# PyMIC

## *Release 0.4*

**HiLab**

**Feb 26, 2023**

# GETTING STARTED

PyMIC is a pytorch-based toolkit for medical image computing with annotation-efficient deep learning. PyMIC is developed to support learning with imperfect labels, including semi-supervised and weakly supervised learning, and learning with noisy annotations.

Check out the *Installation* section for install PyMIC, and go to the *Usage* section for understanding modules for the segmentation pipeline designed in PyMIC. Please follow PyMIC_examples to quickly start with using PyMIC.

**Note:** This project is under active development. It will be updated later.

# CHAPTER ONE

# INSTALLATION

Install PyMIC using pip (e.g., within a Python virtual environment):

```
pip install PYMIC
```

Alternatively, you can download or clone the code from GitHub and install PyMIC by

```
git clone https://github.com/HiLab-git/PyMIC
cd PyMIC
python setup.py install
```

PyMIC requires Python 3.6 (or higher) and depends on the following packages. They will be automatically installed when using *pip install*.

- pandas
- h5py
- NumPy
- scikit-image
- SciPy
- SimpleITK

CHAPTER

# TWO

# USAGE

This usage gives details of how to use PyMIC. Beginners can easily start with training a deep learning model with configure files. When you are more familar with the PyMIC pipeline, you can define your customized modules and reuse the remaining parts of the pipeline, with minimal workload.

## 2.1 Quick Start

### 2.1.1 Train and Test

PyMIC accepts a configuration file for runing. For example, to train a network for segmentation with full supervision, run the fullowing command:

```
pymic_train myconfig.cfg
```

After training, run the following command for testing:

```
pymic_test myconfig.cfg
```

**Tip:** We provide several examples in PyMIC_examples. Please run these examples to quickly start with using PyMIC.

### 2.1.2 Configuration File

PyMIC uses configuration files to specify the setting and parameters of a deep learning pipeline, so that users can reuse the code and minimize their workload. Users can use configuration files to config almost all the componets involved, such as dataset, network structure, loss function, optimizer, learning rate scheduler and post processing methods, etc.

**Note:** Genreally, the configuration file have four sections: `dataset`, `network`, `training` and `testing`.

The following is an example configuration file used for segmentation of lung from radiograph, which can be find in PyMIC_examples/segmentation/JSRT.

```
[dataset]
# tensor type (float or double)
tensor_type = float

task_type = seg
```

(continues on next page)

(continued from previous page)

```
root_dir  = ../../PyMIC_data/JSRT
train_csv = config/jsrt_train.csv
valid_csv = config/jsrt_valid.csv
test_csv  = config/jsrt_test.csv

train_batch_size = 4

# data transforms
train_transform = [NormalizeWithMeanStd, RandomCrop, LabelConvert, LabelToProbability]
valid_transform = [NormalizeWithMeanStd, LabelConvert, LabelToProbability]
test_transform  = [NormalizeWithMeanStd]

NormalizeWithMeanStd_channels = [0]
RandomCrop_output_size = [240, 240]

LabelConvert_source_list = [0, 255]
LabelConvert_target_list = [0, 1]


[network]
# this section gives parameters for network
# the keys may be different for different networks

# type of network
net_type = UNet2D

# number of class, required for segmentation task
class_num     = 2
in_chns       = 1
feature_chns  = [16, 32, 64, 128, 256]
dropout       = [0,  0,  0.3, 0.4, 0.5]
bilinear      = False
multiscale_pred = False

[training]
# list of gpus
gpus = [0]

loss_type     = DiceLoss

# for optimizers
optimizer     = Adam
learning_rate = 1e-3
momentum      = 0.9
weight_decay  = 1e-5

# for lr scheduler (MultiStepLR)
lr_scheduler  = MultiStepLR
lr_gamma      = 0.5
lr_milestones = [2000, 4000, 6000]

ckpt_save_dir    = model/unet
```

(continues on next page)

(continued from previous page)

```
ckpt_prefix = unet

# start iter
iter_start = 0
iter_max   = 8000
iter_valid = 200
iter_save  = 8000

[testing]
# list of gpus
gpus       = [0]

# checkpoint mode can be [0-latest, 1-best, 2-specified]
ckpt_mode         = 0
output_dir        = result/unet

# convert the label of prediction output
label_source = [0, 1]
label_target = [0, 255]
```

### 2.1.3 Evaluation

To evaluate a model's prediction results compared with the ground truth, use the `pymic_eval_seg` and `pymic_eval_cls` commands for segmentation and classfication tasks, respectively. Both of them accept a configuration file to specify the evaluation metrics, predicted results, ground truth and other information.

For example, for segmentation tasks, run:

```
pymic_eval_seg evaluation.cfg
```

The configuration file is like (an example from PyMIC_examples/seg_ssl/ACDC):

```
[evaluation]
metric_list = [dice, hd95]
label_list = [1,2,3]
organ_name = heart

ground_truth_folder_root  = ../../PyMIC_data/ACDC/preprocess
segmentation_folder_root  = result/unet2d_urpc
evaluation_image_pair     = config/data/image_test_gt_seg.csv
```

See *pymic.util.evaluation_seg.evaluation* for details of the configuration required.

For classification tasks, run:

```
pymic_eval_cls evaluation.cfg
```

The configuration file is like (an example from PyMIC_examples/classification/CHNCXR):

```
[evaluation]
metric_list = [accuracy, auc]
ground_truth_csv = config/cxr_test.csv
```

(continues on next page)

(continued from previous page)

```
predict_csv   = result/resnet18.csv
predict_prob_csv   = result/resnet18_prob.csv
```

See *pymic.util.evaluation_cls.main* for details of the configuration required.

## 2.2 Fully Supervised Learning

### 2.2.1 SegmentationAgent

*pymic.net_run.agent_seg.SegmentationAgent* is the general class used for training and inference of deep learning models. You just need to specify a configuration file to initialize an instance of that class. An example code to use it is:

```
from pymic.util.parse_config import *
from pymic.net_run.agent_seg import SegmentationAgent

config_name = "a_config_file.cfg"
config   = parse_config(config_name)
config   = synchronize_config(config)
stage    = "train"  # or "test"
agent    = SegmentationAgent(config, stage)
agent.run()
```

The above code will use the dataset, network and loss function, etc specifcied in the configuration file for running.

**Tip:** If you use the built-in modules such as `UNet` and `Dice` + `CrossEntropy` loss for segmentation, you don't need to write the above code. Just just use the `pymic_train` command. See examples in PyMIC_examples/segmentation/.

### 2.2.2 Dataset

PyMIC provides two types of datasets for loading images from disk to memory: `NiftyDataset` and `H5DataSet`. `NiftyDataset` is designed for 2D and 3D images in common formats such as png, jpeg, bmp and nii.gz. `H5DataSet` is used for hdf5 data that are more efficient to load.

To use `NiftyDataset`, users need to specify the root path of the dataset and the csv file storing the image and label file names. The configurations include the following items:

- `tensor_type`: data type for tensors. Should be `float` or `double`.
- `task_type`: should be `seg` for segmentation tasks.
- `root_dir` (string): the root dir of dataset.
- `modal_num` (int, default is 1): modalities number. For images with N modalities, each modality should be saved in an indepdent file.
- `train_csv` (string): the path of csv file for training set.
- `valid_csv` (string): the path of csv file for validation set.
- `test_csv` (string): the path of csv file for testing set.
- `train_batch_size` (int): the batch size for training set.

- `valid_batch_size` (int, optional): the batch size for validation set. The defualt value is set as `train_batch_size`.
- `test_batch_size` (int, optional): the batch size for testing set. The defualt value is 1.

The csv file should have at least two columns (fields), one for `image` and the other for `label`. If the input image have multiple modalities with each modality saved in a single file, then the csv file should have N + 1 columns, where the first N columns are for the N modalities, and the last column is for the label. The following is an example for configuration of dataset.

```
[dataset]
# tensor type (float or double)
tensor_type = float
task_type = seg
root_dir  = ../../PyMIC_data/JSRT
train_csv = config/jsrt_train.csv
valid_csv = config/jsrt_valid.csv
test_csv  = config/jsrt_test.csv
train_batch_size = 4
```

To use your own dataset, you can define a dataset as a child class of `NiftyDataset`, `H5DataSet`, or `torch.utils.data.Dataset` , and use `SegmentationAgent.set_datasets()` to set the customized datasets. For example:

```
from torch.utils.data import Dataset
from pymic.net_run.agent_seg import SegmentationAgent

class MyDataset(Dataset):
   ...
   # define your custom dataset here

trainset, valset, testset = MyDataset(...), MyDataset(...), MyDataset(...)
agent = SegmentationAgent(config, stage)
agent.set_datasets(trainset, valset, testset)
agent.run()
```

### 2.2.3 Transforms

Several transforms are defined in PyMIC to preprocess or augment the data before sending it to the network. The `TransformDict` in *pymic.transform.trans_dict* lists all the built-in transforms supported in PyMIC.

In the configuration file, users can specify the transforms required for training, validation and testing data, respectively. The parameters of each tranform class should also be provided, such as following:

```
# data transforms
train_transform = [Pad, RandomRotate, RandomCrop, RandomFlip, NormalizeWithMeanStd,␣
↪GammaCorrection, GaussianNoise, LabelToProbability]
valid_transform = [NormalizeWithMeanStd, Pad, LabelToProbability]
test_transform  = [NormalizeWithMeanStd, Pad]

# the inverse transform will be enabled during testing
Pad_output_size = [8, 256, 256]
Pad_ceil_mode   = False
Pad_inverse     = True
```

(continues on next page)

(continued from previous page)

```
RandomRotate_angle_range_d = [-90, 90]
RandomRotate_angle_range_h = None
RandomRotate_angle_range_w = None

RandomCrop_output_size = [6, 192, 192]
RandomCrop_foreground_focus = False
RandomCrop_foreground_ratio = None
Randomcrop_mask_label       = None

RandomFlip_flip_depth  = False
RandomFlip_flip_height = True
RandomFlip_flip_width  = True

NormalizeWithMeanStd_channels = [0]

GammaCorrection_channels  = [0]
GammaCorrection_gamma_min = 0.7
GammaCorrection_gamma_max = 1.5

GaussianNoise_channels = [0]
GaussianNoise_mean     = 0
GaussianNoise_std      = 0.05
GaussianNoise_probability = 0.5
```

For spatial transforms, you can specify whether an inverse transform is enabled or not. Setting the inverse flag as True will transform the prediction output inversely during testing, such as `Pad_inverse = True` shown above. If you want to make images with different shapes to have the same shape before testing, then the correspoinding transform's inverse flag can be set as True, so that the prediction output will be transformed back to the original image space. This is also useful for test time augmentation.

You can also define your own transform operations. To integrate your customized transform to the PyMIC pipeline, just add it to the `TransformDict`, and you can also specify the parameters via a configuration file for the customized transform. The following is some example code for this:

```
from pymic.transform.trans_dict import TransformDict
from pymic.transform.abstract_transform import AbstractTransform
from pymic.net_run.agent_seg import SegmentationAgent

# customized transform
class MyTransform(AbstractTransform):
   def __init__(self, params):
      super(MyTransform, self).__init__(params)
      ...

   def __call__(self, sample):
      ...

   def  inverse_transform_for_prediction(self, sample):
      ...

my_trans_dict = TransformDict
my_trans_dict["MyTransform"] = MyTransform
agent = SegmentationAgent(config, stage)
```

(continues on next page)

(continued from previous page)

```
agent.set_transform_dict(my_trans_dict)
agent.run()
```

### 2.2.4 Networks

The configuration file has a `network` section to specify the network's type and hyper-parameters. For example, the following is a configuration for using `2DUNet`:

```
[network]
net_type = UNet2D
# Parameters for UNet2D
class_num      = 2
in_chns        = 1
feature_chns   = [16, 32, 64, 128, 256]
dropout        = [0,  0,  0.3, 0.4, 0.5]
bilinear       = False
multiscale_pred = False
```

The `SegNetDict` in *pymic.net.net_dict_seg* lists all the built-in network structures currently implemented in PyMIC.

