `data_key` and `label_key` to route images or labels to different operations (e.g. the order of interpolation. When you override their implementation, just keep the interface the same.)

```
249 | class SpatialTransform(AbstractTransform):

332     def __call__(self, **data_dict):
333         data = data_dict.get(self.data_key)
334         seg = data_dict.get(self.label_key)
335
```

- No elastic transforms by default.
- Handling with padded borders: In most cases for the default pre-processing configuration, the background/boundary of pre-processed images are set to 0. Therefore the padded values are also 0 (Line

672, also the original paper).

Padding values for segmentation masks are set to -1 (Line 673) and later changed back to 0 (Line 701).

- Resampling order = 3 for images, 0 (nearest-neighbor) for segmentation labels.

## c. Intensity/color augmentations

### Overview

These augmentation functions are also based on [batchgenerators], with low-level implementations on CPU (numpy, scikit-images, etc.)

Most of the augmentation functions are commonly used. A side reference is [bigaug] (https://ieeexplore.ieee.org/document/8995481) which experiments the effects of individual augmentation operations on segmentation performances (incl. those with out-of-domain test samples).

### Where it resides

- `/nnunetv2/training/nnUNetTrainer/nnUNetTrainer.py`

```
682         tr_transforms.append(GaussianNoiseTransform(p_per_sample=0.1))
683         tr_transforms.append(GaussianBlurTransform((0.5, 1.), different_sigma_per_channel=True, p_per_sample=0.2,
684                                                    p_per_channel=0.5))
685         tr_transforms.append(BrightnessMultiplicativeTransform(multiplier_range=(0.75, 1.25), p_per_sample=0.15))
686         tr_transforms.append(ContrastAugmentationTransform(p_per_sample=0.15))
687         tr_transforms.append(SimulateLowResolutionTransform(zoom_range=(0.5, 1), per_channel=True,
688                                                             p_per_channel=0.5,
689                                                             order_downsample=0, order_upsample=3, p_per_sample=0.25,
690                                                             ignore_axes=ignore_axes))
691         tr_transforms.append(GammaTransform((0.7, 1.5), True, True, retain_stats=True, p_per_sample=0.1))
692         tr_transforms.append(GammaTransform((0.7, 1.5), False, True, retain_stats=True, p_per_sample=0.3))
```

These augmentation callables are defined in `/batchgenerators/transforms/color_transforms.py` and `/batchgenerators/transforms/noise_transforms.py`

### Interesting facts

- Gamma transfroms: applied `twice` (line 691-692), one on original intensities, one on inverted intensities (see `batchgenerators/augmentations/color_augmentations.py` line 109-110)

```
107 def augment_gamma(data_sample, gamma_range=(0.5, 2), invert_image=False, epsilon=1e-7, per_channel=False,
108                   retain_stats: Union[bool, Callable[[], bool]] = False):
109     if invert_image:
110         data_sample = - data_sample
```

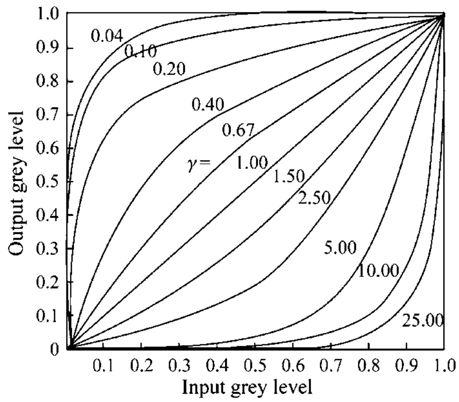



Illustration of Gamma transform functions with different \gamma ‘s. Source: [https://www.researchgate.net/figure/Gamma-transformation-of-grey-level_fig5_271632132](https://www.researchgate.net/figure/Gamma-transformation-of-grey-level_fig5_271632132)

## d. Additional augmentations for cascaded training (low-res)

### Overview

Cascaded training:

*“… a 3D U-Net cascade where first a 3D U-Net operates on **low resolution** images and then a second high-resolution 3D U-Net **refined** the predictions of the former (for 3D datasets with large image sizes only)…”*

The pitfall for the cascaded scheme is that (based on the understanding of my own): if the low-res model yields perfect fitting in the training stage, the high-res U-Net may degrade and learns `identity mapping` instead. As a result, in testing, when the prediction of low-res model is not ideal, the high-res model may not be sufficiently robust to make corrections. The author therefore introduces stochasticity to the ground truth for the low-res model.

Another way of interpreting: there may be multiple possible high-res segementations corresponding to the same low-res image.

To this end, the author makes perturbations: dilation, erosion, opening, closing, and noises to the low-resolution segmentation labels.