You can also define your own networks. To integrate your customized network to the PyMIC pipeline, just call `set_network()` of `SegmentationAgent`. The following is some example code for this:

```
import torch.nn as nn
from pymic.net_run.agent_seg import SegmentationAgent

# customized network
class MyNetwork(nn.Module):
   def __init__(self, params):
      super(MyNetwork, self).__init__()
      ...

   def forward(self, x):
      ...

net = MyNetwork(params)
agent = SegmentationAgent(config, stage)
agent.set_network(net)
agent.run()
```

### 2.2.5 Loss Functions

The setting of loss function is in the `training` section of the configuration file, where the loss function name and hyper-parameters should be provided. The `SegLossDict` in *pymic.loss.loss_dict_seg* lists all the built-in loss functions currently implemented in PyMIC.

The following is an example of the setting of loss:

```
loss_type = DiceLoss
loss_softmax = True
```

Note that PyMIC supports using a combination of loss functions. Just set `loss_type` as a list of loss functions, and use `loss_weight` to specify the weight of each loss, such as the following:

```
loss_type     = [DiceLoss, CrossEntropyLoss]
loss_weight   = [0.5, 0.5]
```

You can also define your own loss functions. To integrate your customized loss function to the PyMIC pipeline, just add it to the `SegLossDict`, and you can also specify the parameters via a configuration file for the customized loss. The following is some example code for this:

```
from pymic.loss.loss_dict_seg import SegLossDict
from pymic.net_run.agent_seg import SegmentationAgent

# customized loss
class MyLoss(nn.Module):
   def __init__(self, params = None):
      super(MyLoss, self).__init__()
      ...

   def forward(self, loss_input_dict):
      ...

my_loss_dict = SegLossDict
my_loss_dict["MyLoss"] = MyLoss
agent = SegmentationAgent(config, stage)
agent.set_loss_dict(my_loss_dict)
agent.run()
```

### 2.2.6 Training Options

In addition to the loss fuction, users can specify several training options in the `training` section of the configuration file.

#### Itreations

For training iterations, the following parameters need to be specified in the configuration file:

- `iter_max`: the maximal allowed iteration for training.
- `iter_valid`: if the value is K, it means evaluating the performance on the validaiton set for every K steps.
- `iter_save`: The iteations for saving the model. If the value is k, it means the model will be saved every k iterations. It can also be a list of integer numbers, which specifies the iterations to save the model.
- `early_stop_patience`: if the value is k, it means the training will stop when the performance on the validation set does not improve for k iteations.

#### Optimizer

For optimizer, users need to set `optimizer`, `learning_rate`, `momentum` and `weight_decay`. The built-in optimizers include `SGD`, `Adam`, `SparseAdam`, `Adadelta`, `Adagrad`, `Adamax`, `ASGD`, `LBFGS`, `RMSprop` and `Rprop` that are implemented in *torch.optim*.

You can also use customized optimizers via *SegmentationAgent.set_optimizer()*.

#### Learning Rate Scheduler

The current built-in learning rate schedulers are `ReduceLROnPlateau` and `MultiStepLR`, which can be specified in `lr_scheduler` in the configuration file.

Parameters related to `ReduceLROnPlateau` include `lr_gamma`. Parameters related to `MultiStepLR` include `lr_gamma` and `lr_milestones`.

You can also use customized lr schedulers via *SegmentationAgent.set_scheduler()*.

#### Other Options

Other options for training include:

- `gpus`: a list of GPU index for training the model. If the length is larger than one (such as [0, 1]), it means the model will be trained on multiple GPUs parallelly.
- `deterministic` (bool, default is True): set the training deterministic or not.
- `random_seed` (int, optioinal): the random seed customized by user. Default value is 1.
- `ckpt_save_dir`: the path to the folder for saving the trained models.
- `ckpt_prefix`: the prefix of the name to save the checkpoints.

### 2.2.7 Inference Options

There are several options for inference after training the model. You can also select the GPUs for testing, enable sliding window inference or inference with test-time augmentation, etc. The following is a list of options availble for inference:

- `gpus`: a list of GPU index. Atually, only the first GPU in the list is used.
- `evaluation_mode` (bool, default is True): set the model to evaluation mode or not.
- `test_time_dropout` (bool, default is False): use test-time dropout or not.
- `ckpt_mode` (int): which checkpoint is used. 0–the last checkpoint; 1–the checkpoint with the best performance on the validation set; 2–a specified checkpoint.
- `ckpt_name` (string, optinal): the full path to the checkpoint if ckpt_mode = 2.
- `post_process` (string, default is None): the post process method after inference. The current available post processing is *`pymic.util.post_process.PostKeepLargestComponent`*. Uses can also specify customized post process methods via *SegmentationAgent.set_postprocessor()*.
- `sliding_window_enable` (bool, default is False): use sliding window for inference or not.
- `sliding_window_size` (optinal): a list for sliding window size when sliding_window_enable = True.
- `sliding_window_stride` (optinal): a list for sliding window stride when sliding_window_enable = True.
- `tta_mode` (int, default is 0): the mode for Test Time Augmentation (TTA). 0–not using TTA; 1–using TTA based on horizontal and vertical flipping.

- `output_dir` (string): the dir to save the prediction output.
- `ignore_dir` (bool, default is True): if the input image name has a /, it will be replaced with _ in the output file name.
- `save_probability` (bool, default is False): save the output probability for each class.
- `label_source` (list, default is None): a list of label to be converted after prediction. For example, *label_source* = [0, 1] and *label_target* = [0, 255] will convert label value from 1 to 255.
- `label_target` (list, default is None): a list of label after conversion. Use this with *label_source*.
- `filename_replace_source` (string, default is None): the substring in the filename will be replaced with a new substring specified by *filename_replace_target*.
- `filename_replace_target` (string, default is None): work with *filename_replace_source*.

## 2.3 Semi-Supervised Learning

### 2.3.1 SSL Configurations

In the configuration file for semi-supervised segmentation, in addition to those used in fully supervised learning, there are some items specificalized for semi-supervised learning.

Users should provide values for the following items in `dataset` section of the configuration file:

- `supervise_type` (string): The value should be "*semi_sup*".
- `train_csv_unlab` (string): the csv file for unlabeled dataset. Note that `train_csv` is only used for labeled dataset.
- `train_batch_size_unlab` (int): the batch size for unlabeled dataset. Note that *train_batch_size* means the batch size for the labeled dataset.
- `train_transform_unlab` (list): a list of transforms used for unlabeled data.

The following is an example of the `dataset` section for semi-supervised learning:

```
...

tensor_type    = float
task_type      = seg
supervise_type = semi_sup

root_dir  = ../../PyMIC_data/ACDC/preprocess/
train_csv = config/data/image_train_r10_lab.csv
train_csv_unlab = config/data/image_train_r10_unlab.csv
valid_csv = config/data/image_valid.csv
test_csv  = config/data/image_test.csv

train_batch_size = 4
train_batch_size_unlab = 4

# data transforms
train_transform = [Pad, RandomRotate, RandomCrop, RandomFlip, NormalizeWithMeanStd,␣
↪GammaCorrection, GaussianNoise, LabelToProbability]
train_transform_unlab = [Pad, RandomRotate, RandomCrop, RandomFlip, NormalizeWithMeanStd,
```

(continues on next page)

(continued from previous page)

```
↪ GammaCorrection, GaussianNoise]
valid_transform       = [NormalizeWithMeanStd, Pad, LabelToProbability]
test_transform        = [NormalizeWithMeanStd, Pad]
...
```

In addition, there is a `semi_supervised_learning` section that is specifically designed for SSL methods. In that section, users need to specify the `method_name` and configurations related to the SSL method. For example, the correspoinding configuration for CPS is:

```
...
[semi_supervised_learning]
method_name    = CPS
regularize_w   = 0.1
rampup_start   = 1000
rampup_end     = 20000
...
```

**Note:** The configuration items vary with different SSL methods. Please refer to the API of each built-in SSL method for details of the correspoinding configuration. See examples in PyMIC_examples/seg_ssl/.

### 2.3.2 Built-in SSL Methods

*pymic.net_run.semi_sup.ssl_abstract.SSLSegAgent* is the abstract class used for semi-supervised learning. The built-in SSL methods are child classes of *SSLSegAgent*. The available SSL methods implemnted in PyMIC are listed in *pymic.net_run.semi_sup.SSLMethodDict*, and they are:

- `EntropyMinimization`: (NeurIPS 2005) Using entorpy minimization to regularize unannotated samples.
- `MeanTeacher`: (NeurIPS 2017) Use self-ensembling mean teacher to supervise the student model on unannotated samples.
- `UAMT`: (MICCAI 2019) Uncertainty aware mean teacher.
- `CCT`: (CVPR 2020) Cross-consistency training.
- `CPS`: (CVPR 2021) Cross-pseudo supervision.
- `URPC`: (MIA 2022) Uncertainty rectified pyramid consistency.

### 2.3.3 Customized SSL Methods

PyMIC alo supports customizing SSL methods by inheriting the *SSLSegAgent* class. You may only need to rewrite the *training()* method and reuse most part of the existing pipeline, such as data loading, validation and inference methods. For example:

```
from pymic.net_run.semi_sup import SSLSegAgent

class MySSLMethod(SSLSegAgent):
  def __init__(self, config, stage = 'train'):
      super(MySSLMethod, self).__init__(config, stage)
      ...
```

(continues on next page)

(continued from previous page)

```
  def training(self):
      ...

agent = MySSLMethod(config, stage)
agent.run()
```

You may need to check the source code of built-in SSL methods to be more familar with how to implement your own SSL method.

## 2.4 Weakly-Supervised Learning

**Note:** Currently, the weakly supervised methods supported by PyMIC are only for learning from partial annotations, such scribble-based annotation. Learning from image-level or point annotations may involve several training stages and will be considered in the future.

### 2.4.1 WSL Configurations

In the configuration file for weakly supervised learning, in addition to those used in fully supervised learning, there are some items specificalized for weakly-supervised learning.

First, `supervise_type` should be set as "*weak_sup*" in the `dataset` section.

Second, in the `train_transform` list, a special transform named *PartialLabelToProbability* should be used to transform patial labels into a one-hot probability map and a weighting map of pixels (i.e., the weight of a pixel is 1 if labeled and 0 otherwise). The patial cross entropy loss on labeled pixels is actually implemented by a weighted cross entropy loss. The loss setting is *loss_type* = *CrossEntropyLoss*.

Thirdly, there is a `weakly_supervised_learning` section that is specifically designed for WSL methods. In that section, users need to specify the `method_name` and configurations related to the WSL method. For example, the correspoinding configuration for GatedCRF is:

```
[dataset]
...
supervise_type = weak_sup
root_dir  = ../../PyMIC_data/ACDC/preprocess
train_csv = config/data/image_train.csv
valid_csv = config/data/image_valid.csv
test_csv  = config/data/image_test.csv

train_batch_size = 4

# data transforms
train_transform = [Pad, RandomCrop, RandomFlip, NormalizeWithMeanStd,␣
↪PartialLabelToProbability]
valid_transform = [NormalizeWithMeanStd, Pad, LabelToProbability]
test_transform  = [NormalizeWithMeanStd, Pad]
...

[network]
```

(continues on next page)

(continued from previous page)

```
...

[training]
...
loss_type     = CrossEntropyLoss
...

[weakly_supervised_learning]
method_name    = GatedCRF
regularize_w   = 0.1
rampup_start   = 2000
rampup_end     = 15000
GatedCRFLoss_W0     = 1.0
GatedCRFLoss_XY0    = 5
GatedCRFLoss_rgb    = 0.1
GatedCRFLoss_W1     = 1.0
GatedCRFLoss_XY1    = 3
GatedCRFLoss_Radius = 5

[testing]
...
```

**Note:** The configuration items vary with different WSL methods. Please refer to the API of each built-in WSL method for details of the correspoinding configuration. See examples in PyMIC_examples/seg_wsl/.

### 2.4.2 Built-in WSL Methods

*pymic.net_run.weak_sup.wsl_abstract.WSLSegAgent* is the abstract class used for weakly-supervised learning. The built-in WSL methods are child classes of *WSLSegAgent*. The available WSL methods implemnted in PyMIC are listed in *pymic.net_run.weak_sup.WSLMethodDict*, and they are:

- `EntropyMinimization`: (NeurIPS 2005) Using entorpy minimization to regularize unannotated pixels.
- `GatedCRF`: (arXiv 2019) Use gated CRF to regularize unannotated pixels.
- `TotalVariation`: (arXiv 2022) Use Total Variation to regularize unannotated pixels.
- `MumfordShah`: (TIP 2020) Use Mumford Shah loss to regularize unannotated pixels.
- `USTM`: (PR 2022) Adapt USTM with transform-consistency regularization.
- `DMPLS`: (MICCAI 2022) Dynamically mixed pseudo label supervision

### 2.4.3 Customized WSL Methods

PyMIC alo supports customizing WSL methods by inheriting the *WSLSegAgent* class. You may only need to rewrite the *training()* method and reuse most part of the existing pipeline, such as data loading, validation and inference methods. For example:

```
from pymic.net_run.weak_sup import WSLSegAgent

class MyWSLMethod(WSLSegAgent):
  def __init__(self, config, stage = 'train'):
      super(MyWSLMethod, self).__init__(config, stage)
      ...

  def training(self):
      ...

agent = MyWSLMethod(config, stage)
agent.run()
```

You may need to check the source code of built-in WSL methods to be more familar with how to implement your own WSL method.

## 2.5 Noisy Label Learning

**Note:** Some NLL methods only use noise-robust loss functions without complex training process, and just combining the standard `SegmentationAgent` with such loss function works for training.

### 2.5.1 NLL Configurations

In the configuration file for noisy label learning, in addition to those used in standard fully supervised learning, there is a `noisy_label_learning` section that is specifically designed for NLL methods. In that section, users need to specify the `method_name` and configurations related to the NLL method. `supervise_type` should be set as "*noisy_label*" in the `dataset` section.

For example, the correspoinding configuration for CoTeaching is:

```
[dataset]
...
supervise_type = noisy_label
...

[network]
...

[training]
...

[noisy_label_learning]
method_name  = CoTeaching
co_teaching_select_ratio  = 0.8
```

(continues on next page)

(continued from previous page)

```
rampup_start = 1000
rampup_end   = 8000

[testing]
...
```

**Note:** The configuration items vary with different NLL methods. Please refer to the API of each built-in NLL method for details of the correspoinding configuration. See examples in PyMIC_examples/seg_nll/.

### 2.5.2 Built-in NLL Methods

Some NLL methods only use noise-robust loss functions. They are used with a standard fully supervised training paradigm. Just set `supervise_type` = *fully_sup*, and use `loss_type` to one of them in the configuration file:

- `GCELoss`: (NeurIPS 2018) Generalized cross entropy loss.
- `MAELoss`: (AAAI 2017) Mean Absolute Error loss.
- `NRDiceLoss`: (TMI 2020) Noise-robust Dice loss.

The other NLL methods are implemented in child classes of *`pymic.net_run.agent_seg.SegmentationAgent`*, and they are:

- `CLSLSR`: (MICCAI 2020) Confident learning with spatial label smoothing regularization.
- `CoTeaching`: (NeurIPS 2018) Co-teaching between two networks for learning from noisy labels.
- `TriNet`: (MICCAI 2020) Tri-network combined with sample selection.
- `DAST`: (JBHI 2022) Divergence-aware selective training.

### 2.5.3 Customized NLL Methods

PyMIC alo supports customized NLL methods by inheriting the *SegmentationAgent* class. You may only need to rewrite the *training()* method and reuse most part of the existing pipeline, such as data loading, validation and inference methods. For example:

```
from pymic.net_run.agent_seg import SegmentationAgent

class MyNLLMethod(SegmentationAgent):
  def __init__(self, config, stage = 'train'):
      super(MyNLLMethod, self).__init__(config, stage)
      ...

  def training(self):
      ...

agent = MyNLLMethod(config, stage)
agent.run()
```

You may need to check the source code of built-in NLL methods to be more familar with how to implement your own NLL method.

In addition, if you want to design a new noise-robust loss fucntion, just follow *Fully Supervised Learning* to impelement and use the customized loss.

# CHAPTER THREE

# API

## 3.1 pymic.io package

### 3.1.1 Submodules

### 3.1.2 pymic.io.h5_dataset module

**class** pymic.io.h5_dataset.**H5DataSet**(*root_dir*, *sample_list_name*, *transform=None*)

Bases: `Dataset`

Dataset for loading images stored in h5 format. It generates 4D tensors with dimention order [C, D, H, W] for 3D images, and 3D tensors with dimention order [C, H, W] for 2D images

**Args:**

root_dir (str): thr root dir of images.

sample_list_name (str): a file name for sample list.

tranform (list): A list of transform objects applied on a sample.

**class** pymic.io.h5_dataset.**TwoStreamBatchSampler**(*primary_indices*, *secondary_indices*, *batch_size*, *secondary_batch_size*)

Bases: `Sampler`

Iterate two sets of indices

An 'epoch' is one iteration through the primary indices. During the epoch, the secondary indices are iterated through as many times as needed.

pymic.io.h5_dataset.**grouper**(*iterable*, *n*)

Collect data into fixed-length chunks or blocks

pymic.io.h5_dataset.**iterate_eternally**(*indices*)

pymic.io.h5_dataset.**iterate_once**(*iterable*)

### 3.1.3 pymic.io.image_read_write module

pymic.io.image_read_write.**load_image_as_nd_array**(*image_name*)

Load an image and return a 4D array with shape [C, D, H, W], or 3D array with shape [C, H, W].

**Parameters**
**filename** – (str) The input file name

**Returns**
A dictionay storing data array, origin, spacing and direction.

pymic.io.image_read_write.**load_nifty_volume_as_4d_array**(*filename*)

Read a nifty image and return a dictionay storing data array, origin, spacing and direction.

output['data_array'] 4D array with shape [C, D, H, W];

output['spacing'] A list of spacing in z, y, x axis;

output['direction'] A 3x3 matrix for direction.

**Parameters**
**filename** – (str) The input file name

**Returns**
A dictionay storing data array, origin, spacing and direction.

pymic.io.image_read_write.**load_rgb_image_as_3d_array**(*filename*)

Read an RGB image and return a dictionay storing data array, origin, spacing and direction.

output['data_array'] 3D array with shape [D, H, W];

output['spacing'] a list of spacing in z, y, x axis;

output['direction'] a 3x3 matrix for direction.

**Parameters**
**filename** – (str) The input file name

**Returns**
A dictionay storing data array, origin, spacing and direction.

pymic.io.image_read_write.**rotate_nifty_volume_to_LPS**(*filename_or_image_dict*, *origin=None*, *direction=None*)

Rotate the axis of a 3D volume to LPS

**Parameters**

- **filename_or_image_dict** – (str) Filename of the nifty file (str) or image dictionary returned by load_nifty_volume_as_4d_array. If supplied with the former, the flipped image data will be saved to override the original file. If supplied with the later, only flipped image data will be returned.
- **origin** – (list/tuple) The origin of the image.
- **direction** – (list or tuple) The direction of the image.

**Returns**
A dictionary for image data and meta info, with `data_array`, `origin`, `direction` and `spacing`.

pymic.io.image_read_write.**save_array_as_nifty_volume**(*data*, *image_name*, *reference_name=None*)

Save a numpy array as nifty image

**Parameters**

- **data** – (numpy.ndarray) A numpy array with shape [Depth, Height, Width].
- **image_name** – (str) The ouput file name.
- **reference_name** – (str) File name of the reference image of which meta information is used.

pymic.io.image_read_write.**save_array_as_rgb_image**(*data*, *image_name*)

Save a numpy array as rgb image.

**Parameters**

- **data** – (numpy.ndarray) A numpy array with shape [3, H, W] or [H, W, 3] or [H, W].
- **image_name** – (str) The output file name.

pymic.io.image_read_write.**save_nd_array_as_image**(*data*, *image_name*, *reference_name=None*)

Save a 3D or 2D numpy array as medical image or RGB image

**Parameters**

- **data** – (numpy.ndarray) A numpy array with shape [3, H, W] or [H, W, 3] or [H, W].
- **reference_name** – (str) File name of the reference image of which meta information is used.

### 3.1.4 pymic.io.nifty_dataset module

class pymic.io.nifty_dataset.**ClassificationDataset**(*root_dir*, *csv_file*, *modal_num=1*, *class_num=2*, *with_label=False*, *transform=None*)

Bases: `NiftyDataset`

Dataset for loading images for classification. It generates 4D tensors with dimention order [C, D, H, W] for 3D images, and 3D tensors with dimention order [C, H, W] for 2D images.

**Parameters**

- **root_dir** – (str) Directory with all the images.
- **csv_file** – (str) Path to the csv file with image names.
- **modal_num** – (int) Number of modalities.
- **class_num** – (int) Class number of the classificaiton task.
- **with_label** – (bool) Load the data with segmentation ground truth or not.
- **transform** – (list) List of transforms to be applied on a sample. The built-in transforms can listed in `pymic.transform.trans_dict`.

class pymic.io.nifty_dataset.**NiftyDataset**(*root_dir*, *csv_file*, *modal_num=1*, *with_label=False*, *transform=None*)

Bases: `Dataset`

Dataset for loading images for segmentation. It generates 4D tensors with dimention order [C, D, H, W] for 3D images, and 3D tensors with dimention order [C, H, W] for 2D images.

**Parameters**

- **root_dir** – (str) Directory with all the images.
- **csv_file** – (str) Path to the csv file with image names.
- **modal_num** – (int) Number of modalities.

- **with_label** – (bool) Load the data with segmentation ground truth or not.
- **transform** – (list) List of transforms to be applied on a sample. The built-in transforms can listed in *pymic.transform.trans_dict*.