### Where it resides

- `/nnunetv2/training/nnUNetTrainer/nnUNetTrainer.py`

```python
        if is_cascaded:
            assert foreground_labels is not None, 'We need foreground_labels for cascade augmentations'
            tr_transforms.append(MoveSegAsOneHotToData(1, foreground_labels, 'seg', 'data'))
            tr_transforms.append(ApplyRandomBinaryOperatorTransform(
                channel_idx=list(range(-len(foreground_labels), 0)),
                p_per_sample=0.4,
                key="data",
                strel_size=(1, 8),
                p_per_label=1))
            tr_transforms.append(
                RemoveRandomConnectedComponentFromOneHotEncodingTransform(
                    channel_idx=list(range(-len(foreground_labels), 0)),
                    key="data",
                    p_per_sample=0.2,
                    fill_with_other_class_p=0,
                    dont_do_if_covers_more_than_x_percent=0.15))
```

### Key facts

- `MoveSegAsOneHotToData`: get the one_hot labels of the foreground.
- `ApplyRandomBinaryOperatorTransform`: perform random perturbations: dilation, erosion, closing, opening... to one-hot labels.
- `RemoveRandomConnectedComponetFromOne …`

  Randomly removing connected components with size smaller than a certain range and/or move that to other classes.

  Connected component operations come from [acvl_utils](https://github.com/MIC-DKFZ/acvl_utils/blob/master/acvl_utils/morphology/morphology_helper.py)

  Why? Simulating erroneous predictions for improving robustness (?my own guess).
- Additional steps for region-based training (merging multiple labels to one region, specific to BraTS):

```python
        if regions is not None:
            # the ignore label must also be converted
            tr_transforms.append(ConvertSegmentationToRegionsTransform(list(regions) + [ignore_label]
                                                                       if ignore_label is not None else regions,
                                                                       'target', 'target'))
```

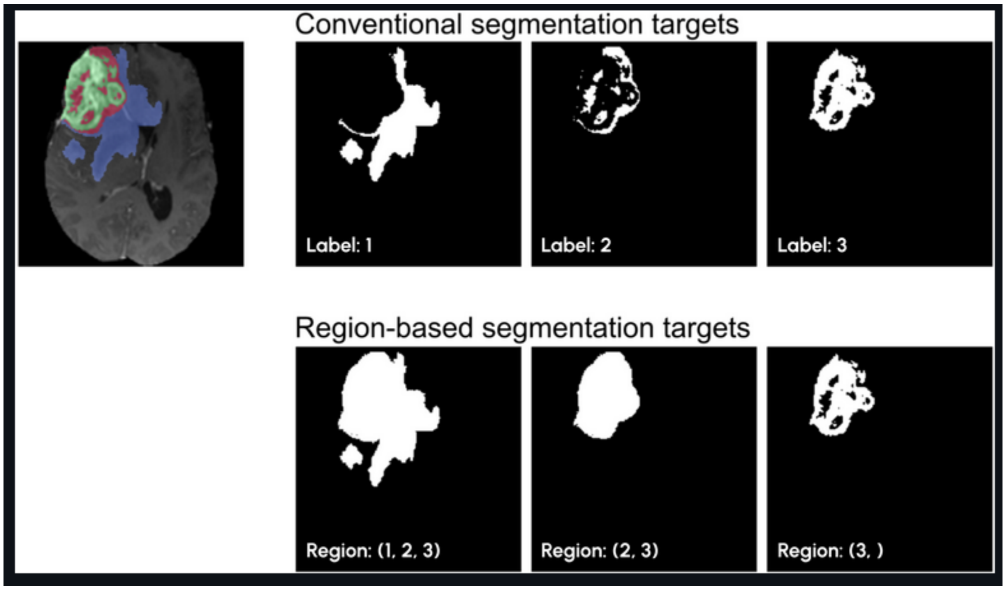



Illustration of region-based-training, source: https://github.com/MIC-DKFZ/nnUNet/blob/master/documentation/region_based_training.md

- Lower resolutions for deep supervision (in practice, very helpful for domain robustness). Implemented using `scikit-image`.

```
        if deep_supervision_scales is not None:
            tr_transforms.append(DownsampleSegForDSTransform2(deep_supervision_scales, 0, input_key='target',
                                                              output_key='target'))
```

- Output format: Now goes from numpy array to pytorch tensor.

```
        tr_transforms.append(NumpyToTensor(['data', 'target'], 'float'))
```

---

## 3. Comments

**Advantages**:

- Off-the-shelf augmentation configurations that work well for most cases, `little domain-specific knowledge` is needed for the dataset.

  Not sure how the developers come up with this set of configurations (It is a large search space).

- Suitable for most of `medical image segmentation challenges`: giving you a good-enough baseline in the shortest time without having you wasting additional efforts on tuning data augmentation configurations.