### 3.1.5 Module contents

## 3.2 pymic.layer package

### 3.2.1 Submodules

### 3.2.2 pymic.layer.activation module

pymic.layer.activation.**get_acti_func**(*acti_func*, *params*)

### 3.2.3 pymic.layer.convolution module

**class** pymic.layer.convolution.**ConvolutionLayer**(*in_channels*, *out_channels*, *kernel_size*, *dim=3*, *stride=1*, *padding=0*, *dilation=1*, *conv_group=1*, *bias=True*, *norm_type='batch_norm'*, *norm_group=1*, *acti_func=None*)

Bases: Module

A compose layer with the following components: convolution -> (batch_norm / layer_norm / group_norm / instance_norm) -> (activation) -> (dropout) Batch norm and activation are optional.

**Parameters**

- **in_channels** – (int) The input channel number.
- **out_channels** – (int) The output channel number.
- **kernel_size** – The size of convolution kernel. It can be either a single int or a tupe of two or three ints.
- **dim** – (int) The dimention of convolution (2 or 3).
- **stride** – (int) The stride of convolution.
- **padding** – (int) Padding size.
- **dilation** – (int) Dilation rate.
- **conv_group** – (int) The groupt number of convolution.
- **bias** – (bool) Add bias or not for convolution.
- **norm_type** – (str or None) Normalization type, can be *batch_norm*, 'group_norm'.
- **norm_group** – (int) The number of group for group normalization.
- **acti_func** – (str or None) Activation funtion.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training:** bool

`class pymic.layer.convolution.DepthSeperableConvolutionLayer`(*in_channels*, *out_channels*, *kernel_size*, *dim=3*, *stride=1*, *padding=0*, *dilation=1*, *conv_group=1*, *bias=True*, *norm_type='batch_norm'*, *norm_group=1*, *acti_func=None*)

Bases: `Module`

Depth seperable convolution with the following components: 1x1 conv -> group conv -> (batch_norm / layer_norm / group_norm / instance_norm) -> (activation) -> (dropout) Batch norm and activation are optional.

**Parameters**

- **in_channels** – (int) The input channel number.
- **out_channels** – (int) The output channel number.
- **kernel_size** – The size of convolution kernel. It can be either a single int or a tupe of two or three ints.
- **dim** – (int) The dimention of convolution (2 or 3).
- **stride** – (int) The stride of convolution.
- **padding** – (int) Padding size.
- **dilation** – (int) Dilation rate.
- **conv_group** – (int) The groupt number of convolution.
- **bias** – (bool) Add bias or not for convolution.
- **norm_type** – (str or None) Normalization type, can be *batch_norm*, ‘group_norm’.
- **norm_group** – (int) The number of group for group normalization.
- **acti_func** – (str or None) Activation funtion.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training:** bool

### 3.2.4 pymic.layer.deconvolution module

**class** pymic.layer.deconvolution.**DeconvolutionLayer**(*in_channels*, *out_channels*, *kernel_size*, *dim=3*, *stride=1*, *padding=0*, *output_padding=0*, *dilation=1*, *groups=1*, *bias=True*, *batch_norm=True*, *acti_func=None*)

Bases: `Module`

A compose layer with the following components: deconvolution -> (batch_norm / layer_norm / group_norm / instance_norm) -> (activation) -> (dropout) Batch norm and activation are optional.

**Parameters**

- **in_channels** – (int) The input channel number.
- **out_channels** – (int) The output channel number.
- **kernel_size** – The size of convolution kernel. It can be either a single int or a tupe of two or three ints.
- **dim** – (int) The dimention of convolution (2 or 3).
- **stride** – (int) The stride of convolution.
- **padding** – (int) Padding size.
- **dilation** – (int) Dilation rate.
- **groups** – (int) The groupt number of convolution.
- **bias** – (bool) Add bias or not for convolution.
- **batch_norm** – (bool) Use batch norm or not.
- **acti_func** – (str or None) Activation funtion.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training:** bool

**class** pymic.layer.deconvolution.**DepthSeperableDeconvolutionLayer**(*in_channels*, *out_channels*, *kernel_size*, *dim=3*, *stride=1*, *padding=0*, *output_padding=0*, *dilation=1*, *groups=1*, *bias=True*, *batch_norm=True*, *acti_func=None*)

Bases: `Module`

Depth seperable deconvolution with the following components: 1x1 conv -> deconv -> (batch_norm / layer_norm / group_norm / instance_norm) -> (activation) -> (dropout) Batch norm and activation are optional.

**Parameters**

- **in_channels** – (int) The input channel number.

- **out_channels** – (int) The output channel number.
- **kernel_size** – The size of convolution kernel. It can be either a single int or a tupe of two or three ints.
- **dim** – (int) The dimention of convolution (2 or 3).
- **stride** – (int) The stride of convolution.
- **padding** – (int) Padding size for input.
- **output_padding** – (int) Padding size for ouput.
- **dilation** – (int) Dilation rate.
- **groups** – (int) The groupt number of convolution.
- **bias** – (bool) Add bias or not for convolution.
- **batch_norm** – (bool) Use batch norm or not.
- **acti_func** – (str or None) Activation funtion.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training:** `bool`

### 3.2.5 pymic.layer.space2channel module

class pymic.layer.space2channel.**ChannelToSpace3D**

Bases: `Module`

Channel to space transform for 3D input.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training:** `bool`

class pymic.layer.space2channel.**SpaceToChannel3D**

Bases: `Module`

Space to channel transform for 3D input.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training:** `bool`

### 3.2.6 Module contents

## 3.3 pymic.loss package

### 3.3.1 Subpackages

#### pymic.loss.cls package

#### Submodules

#### pymic.loss.cls.basic module

**class** `pymic.loss.cls.basic.`**AbstractClassificationLoss**(*params=None*)

Bases: `Module`

Abstract Classification Loss.

**forward**(*loss_input_dict*)

The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields.

**Parameters**

- **prediction** – A prediction with shape of [N, C] where C is the class number.
- **ground_truth** – The corresponding ground truth, with shape of [N, 1].

Note that *prediction* is the digit output of a network, before using softmax.

**training:** `bool`

**class** `pymic.loss.cls.basic.`**CrossEntropyLoss**(*params=None*)

Bases: *AbstractClassificationLoss*

Standard Softmax-based CE loss.

**forward**(*loss_input_dict*)

The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields.

**Parameters**

- **prediction** – A prediction with shape of [N, C] where C is the class number.
- **ground_truth** – The corresponding ground truth, with shape of [N, 1].

Note that *prediction* is the digit output of a network, before using softmax.

**training: bool**

class pymic.loss.cls.basic.**L1Loss**(*params=None*)

Bases: *AbstractClassificationLoss*

L1 (MAE) loss for classification

**forward**(*loss_input_dict*)

The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields.

**Parameters**

- **prediction** – A prediction with shape of [N, C] where C is the class number.
- **ground_truth** – The corresponding ground truth, with shape of [N, 1].

Note that *prediction* is the digit output of a network, before using softmax.

**training: bool**

class pymic.loss.cls.basic.**MSELoss**(*params=None*)

Bases: *AbstractClassificationLoss*

Mean Square Error loss for classification.

**forward**(*loss_input_dict*)

The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields.

**Parameters**

- **prediction** – A prediction with shape of [N, C] where C is the class number.
- **ground_truth** – The corresponding ground truth, with shape of [N, 1].

Note that *prediction* is the digit output of a network, before using softmax.

**training: bool**

class pymic.loss.cls.basic.**NLLLoss**(*params=None*)

Bases: *AbstractClassificationLoss*

The negative log likelihood loss for classification.

**forward**(*loss_input_dict*)

The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields.

**Parameters**

- **prediction** – A prediction with shape of [N, C] where C is the class number.
- **ground_truth** – The corresponding ground truth, with shape of [N, 1].

Note that *prediction* is the digit output of a network, before using softmax.

**training: bool**

class pymic.loss.cls.basic.**SigmoidCELoss**(*params=None*)

Bases: *AbstractClassificationLoss*

Sigmoid-based CE loss.

**forward**(*loss_input_dict*)

The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields.

**Parameters**

- **prediction** – A prediction with shape of [N, C] where C is the class number.
- **ground_truth** – The corresponding ground truth, with shape of [N, 1].

Note that *prediction* is the digit output of a network, before using softmax.

**training: bool**

### pymic.loss.cls.util module

pymic.loss.cls.util.**get_soft_label**(*input_tensor*, *num_class*, *data_type='float'*)

Convert a label tensor to one-hot soft label.

**Parameters**

- **input_tensor** – Tensor with shape of [B, 1].
- **output_tensor** – Tensor with shape of [B, num_class].
- **num_class** – (int) Class number.
- **data_type** – (str) *float* or *double*.

### Module contents

### pymic.loss.seg package

### Submodules

### pymic.loss.seg.abstract module

class pymic.loss.seg.abstract.**AbstractSegLoss**(*params=None*)

Bases: Module

Abstract class for loss function of segmentation tasks. The parameters should be written in the *params* dictionary, and it has the following fields:

**Parameters**

**loss_softmax** – (optional, bool) Apply softmax to the prediction of network or not. Default is True.

**forward**(*loss_input_dict*)

Forward pass for calculating the loss. The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields:

**Parameters**

- **prediction** – (tensor) Prediction of a network, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **ground_truth** – (tensor) Ground truth, with the shape of [N, C, D, H, W] or [N, C, H, W].

- **pixel_weight** – (optional) Pixel-wise weight map, with the shape of [N, 1, D, H, W] or [N, 1, H, W]. Default is None.

**Returns**
Loss function value.

**training: bool**

### pymic.loss.seg.ce module

**class** pymic.loss.seg.ce.**CrossEntropyLoss**(*params=None*)

Bases: *AbstractSegLoss*

Cross entropy loss for segmentation tasks.

The parameters should be written in the *params* dictionary, and it has the following fields:

**Parameters**
**loss_softmax** – (optional, bool) Apply softmax to the prediction of network or not. Default is True.

**forward**(*loss_input_dict*)

Forward pass for calculating the loss. The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields:

**Parameters**

- **prediction** – (tensor) Prediction of a network, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **ground_truth** – (tensor) Ground truth, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **pixel_weight** – (optional) Pixel-wise weight map, with the shape of [N, 1, D, H, W] or [N, 1, H, W]. Default is None.

**Returns**
Loss function value.

**training: bool**

**class** pymic.loss.seg.ce.**GeneralizedCELoss**(*params*)

Bases: *AbstractSegLoss*

Generalized cross entropy loss to deal with noisy labels.

- Reference: Z. Zhang et al. Generalized Cross Entropy Loss for Training Deep Neural Networks with Noisy Labels, NeurIPS 2018.

The parameters should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **loss_softmax** – (bool) Apply softmax to the prediction of network or not.
- **loss_gce_q** – (float): hyper-parameter in the range of (0, 1).
- **loss_with_pixel_weight** – (optional, bool): Use pixel weighting or not.
- **loss_class_weight** – (optional, list or none): If not none, a list of weight for each class.

**forward**(*loss_input_dict*)

Forward pass for calculating the loss. The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields:

**Parameters**

- **prediction** – (tensor) Prediction of a network, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **ground_truth** – (tensor) Ground truth, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **pixel_weight** – (optional) Pixel-wise weight map, with the shape of [N, 1, D, H, W] or [N, 1, H, W]. Default is None.

**Returns**

Loss function value.

**training: bool**

## pymic.loss.seg.combined module

**class** pymic.loss.seg.combined.**CombinedLoss**(*params*, *loss_dict*)

Bases: *AbstractSegLoss*

A combination of a list of loss functions. Parameters should be saved in the *params* dictionary.

**Parameters**

- **loss_softmax** – (optional, bool) Apply softmax to the prediction of network or not. Default is True.
- **loss_type** – (list) A list of loss function name.
- **loss_weight** – (list) A list of weights for each loss fucntion.
- **loss_dict** – (dictionary) A dictionary of avaiable loss functions.

**forward**(*loss_input_dict*)

Forward pass for calculating the loss. The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields:

**Parameters**

- **prediction** – (tensor) Prediction of a network, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **ground_truth** – (tensor) Ground truth, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **pixel_weight** – (optional) Pixel-wise weight map, with the shape of [N, 1, D, H, W] or [N, 1, H, W]. Default is None.

**Returns**

Loss function value.

**training: bool**

### pymic.loss.seg.deep_sup module

class pymic.loss.seg.deep_sup.**DeepSuperviseLoss**(*params*)

Bases: `AbstractSegLoss`

Combine deep supervision with a basic loss function. Arguments should be provided in the *params* dictionary, and it has the following fields:

**Parameters**

- **loss_softmax** – (optional, bool) Apply softmax to the prediction of network or not. Default is True.
- **base_loss** – (nn.Module) The basic function used for each scale.
- **deep_supervise_weight** – (list) A list of weight for each deep supervision scale.
- **deep_supervise_model** – (int) Mode for deep supervision when the prediction has a smaller shape than the ground truth. 0: upsample the prediction to the size of the ground truth. 1: downsample the ground truth to the size of the prediction via interpolation. 2: downsample the ground truth via adaptive average pooling.

**forward**(*loss_input_dict*)

Forward pass for calculating the loss. The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields:

**Parameters**

- **prediction** – (tensor) Prediction of a network, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **ground_truth** – (tensor) Ground truth, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **pixel_weight** – (optional) Pixel-wise weight map, with the shape of [N, 1, D, H, W] or [N, 1, H, W]. Default is None.

**Returns**

Loss function value.

**training: bool**

pymic.loss.seg.deep_sup.**match_prediction_and_gt_shape**(*pred*, *gt*, *mode=0*)

### pymic.loss.seg.dice module

class pymic.loss.seg.dice.**DiceLoss**(*params=None*)

Bases: `AbstractSegLoss`

Dice loss for segmentation tasks. The parameters should be written in the *params* dictionary, and it has the following fields:

**Parameters**

**loss_softmax** – (bool) Apply softmax to the prediction of network or not.

**forward**(*loss_input_dict*)

Forward pass for calculating the loss. The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields:

**Parameters**

- **prediction** – (tensor) Prediction of a network, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **ground_truth** – (tensor) Ground truth, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **pixel_weight** – (optional) Pixel-wise weight map, with the shape of [N, 1, D, H, W] or [N, 1, H, W]. Default is None.

**Returns**
Loss function value.

**training: bool**

**class** `pymic.loss.seg.dice.FocalDiceLoss`(*params=None*)

Bases: *AbstractSegLoss*

Focal Dice according to the following paper:

- Pei Wang and Albert C. S. Chung, Focal Dice Loss and Image Dilation for Brain Tumor Segmentation, 2018.

The parameters should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **loss_softmax** – (bool) Apply softmax to the prediction of network or not.
- **FocalDiceLoss_beta** – (float) The hyper-parameter to set (>=1.0).

**forward**(*loss_input_dict*)

Forward pass for calculating the loss. The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields:

**Parameters**

- **prediction** – (tensor) Prediction of a network, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **ground_truth** – (tensor) Ground truth, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **pixel_weight** – (optional) Pixel-wise weight map, with the shape of [N, 1, D, H, W] or [N, 1, H, W]. Default is None.

**Returns**
Loss function value.

**training: bool**

**class** `pymic.loss.seg.dice.NoiseRobustDiceLoss`(*params*)

Bases: *AbstractSegLoss*

Noise-robust Dice loss according to the following paper.

- G. Wang et al. A Noise-Robust Framework for Automatic Segmentation of COVID-19 Pneumonia Lesions From CT Images, IEEE TMI, 2020.

The parameters should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **loss_softmax** – (bool) Apply softmax to the prediction of network or not.
- **NoiseRobustDiceLoss_gamma** – (float) The hyper-parameter gammar to set (1, 2).

`forward`(*loss_input_dict*)

Forward pass for calculating the loss. The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields:

**Parameters**

- **prediction** – (tensor) Prediction of a network, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **ground_truth** – (tensor) Ground truth, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **pixel_weight** – (optional) Pixel-wise weight map, with the shape of [N, 1, D, H, W] or [N, 1, H, W]. Default is None.

**Returns**

Loss function value.

`training: bool`

### pymic.loss.seg.exp_log module

`class pymic.loss.seg.exp_log.ExpLogLoss`(*params*)

Bases: *AbstractSegLoss*

The exponential logarithmic loss in this paper:

- K. Wong et al.: 3D Segmentation with Exponential Logarithmic Loss for Highly Unbalanced Object Sizes. MICCAI 2018.

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **loss_softmax** – (bool) Apply softmax to the prediction of network or not.
- **ExpLogLoss_w_dice** – (float) Weight of ExpLog Dice loss in the range of [0, 1].
- **ExpLogLoss_gamma** – (float) Hyper-parameter gamma.

`forward`(*loss_input_dict*)

Forward pass for calculating the loss. The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields:

**Parameters**

- **prediction** – (tensor) Prediction of a network, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **ground_truth** – (tensor) Ground truth, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **pixel_weight** – (optional) Pixel-wise weight map, with the shape of [N, 1, D, H, W] or [N, 1, H, W]. Default is None.

**Returns**

Loss function value.

`training: bool`

### pymic.loss.seg.gatedcrf module

The code is adapted from the original implementation at Github.

`class pymic.loss.seg.gatedcrf.GatedCRFLoss`

Bases: `Module`

Gated CRF Loss for Weakly Supervised Semantic Image Segmentation. This loss function promotes consistent label assignment guided by input features, such as RGBXY.

- Reference: Anton Obukhov, Stamatios Georgoulis, Dengxin Dai and Luc Van Gool: Gated CRF Loss for Weakly Supervised Semantic Image Segmentation. CoRR 2019.

**forward**(*y_hat_softmax*, *kernels_desc*, *kernels_radius*, *sample*, *height_input*, *width_input*, *mask_src=None*, *mask_dst=None*, *compatibility=None*, *custom_modality_downsamplers=None*, *out_kernels_vis=False*)

Performs the forward pass of the loss.

**Parameters**

- **y_hat_softmax** – A tensor of predicted per-pixel class probabilities of size NxCxHxW
- **kernels_desc** – A list of dictionaries, each describing one Gaussian kernel composition from modalities. The final kernel is a weighted sum of individual kernels. Following example is a composition of RGBXY and XY kernels: kernels_desc: [{'weight': 0.9,'xy': 6,'rgb': 0.1},{'weight': 0.1,'xy': 6}]
- **kernels_radius** – Defines size of bounding box region around each pixel in which the kernel is constructed.
- **sample** – A dictionary with modalities (except 'xy') used in kernels_desc parameter. Each of the provided modalities is allowed to be larger than the shape of y_hat_softmax, in such case downsampling will be invoked. Default downsampling method is area resize; this can be overriden by setting. custom_modality_downsamplers parameter.
- **height_input** (*width_input,*) – Dimensions of the full scale resolution of modalities
- **mask_src** – (optional) Source mask.
- **mask_dst** – (optional) Destination mask.
- **compatibility** – (optional) Classes compatibility matrix, defaults to Potts model.
- **custom_modality_downsamplers** – A dictionary of modality downsampling functions.
- **out_kernels_vis** – Whether to return a tensor with kernels visualized with some step.

**Returns**

Loss function value.

**training**: bool

### pymic.loss.seg.mse module

class pymic.loss.seg.mse.**MAELoss**(*params=None*)

Bases: `AbstractSegLoss`

Mean Absolute Loss for segmentation tasks. The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**
**loss_softmax** – (bool) Apply softmax to the prediction of network or not.

**forward**(*loss_input_dict*)

Forward pass for calculating the loss. The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields:

**Parameters**

- **prediction** – (tensor) Prediction of a network, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **ground_truth** – (tensor) Ground truth, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **pixel_weight** – (optional) Pixel-wise weight map, with the shape of [N, 1, D, H, W] or [N, 1, H, W]. Default is None.

**Returns**
Loss function value.

**training: bool**

class pymic.loss.seg.mse.**MSELoss**(*params=None*)

Bases: `AbstractSegLoss`

Mean Sequare Loss for segmentation tasks. The parameters should be written in the *params* dictionary, and it has the following fields:

**Parameters**
**loss_softmax** – (bool) Apply softmax to the prediction of network or not.

**forward**(*loss_input_dict*)

Forward pass for calculating the loss. The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields:

**Parameters**

- **prediction** – (tensor) Prediction of a network, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **ground_truth** – (tensor) Ground truth, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **pixel_weight** – (optional) Pixel-wise weight map, with the shape of [N, 1, D, H, W] or [N, 1, H, W]. Default is None.

**Returns**
Loss function value.

**training: bool**

### pymic.loss.seg.mumford_shah module

class pymic.loss.seg.mumford_shah.**MumfordShahLoss**(*params=None*)

Bases: Module

Implementation of Mumford Shah Loss for weakly supervised learning.

- Reference: Boah Kim and Jong Chul Ye: Mumford–Shah Loss Functional for Image Segmentation With Deep Learning. IEEE TIP, 2019.

The oringial implementation is availabel at Github. Currently only 2D version is supported.

The parameters should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **loss_softmax** – (bool) Apply softmax to the prediction of network or not.
- **MumfordShahLoss_penalty** – (optional, str) *l1* or *l2*. Default is *l1*.
- **MumfordShahLoss_lambda** – (optional, float) Hyper-parameter lambda, default is 1.0.

**forward**(*loss_input_dict*)

Forward pass for calculating the loss. The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields:

**Parameters**

- **prediction** – (tensor) Prediction of a network, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **image** – (tensor) Image, with the shape of [N, C, D, H, W] or [N, C, H, W].

**Returns**

Loss function value.

**get_gradient_loss**(*pred*, *penalty='l2'*)

**get_levelset_loss**(*output*, *target*)

Get the level set loss value.

**Parameters**

- **output** – (tensor) softmax output of a network.
- **target** – (tensor) the input image.

**Returns**

the level set loss.

**training: bool**

### pymic.loss.seg.slsr module

class pymic.loss.seg.slsr.**SLSRLoss**(*params=None*)

Bases: `AbstractSegLoss`

Spatial Label Smoothing Regularization (SLSR) loss for learning from noisy annotatins. This loss requires pixel weighting, please make sure that a *pixel_weight* field is provided for the csv file of the training images.

The pixel wight here is actually the confidence mask, i.e., if the value is one, it means the label of corresponding pixel is noisy and should be smoothed.

- Reference: Minqing Zhang, Jiantao Gao et al.: Characterizing Label Errors: Confident Learning for Noisy-Labeled Image Segmentation, MICCAI 2020.

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **loss_softmax** – (bool) Apply softmax to the prediction of network or not.
- **slsrloss_epsilon** – (optional, float) Hyper-parameter epsilon. Default is 0.25.