**Comments**:

- If stronger augmentations are needed (based on **adequate understanding of your datasets and tasks**), variants can be found in `nnunetv2/training/nnUNetTrainer/variants/data_augmentation/` ( `moreDA` and `insaneDA` for v1 code)

  (e.g. domain generalization-related challenges favors stronger/task-specific DA)
- Two levels of parallelization: home-made ones and those by native `numpy`. Be aware of that when sharing CPU resources with other users.
- The augmentation part `alone` may not always give you the same magic compared with nnUNet as a whole. (Same conclusion from the author of nnUNet).

---

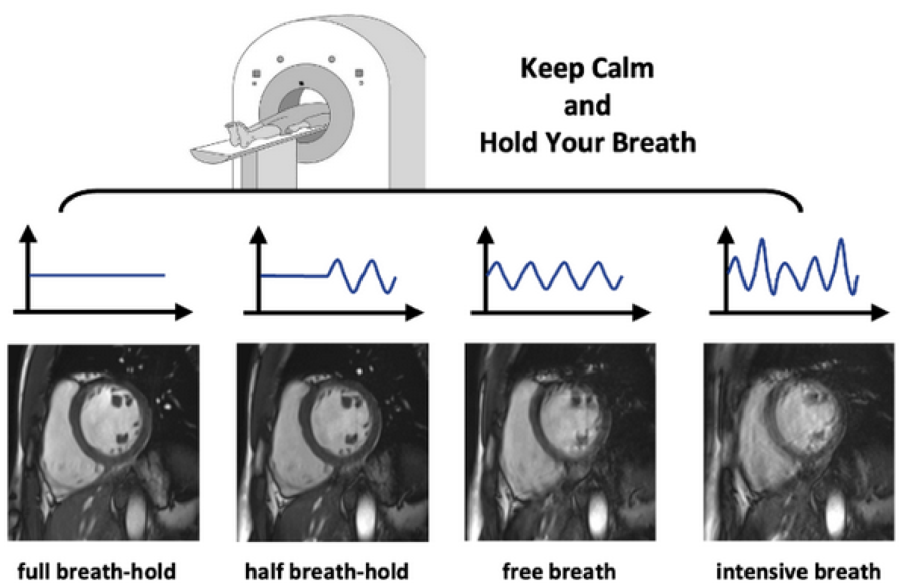
# 4. Beyond pre-configured data augmentations

Data augmentation is crucial for tasks that relate to `low-resource training`, `semi-supervised learning` , and `domain generalization`. In those cases, more advanced data augmentations can be tailored to the specific tasks.

Taking CMRxMotion and FeTA 2022 as examples. The winning solutions to them do employ nnUNet **but** importantly, they tailor data augmentation to their specific datasets and tasks.

The core idea for designing augmentation policy is to `simulate the diversity of the real underlying data`, based on generic assumptions about medical images and/or on your domain knowledge of the task at hand:

- [CMR-Motion](http://cmr.miccai.cloud/):

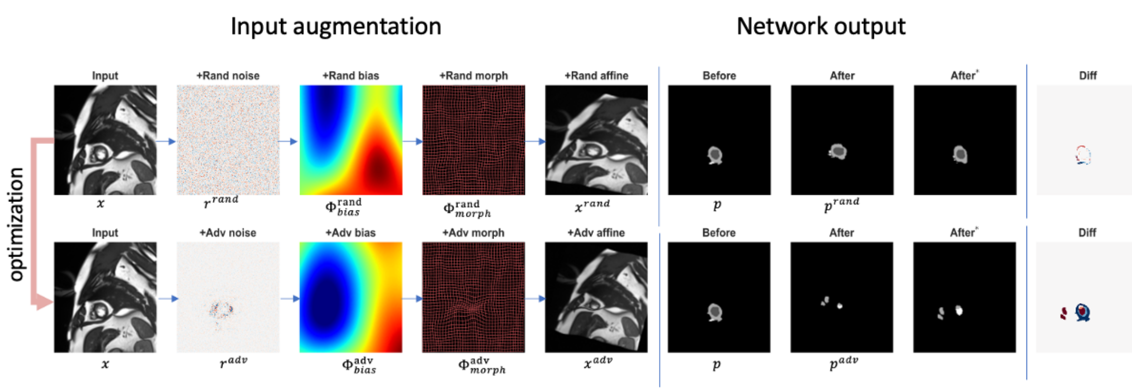


Task: Segmenting short axis CINE cardiac MRIs that may contain unexpected respiratory artifacts (see the right three examples). Essentially a `domain generalization` challenge.

Analysis: The performance bottleneck is to handle `unforeseen respiratory` in MRI. Beyond these motion patterns, other stochacities caused by imaging acquisition processes exist, and lead to a diverse test dataset that may be different from the training data.