**forward**(*loss_input_dict*)

Forward pass for calculating the loss. The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields:

**Parameters**

- **prediction** – (tensor) Prediction of a network, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **ground_truth** – (tensor) Ground truth, with the shape of [N, C, D, H, W] or [N, C, H, W].
- **pixel_weight** – (optional) Pixel-wise weight map, with the shape of [N, 1, D, H, W] or [N, 1, H, W]. Default is None.

**Returns**

Loss function value.

**training: bool**

### pymic.loss.seg.ssl module

class pymic.loss.seg.ssl.**EntropyLoss**(*params=None*)

Bases: `AbstractSegLoss`

Entropy Minimization for segmentation tasks. The parameters should be written in the *params* dictionary, and it has the following fields:

**Parameters**

**loss_softmax** – (bool) Apply softmax to the prediction of network or not.

**forward**(*loss_input_dict*)

Forward pass for calculating the loss. The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields:

**Parameters**

**prediction** – (tensor) Prediction of a network, with the shape of [N, C, D, H, W] or [N, C, H, W].

**Returns**
Loss function value.

**training: bool**

**class pymic.loss.seg.ssl.TotalVariationLoss**(*params=None*)

Bases: *AbstractSegLoss*

Total Variation Loss for segmentation tasks. The parameters should be written in the *params* dictionary, and it has the following fields:

**Parameters**
**loss_softmax** – (bool) Apply softmax to the prediction of network or not.

**forward**(*loss_input_dict*)

Forward pass for calculating the loss. The arguments should be written in the *loss_input_dict* dictionary, and it has the following fields:

**Parameters**
**prediction** – (tensor) Prediction of a network, with the shape of [N, C, D, H, W] or [N, C, H, W].

**Returns**
Loss function value.

**training: bool**

### pymic.loss.seg.util module

**pymic.loss.seg.util.get_classwise_dice**(*predict*, *soft_y*, *pix_w=None*)

Get dice scores for each class in predict (after softmax) and soft_y.

**Parameters**

- **predict** – (tensor) Prediction of a segmentation network after softmax.
- **soft_y** – (tensor) The one-hot segmentation ground truth.
- **pix_w** – (optional, tensor) The pixel weight map. Default is None.

**Returns**
Dice score for each class.

**pymic.loss.seg.util.get_soft_label**(*input_tensor*, *num_class*, *data_type='float'*)

Convert a label tensor to one-hot label for segmentation tasks.

**Parameters**

- **input_tensor** – (tensor) Tensor with shae [B, 1, D, H, W] or [B, 1, H, W].
- **num_class** – (int) The class number.
- **data_type** – (optional, str) Type of data, *float* (default) or *double*.

**Returns**
A tensor with shape [B, num_class, D, H, W] or [B, num_class, H, W]

**pymic.loss.seg.util.reshape_prediction_and_ground_truth**(*predict*, *soft_y*)

Reshape input variables two 2D.

**Parameters**

- **predict** – (tensor) A tensor of shape [N, C, D, H, W] or [N, C, H, W].
- **soft_y** – (tensor) A tensor of shape [N, C, D, H, W] or [N, C, H, W].

**Returns**
Two output tensors with shape [voxel_n, C] that correspond to the two inputs.

`pymic.loss.seg.util.reshape_tensor_to_2D(x)`
Reshape input tensor of shape [N, C, D, H, W] or [N, C, H, W] to [voxel_n, C]

#### Module contents

### 3.3.2 Submodules

### 3.3.3 pymic.loss.loss_dict_cls module

Built-in loss functions for classification.

- CrossEntropyLoss *pymic.loss.cls.basic.CrossEntropyLoss*
- SigmoidCELoss *pymic.loss.cls.basic.SigmoidCELoss*
- L1Loss *pymic.loss.cls.basic.L1Loss*
- MSELoss *pymic.loss.cls.basic.MSELoss*
- NLLLoss *pymic.loss.cls.basic.NLLLoss*

### 3.3.4 pymic.loss.loss_dict_seg module

Built-in loss functions for segmentation. The following are for fully supervised learning, or learnig from noisy labels:

- CrossEntropyLoss *pymic.loss.seg.ce.CrossEntropyLoss*
- GeneralizedCELoss *pymic.loss.seg.ce.GeneralizedCELoss*
- DiceLoss *pymic.loss.seg.dice.DiceLoss*
- FocalDiceLoss *pymic.loss.seg.dice.FocalDiceLoss*
- NoiseRobustDiceLoss *pymic.loss.seg.dice.NoiseRobustDiceLoss*
- ExpLogLoss *pymic.loss.seg.exp_log.ExpLogLoss*
- MAELoss *pymic.loss.seg.mse.MAELoss*
- MSELoss *pymic.loss.seg.mse.MSELoss*
- SLSRLoss *pymic.loss.seg.slsr.SLSRLoss*

The following are for semi-supervised or weakly supervised learning:

- EntropyLoss *pymic.loss.seg.ssl.EntropyLoss*
- GatedCRFLoss: *pymic.loss.seg.gatedcrf.GatedCRFLoss*
- MumfordShahLoss *pymic.loss.seg.mumford_shah.MumfordShahLoss*
- TotalVariationLoss *pymic.loss.seg.ssl.TotalVariationLoss*

### 3.3.5 Module contents

## 3.4 pymic.net package

### 3.4.1 Subpackages

#### pymic.net.cls package

#### Submodules

#### pymic.net.cls.torch_pretrained_net module

class pymic.net.cls.torch_pretrained_net.**BuiltInNet**(*params*)

Bases: `Module`

Built-in Network in Pytorch for classification. Parameters should be set in the *params* dictionary that contains the following fields:

**Parameters**

- **input_chns** – (int) Input channel number, default is 3.
- **pretrain** – (bool) Using pretrained model or not, default is True.
- **update_mode** – (str) The strategy for updating layers: "*all*" means updating all the layers, and "*last*" (by default) means updating the last layer, as well as the first layer when *input_chns* is not 3.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**get_parameters_to_update**()

**training: bool**

class pymic.net.cls.torch_pretrained_net.**MobileNetV2**(*params*)

Bases: `BuiltInNet`

MobileNetV2 for classification. Parameters should be set in the *params* dictionary that contains the following fields:

**Parameters**

- **input_chns** – (int) Input channel number, default is 3.
- **pretrain** – (bool) Using pretrained model or not, default is True.
- **update_mode** – (str) The strategy for updating layers: "*all*" means updating all the layers, and "*last*" (by default) means updating the last layer, as well as the first layer when *input_chns* is not 3.

**get_parameters_to_update**()

**training**: bool

class pymic.net.cls.torch_pretrained_net.**ResNet18**(*params*)

Bases: *BuiltInNet*

ResNet18 for classification. Parameters should be set in the *params* dictionary that contains the following fields:

**Parameters**

- **input_chns** – (int) Input channel number, default is 3.
- **pretrain** – (bool) Using pretrained model or not, default is True.
- **update_mode** – (str) The strategy for updating layers: "*all*" means updating all the layers, and "*last*" (by default) means updating the last layer, as well as the first layer when *input_chns* is not 3.

**get_parameters_to_update**()

**training**: bool

class pymic.net.cls.torch_pretrained_net.**VGG16**(*params*)

Bases: *BuiltInNet*

VGG16 for classification. Parameters should be set in the *params* dictionary that contains the following fields:

**Parameters**

- **input_chns** – (int) Input channel number, default is 3.
- **pretrain** – (bool) Using pretrained model or not, default is True.
- **update_mode** – (str) The strategy for updating layers: "*all*" means updating all the layers, and "*last*" (by default) means updating the last layer, as well as the first layer when *input_chns* is not 3.

**get_parameters_to_update**()

**training**: bool

### Module contents

### pymic.net.net2d package

### Submodules

### pymic.net.net2d.cople_net module

class pymic.net.net2d.cople_net.**ASPPBlock**(*in_channels*, *out_channels_list*, *kernel_size_list*, *dilation_list*)

Bases: Module

ASPP block.

`forward(x)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**training: bool**

`class pymic.net.net2d.cople_net.COPLENet(params)`

Bases: `Module`

Implementation of of COPLENet for COVID-19 pneumonia lesion segmentation from CT images.

- Reference: G. Wang et al. A Noise-robust Framework for Automatic Segmentation of COVID-19 Pneumonia Lesions from CT Images. IEEE Transactions on Medical Imaging, 39(8),2020:2653-2663.

Parameters are given in the *params* dictionary, and should include the following fields:

**Parameters**

- **in_chns** – (int) Input channel number.
- **feature_chns** – (list) Feature channel for each resolution level. The length should be 5, such as [16, 32, 64, 128, 256].
- **dropout** – (list) The dropout ratio for each resolution level. The length should be the same as that of *feature_chns*.
- **class_num** – (int) The class number for segmentation task.
- **bilinear** – (bool) Using bilinear for up-sampling or not. If False, deconvolution will be used for up-sampling.

`forward(x)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**training: bool**

`class pymic.net.net2d.cople_net.ConvBNActBlock(in_channels, out_channels, dropout_p)`

Bases: `Module`

Two convolution layers with batch norm, leaky relu, dropout and SE block.

`forward(x)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training: bool**

class pymic.net.net2d.cople_net.**ConvLayer**(*in_channels*, *out_channels*, *kernel_size=1*)

Bases: `Module`

A combination of Conv2d, BatchNorm2d and LeakyReLU.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training: bool**

class pymic.net.net2d.cople_net.**DownBlock**(*in_channels*, *out_channels*, *dropout_p*)

Bases: `Module`

Downsampling by a concantenation of max-pool and avg-pool, followed by ConvBNActBlock.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training: bool**

class pymic.net.net2d.cople_net.**SEBlock**(*in_channels*, *r*)

Bases: `Module`

A Modified Squeeze-and-Excitation block for spatial attention.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training: bool**

**class** pymic.net.net2d.cople_net.**UpBlock**(*in_channels1*, *in_channels2*, *out_channels*, *bilinear=True*, *dropout_p=0.5*)

Bases: `Module`

Upssampling followed by ConvBNActBlock.

**forward**(*x1*, *x2*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**training: bool**

### pymic.net.net2d.scse2d module

2D implementation of:

1. Channel Squeeze and Excitation
2. Spatial Squeeze and Excitation
3. Concurrent Spatial and Channel Squeeze & Excitation

Oringinal file is on Github.

**class** pymic.net.net2d.scse2d.**ChannelSELayer**(*num_channels*, *reduction_ratio=2*)

Bases: `Module`

Re-implementation of Squeeze-and-Excitation (SE) block.

- Reference: Jie Hu, Li Shen, Gang Sun: Squeeze-and-Excitation Networks. CVPR 2018.

**Parameters**

- **num_channels** – Number of input channels
- **reduction_ratio** – By how much should the num_channels should be reduced.

**forward**(*input_tensor*)

**Parameters**
**input_tensor** – X, shape = (batch_size, num_channels, H, W)

**Returns**
output tensor

**training: bool**

**class** pymic.net.net2d.scse2d.**ChannelSpatialSELayer**(*num_channels*, *reduction_ratio=2*)

Bases: `Module`

Re-implementation of concurrent spatial and channel squeeze & excitation.

- Reference: Roy et al., Concurrent Spatial and Channel Squeeze & Excitation in Fully Convolutional Networks, MICCAI 2018.

  **Parameters**

  - **num_channels** – Number of input channels.
  - **reduction_ratio** – By how much should the num_channels should be reduced.

**forward**(*input_tensor*)

**Parameters**
**input_tensor** – X, shape = (batch_size, num_channels, H, W)

**Returns**
output_tensor

**training: bool**

class pymic.net.net2d.scse2d.**SELayer**(*value*)

Bases: Enum

Enum restricting the type of SE Blockes available. So that type checking can be adding when adding these blockes to a neural network:

```
if self.se_block_type == se.SELayer.CSE.value:
    self.SELayer = se.ChannelSpatialSELayer(params['num_filters'])
elif self.se_block_type == se.SELayer.SSE.value:
    self.SELayer = se.SpatialSELayer(params['num_filters'])
elif self.se_block_type == se.SELayer.CSSE.value:
    self.SELayer = se.ChannelSpatialSELayer(params['num_filters'])
```

**CSE = 'CSE'**

**CSSE = 'CSSE'**

**NONE = 'NONE'**

**SSE = 'SSE'**

class pymic.net.net2d.scse2d.**SpatialSELayer**(*num_channels*)

Bases: Module

Re-implementation of SE block – squeezing spatially and exciting channel-wise.

- Reference: Roy et al., Concurrent Spatial and Channel Squeeze & Excitation in Fully Convolutional Networks, MICCAI 2018.

  **Parameters**
  **num_channels** – Number of input channels.

**forward**(*input_tensor*, *weights=None*)

**Parameters**

- **weights** – weights for few shot learning
- **input_tensor** – X, shape = (batch_size, num_channels, H, W)

**Returns**
output_tensor

**training: bool**

### pymic.net.net2d.unet2d module

**class** pymic.net.net2d.unet2d.**ConvBlock**(*in_channels*, *out_channels*, *dropout_p*)

Bases: `Module`

Two convolution layers with batch norm and leaky relu. Droput is used between the two convolution layers.

**Parameters**

- **in_channels** – (int) Input channel number.
- **out_channels** – (int) Output channel number.
- **dropout_p** – (int) Dropout probability.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training: bool**

**class** pymic.net.net2d.unet2d.**Decoder**(*params*)

Bases: `Module`

Decoder of 2D UNet.

Parameters are given in the *params* dictionary, and should include the following fields:

**Parameters**

- **in_chns** – (int) Input channel number.
- **feature_chns** – (list) Feature channel for each resolution level. The length should be 4 or 5, such as [16, 32, 64, 128, 256].
- **dropout** – (list) The dropout ratio for each resolution level. The length should be the same as that of *feature_chns*.
- **class_num** – (int) The class number for segmentation task.
- **bilinear** – (bool) Using bilinear for up-sampling or not. If False, deconvolution will be used for up-sampling.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training: bool**

class pymic.net.net2d.unet2d.**DownBlock**(*in_channels*, *out_channels*, *dropout_p*)

Bases: `Module`

Downsampling followed by ConvBlock

**Parameters**

- **in_channels** – (int) Input channel number.
- **out_channels** – (int) Output channel number.
- **dropout_p** – (int) Dropout probability.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training: bool**

class pymic.net.net2d.unet2d.**Encoder**(*params*)

Bases: `Module`

Encoder of 2D UNet.

Parameters are given in the *params* dictionary, and should include the following fields:

**Parameters**

- **in_chns** – (int) Input channel number.
- **feature_chns** – (list) Feature channel for each resolution level. The length should be 4 or 5, such as [16, 32, 64, 128, 256].
- **dropout** – (list) The dropout ratio for each resolution level. The length should be the same as that of *feature_chns*.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training: bool**

class pymic.net.net2d.unet2d.**UNet2D**(*params*)

Bases: `Module`

An implementation of 2D U-Net.

- Reference: Olaf Ronneberger, Philipp Fischer, Thomas Brox: U-Net: Convolutional Networks for Biomedical Image Segmentation. MICCAI (3) 2015: 234-241

Note that there are some modifications from the original paper, such as the use of batch normalization, dropout, leaky relu and deep supervision.

Parameters are given in the *params* dictionary, and should include the following fields:

**Parameters**

- **in_chns** – (int) Input channel number.
- **feature_chns** – (list) Feature channel for each resolution level. The length should be 4 or 5, such as [16, 32, 64, 128, 256].
- **dropout** – (list) The dropout ratio for each resolution level. The length should be the same as that of *feature_chns*.
- **class_num** – (int) The class number for segmentation task.
- **bilinear** – (bool) Using bilinear for up-sampling or not. If False, deconvolution will be used for up-sampling.
- **multiscale_pred** – (bool) Get multiscale prediction.

`forward`(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training:** `bool`

class pymic.net.net2d.unet2d.**UpBlock**(*in_channels1*, *in_channels2*, *out_channels*, *dropout_p*, *bilinear=True*)

Bases: `Module`

Upsampling followed by ConvBlock

**Parameters**

- **in_channels1** – (int) Channel number of high-level features.
- **in_channels2** – (int) Channel number of low-level features.
- **out_channels** – (int) Output channel number.
- **dropout_p** – (int) Dropout probability.
- **bilinear** – (bool) Use bilinear for up-sampling (by default). If False, deconvolution is used for up-sampling.

`forward`(*x1*, *x2*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training: bool**

## pymic.net.net2d.unet2d_attention module

**class** pymic.net.net2d.unet2d_attention.**AttentionGateBlock**(*chns_l*, *chns_h*)

Bases: `Module`

**forward**(*x_l*, *x_h*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training: bool**

**class** pymic.net.net2d.unet2d_attention.**AttentionUNet2D**(*params*)

Bases: *UNet2D*

**training: bool**

**class** pymic.net.net2d.unet2d_attention.**UpBlockWithAttention**(*in_channels1*, *in_channels2*, *out_channels*, *dropout_p*, *bilinear=True*)

Bases: `Module`

Upsampling followed by ConvBlock

**forward**(*x1*, *x2*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training: bool**

### pymic.net.net2d.unet2d_cct module

class pymic.net.net2d.unet2d_cct.**AuxiliaryDecoder**(*params*, *aux_type*)

Bases: `Module`

An Auxiliary Decoder. *aux_type* should be one of {*DropOut*, *FeatureDrop*, *FeatureNoise* and *VAT*}. Other parameters for the decoder are given in the *params* dictionary, see *pymic.net.net2d.unet2d.Decoder* for details. In addition, the following fields are needed for pertubation:

**Parameters**

- **Uniform_range** – (float) The range of noise. Only needed when *aux_type* `= `*FeatureNoise*.
- **VAT_it** – (float) The iteration number of VAT. Only needed when *aux_type* `= `*VAT*.
- **VAT_xi** – (float) The hyper-parameter xi of VAT. Only needed when *aux_type* `= `*VAT*.
- **VAT_eps** – (float) The hyper-parameter eps of VAT. Only needed when *aux_type* `= `*VAT*.

**feature_based_noise**(*x*)

**feature_drop**(*x*)

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training: bool**

class pymic.net.net2d.unet2d_cct.**UNet2D_CCT**(*params*)

Bases: `Module`

An modification the U-Net with auxiliary decoders according to the CCT paper.

- Reference: Yassine Ouali, Celine Hudelot and Myriam Tami: Semi-Supervised Semantic Segmentation With Cross-Consistency Training. CVPR 2020.

Code adapted from Github.

Parameter for the network backbone are given in the *params* dictionary, see *pymic.net.net2d.unet2d.UNet2D* for details. In addition, the following fields are needed for pertubation in the auxiliary decoders:

**Parameters**

**CCT_aux_decoders** – (list) A list of auxiliary decoder types. Supported values are {*DropOut*, *FeatureDrop*, *FeatureNoise* and *VAT*}.

The parameters for different types of auxiliary decoders should also be given in the *params* dictionary, see *pymic.net.net2d.unet2d_cct.AuxiliaryDecoder* for details.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training:** bool

### pymic.net.net2d.unet2d_dual_branch module

class pymic.net.net2d.unet2d_dual_branch.**UNet2D_DualBranch**(*params*)

Bases: `Module`

A dual branch network using UNet2D as backbone.

- Reference: Xiangde Luo, Minhao Hu, Wenjun Liao, Shuwei Zhai, Tao Song, Guotai Wang, Shaoting Zhang. ScribblScribble-Supervised Medical Image Segmentation via Dual-Branch Network and Dynamically Mixed Pseudo Labels Supervision. MICCAI 2022.

The parameters for the backbone should be given in the *params* dictionary. See *pymic.net.net2d.unet2d.UNet2D* for details. In addition, the following field should be included:

**Parameters**

**output_mode** – (str) How to obtain the result during the inference. *average*: taking average of the two branches. *first*: takeing the result in the first branch. *second*: taking the result in the second branch.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training:** bool

### pymic.net.net2d.unet2d_nest module

class pymic.net.net2d.unet2d_nest.**NestedUNet2D**(*params*)

Bases: `Module`

An implementation of the Nested U-Net.

- Reference: Zongwei Zhou, et al.: UNet++: A Nested U-Net Architecture for Medical Image Segmentation. MICCAI DLMIA workshop, 2018: 3-11.

Note that there are some modifications from the original paper, such as the use of dropout and leaky relu here.

Parameters are given in the *params* dictionary, and should include the following fields:

**Parameters**

- **in_chns** – (int) Input channel number.

- **feature_chns** – (list) Feature channel for each resolution level. The length should be 4 or 5, such as [16, 32, 64, 128, 256].
- **dropout** – (list) The dropout ratio for each resolution level. The length should be the same as that of *feature_chns*.
- **class_num** – (int) The class number for segmentation task.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training:** bool

## pymic.net.net2d.unet2d_scse module

class pymic.net.net2d.unet2d_scse.**ConvScSEBlock**(*in_channels*, *out_channels*, *dropout_p*)

Bases: `Module`

Two convolutional blocks followed by *ChannelSpatialSELayer*. Each block consists of *Conv2d* + *BatchNorm2d* + *LeakyReLU*. A dropout layer is used between the wo blocks.

**Parameters**

- **in_channels** – (int) Input channel number.
- **out_channels** – (int) Output channel number.
- **dropout_p** – (int) Dropout probability.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training:** bool

class pymic.net.net2d.unet2d_scse.**DownBlock**(*in_channels*, *out_channels*, *dropout_p*)

Bases: `Module`

Downsampling followed by *ConvScSEBlock*.

**Parameters**

- **in_channels** – (int) Input channel number.
- **out_channels** – (int) Output channel number.
- **dropout_p** – (int) Dropout probability.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**training: bool**

class pymic.net.net2d.unet2d_scse.**UNet2D_ScSE**(*params*)

Bases: `Module`

Combining 2D U-Net with SCSE module.

- Reference: Abhijit Guha Roy, Nassir Navab, Christian Wachinger: Recalibrating Fully Convolutional Networks With Spatial and Channel "Squeeze and Excitation" Blocks. IEEE Trans. Med. Imaging 38(2): 540-549 (2019).

Parameters are given in the *params* dictionary, and should include the following fields:

**Parameters**

- **in_chns** – (int) Input channel number.
- **feature_chns** – (list) Feature channel for each resolution level. The length should be 5, such as [16, 32, 64, 128, 256].
- **dropout** – (list) The dropout ratio for each resolution level. The length should be the same as that of *feature_chns*.
- **class_num** – (int) The class number for segmentation task.
- **bilinear** – (bool) Using bilinear for up-sampling or not. If False, deconvolution will be used for up-sampling.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**training: bool**

class pymic.net.net2d.unet2d_scse.**UpBlock**(*in_channels1*, *in_channels2*, *out_channels*, *dropout_p*, *bilinear=True*)

Bases: `Module`

Up-sampling followed by *ConvScSEBlock* in U-Net structure.