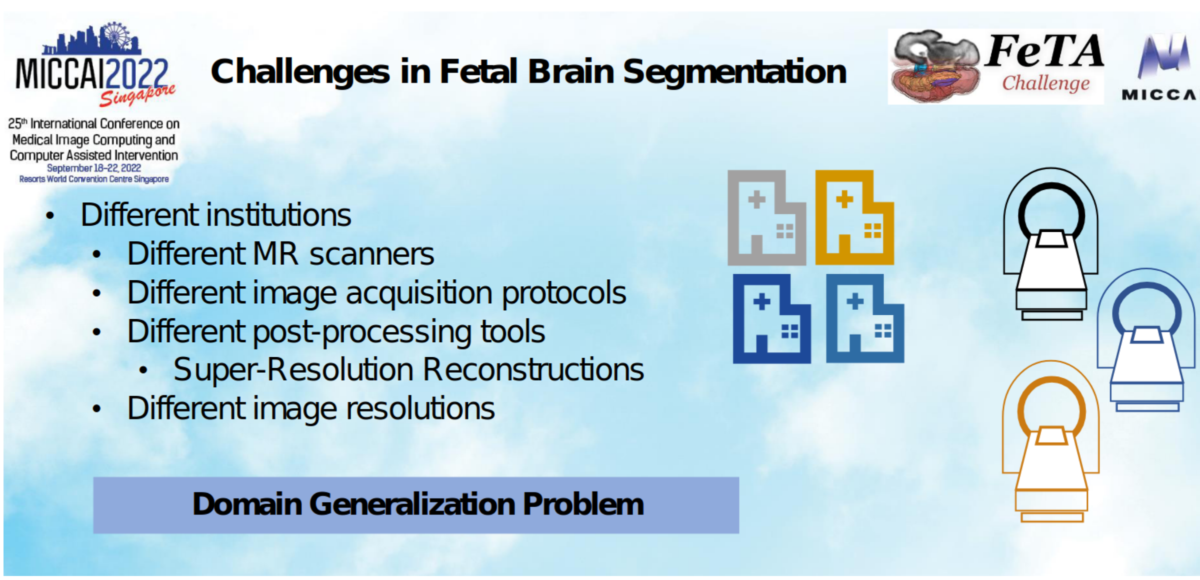
Wining solution (CUHK, two components):

1. augmentation techniques that `simulates the stochascities in the physical MRI acquisition process` [AdvChain](https://github.com/cherise215/advchain): bias field, noises, elastic deformations, etc. `Adversarial training` is used to optimize the augmentation parameters of data augmentation operations.




2. Pre-training on external datasets (ACDC, MSCMRSeg, etc), as allowed by the organizers

- [FeTA](https://feta.grand-challenge.org/)



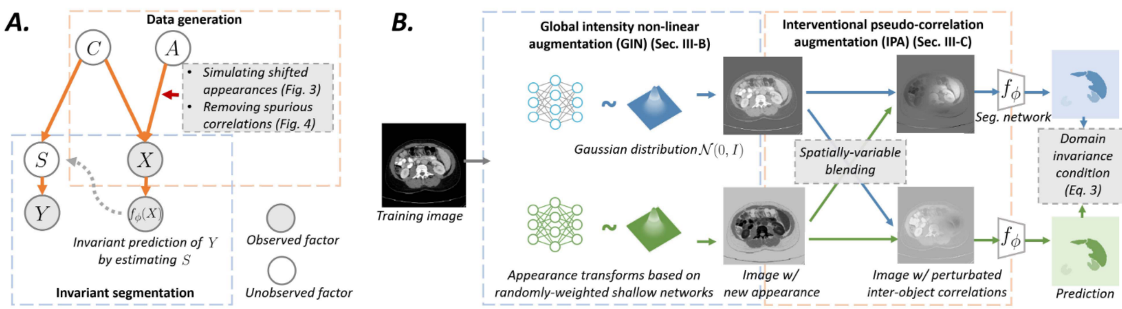
Task: To segment fetal brain MRIs that may come from `unforeseen` institutes. Essentially also a `domain generalization` challenge.

Analysis: The performance bottleneck is to handle `unforeseen intensity distributions`.

Winning solution (TUM & ICL): employing augmentation techniques that cover common distribution shifts in image intensities across datasets, these augmentation includes:

1. Conventional brightness/contrast/random Gamma, etc.
2. Bias-field (specific to MRI, same as described in [AdvChain])
3. Generic and strong domain generalization technique:

   [Causal-SD](https://github.com/cheng-01037/Causality-Medical-Image-Domain-Generalization )




   They also employ `ensembling strategies` among model trained with different augmentation policies, for better overall performance for both in-distribution data and various types of out-of-distribution data (as we do not know whether a test image is in-distribution or out-of-distribution, nor do we know the source of OOD).

Some of the ensemble rules are similar to those described in [Causal-SD].