**Parameters**

- **in_channels1** – (int) Input channel number for low-resolution feature map.
- **in_channels2** – (int) Input channel number for high-resolution feature map.

- **out_channels** – (int) Output channel number.
- **dropout_p** – (int) Dropout probability.
- **bilinear** – (bool) Use bilinear for up-sampling or not.

**forward**(*x1*, *x2*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training:** `bool`

### pymic.net.net2d.unet2d_urpc module

### Module contents

### pymic.net.net3d package

### Submodules

### pymic.net.net3d.scse3d module

3D implementation of:

1. Channel Squeeze and Excitation
2. Spatial Squeeze and Excitation
3. Concurrent Spatial and Channel Squeeze & Excitation

Oringinal file is on Github.

**class** pymic.net.net3d.scse3d.**ChannelSELayer3D**(*num_channels*, *reduction_ratio=2*)

Bases: `Module`

3D implementation of Squeeze-and-Excitation (SE) block.

- Reference: Jie Hu, Li Shen, Gang Sun: Squeeze-and-Excitation Networks. CVPR 2018.

**Parameters**

- **num_channels** – Number of input channels
- **reduction_ratio** – By how much should the num_channels should be reduced

**forward**(*input_tensor*)

**Parameters**
**input_tensor** – X, shape = (batch_size, num_channels, D, H, W)

**Returns**
output tensor

**training: bool**

**class** pymic.net.net3d.scse3d.**ChannelSpatialSELayer3D**(*num_channels*, *reduction_ratio=2*)

Bases: Module

3D Re-implementation of concurrent spatial and channel squeeze & excitation.

- Reference: Roy et al., Concurrent Spatial and Channel Squeeze & Excitation in Fully Convolutional Networks, MICCAI 2018.

**Parameters**

- **num_channels** – Number of input channels
- **reduction_ratio** – By how much should the num_channels should be reduced

**forward**(*input_tensor*)

**Parameters**
**input_tensor** – X, shape = (batch_size, num_channels, D, H, W)

**Returns**
output_tensor

**training: bool**

**class** pymic.net.net3d.scse3d.**SELayer**(*value*)

Bases: Enum

Enum restricting the type of SE Blockes available. So that type checking can be adding when adding these blockes to a neural network:

```
if self.se_block_type == se.SELayer.CSE.value:
    self.SELayer = se.ChannelSpatialSELayer(params['num_filters'])
elif self.se_block_type == se.SELayer.SSE.value:
    self.SELayer = se.SpatialSELayer(params['num_filters'])
elif self.se_block_type == se.SELayer.CSSE.value:
    self.SELayer = se.ChannelSpatialSELayer(params['num_filters'])
```

**CSE = 'CSE'**

**CSSE = 'CSSE'**

**NONE = 'NONE'**

**SSE = 'SSE'**

**class** pymic.net.net3d.scse3d.**SpatialSELayer3D**(*num_channels*)

Bases: Module

3D Re-implementation of SE block – squeezing spatially and exciting channel-wise described in:

- Reference: Roy et al., Concurrent Spatial and Channel Squeeze & Excitation in Fully Convolutional Networks, MICCAI 2018.

**Parameters**
**num_channels** – Number of input channels

**forward**(*input_tensor*, *weights=None*)

**Parameters**

- **weights** – weights for few shot learning
- **input_tensor** – X, shape = (batch_size, num_channels, D, H, W)

**Returns**

output_tensor

**training: bool**

## pymic.net.net3d.unet2d5 module

**class** pymic.net.net3d.unet2d5.**ConvBlockND**(*in_channels*, *out_channels*, *dim=2*, *dropout_p=0.0*)

Bases: `Module`

2D or 3D convolutional block

**Parameters**

- **in_channels** – (int) Input channel number.
- **out_channels** – (int) Output channel number.
- **dim** – (int) Should be 2 or 3, for 2D and 3D convolution, respectively.
- **dropout_p** – (int) Dropout probability.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training: bool**

**class** pymic.net.net3d.unet2d5.**DownBlock**(*in_channels*, *out_channels*, *dim=2*, *dropout_p=0.0*, *downsample=True*)

Bases: `Module`

*ConvBlockND* block followed by downsampling.

**Parameters**

- **in_channels** – (int) Input channel number.
- **out_channels** – (int) Output channel number.
- **dim** – (int) Should be 2 or 3, for 2D and 3D convolution, respectively.
- **dropout_p** – (int) Dropout probability.
- **downsample** – (bool) Use downsample or not after convolution.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**training: bool**

class pymic.net.net3d.unet2d5.**UNet2D5**(*params*)

Bases: `Module`

A 2.5D network combining 3D convolutions with 2D convolutions.

- Reference: Guotai Wang, Jonathan Shapey, Wenqi Li, Reuben Dorent, Alex Demitriadis, Sotirios Bisdas, Ian Paddick, Robert Bradford, Shaoting Zhang, Sébastien Ourselin, Tom Vercauteren: Automatic Segmentation of Vestibular Schwannoma from T2-Weighted MRI by Deep Spatial Attention with Hardness-Weighted Loss. MICCAI (2) 2019: 264-272.

Note that the attention module in the orininal paper is not used here.

Parameters are given in the *params* dictionary, and should include the following fields:

**Parameters**

- **in_chns** – (int) Input channel number.
- **feature_chns** – (list) Feature channel for each resolution level. The length should be 5, such as [16, 32, 64, 128, 256].
- **dropout** – (list) The dropout ratio for each resolution level. The length should be the same as that of *feature_chns*.
- **conv_dims** – (list) The convolution dimension (2 or 3) for each resolution level. The length should be the same as that of *feature_chns*.
- **class_num** – (int) The class number for segmentation task.
- **bilinear** – (bool) Using bilinear for up-sampling or not. If False, deconvolution will be used for up-sampling.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**training: bool**

class pymic.net.net3d.unet2d5.**UpBlock**(*in_channels1*, *in_channels2*, *out_channels*, *dim=2*, *dropout_p=0.0*, *bilinear=True*)

Bases: `Module`

Upsampling followed by *ConvBlockND* block

**Parameters**

- **in_channels1** – (int) Input channel number for low-resolution feature map.
- **in_channels2** – (int) Input channel number for high-resolution feature map.
- **out_channels** – (int) Output channel number.
- **dim** – (int) Should be 2 or 3, for 2D and 3D convolution, respectively.
- **dropout_p** – (int) Dropout probability.
- **bilinear** – (bool) Use bilinear for up-sampling or not.

**forward**(*x1*, *x2*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**training:** bool

### pymic.net.net3d.unet3d module

class pymic.net.net3d.unet3d.**ConvBlock**(*in_channels*, *out_channels*, *dropout_p*)

Bases: `Module`

Two 3D convolution layers with batch norm and leaky relu. Droput is used between the two convolution layers.

**Parameters**

- **in_channels** – (int) Input channel number.
- **out_channels** – (int) Output channel number.
- **dropout_p** – (int) Dropout probability.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**training:** bool

class pymic.net.net3d.unet3d.**Decoder**(*params*)

Bases: `Module`

Decoder of 3D UNet.

Parameters are given in the *params* dictionary, and should include the following fields:

**Parameters**

- **in_chns** – (int) Input channel number.
- **feature_chns** – (list) Feature channel for each resolution level. The length should be 4 or 5, such as [16, 32, 64, 128, 256].
- **dropout** – (list) The dropout ratio for each resolution level. The length should be the same as that of *feature_chns*.
- **class_num** – (int) The class number for segmentation task.
- **trilinear** – (bool) Using bilinear for up-sampling or not. If False, deconvolution will be used for up-sampling.
- **multiscale_pred** – (bool) Get multi-scale prediction.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training:** bool

class pymic.net.net3d.unet3d.**DownBlock**(*in_channels*, *out_channels*, *dropout_p*)

Bases: `Module`

3D downsampling followed by ConvBlock

**Parameters**

- **in_channels** – (int) Input channel number.
- **out_channels** – (int) Output channel number.
- **dropout_p** – (int) Dropout probability.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training:** bool

class pymic.net.net3d.unet3d.**Encoder**(*params*)

Bases: `Module`

Encoder of 3D UNet.

Parameters are given in the *params* dictionary, and should include the following fields:

**Parameters**

- **in_chns** – (int) Input channel number.

- **feature_chns** – (list) Feature channel for each resolution level. The length should be 4 or 5, such as [16, 32, 64, 128, 256].
- **dropout** – (list) The dropout ratio for each resolution level. The length should be the same as that of *feature_chns*.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**training: bool**

class pymic.net.net3d.unet3d.**UNet3D**(*params*)

Bases: `Module`

An implementation of the U-Net.

- Reference: Özgün Çiçek, Ahmed Abdulkadir, Soeren S. Lienkamp, Thomas Brox, Olaf Ronneberger: 3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation. MICCAI (2) 2016: 424-432.

Note that there are some modifications from the original paper, such as the use of batch normalization, dropout, leaky relu and deep supervision.

Parameters are given in the *params* dictionary, and should include the following fields:

**Parameters**

- **in_chns** – (int) Input channel number.
- **feature_chns** – (list) Feature channel for each resolution level. The length should be 4 or 5, such as [16, 32, 64, 128, 256].
- **dropout** – (list) The dropout ratio for each resolution level. The length should be the same as that of *feature_chns*.
- **class_num** – (int) The class number for segmentation task.
- **trilinear** – (bool) Using trilinear for up-sampling or not. If False, deconvolution will be used for up-sampling.
- **multiscale_pred** – (bool) Get multi-scale prediction.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**training: bool**

**class** pymic.net.net3d.unet3d.**UpBlock**(*in_channels1*, *in_channels2*, *out_channels*, *dropout_p*, *trilinear=True*)

Bases: `Module`

3D upsampling followed by ConvBlock

**Parameters**

- **in_channels1** – (int) Channel number of high-level features.
- **in_channels2** – (int) Channel number of low-level features.
- **out_channels** – (int) Output channel number.
- **dropout_p** – (int) Dropout probability.
- **trilinear** – (bool) Use trilinear for up-sampling (by default). If False, deconvolution is used for up-sampling.

**forward**(*x1*, *x2*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training: bool**

### pymic.net.net3d.unet3d_scse module

**class** pymic.net.net3d.unet3d_scse.**ConvScSEBlock3D**(*in_channels*, *out_channels*, *dropout_p*)

Bases: `Module`

Two 3D convolutional blocks followed by *ChannelSpatialSELayer3D*. Each block consists of *Conv3d* + *BatchNorm3d* + *LeakyReLU*. A dropout layer is used between the wo blocks.

**Parameters**

- **in_channels** – (int) Input channel number.
- **out_channels** – (int) Output channel number.
- **dropout_p** – (int) Dropout probability.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training: bool**

**class** pymic.net.net3d.unet3d_scse.**DownBlock**(*in_channels*, *out_channels*, *dropout_p*)

Bases: `Module`

3D Downsampling followed by *ConvScSEBlock3D*.

**Parameters**

- **in_channels** – (int) Input channel number.
- **out_channels** – (int) Output channel number.
- **dropout_p** – (int) Dropout probability.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training:** bool

**class** pymic.net.net3d.unet3d_scse.**UNet3D_ScSE**(*params*)

Bases: `Module`

Combining 3D U-Net with SCSE module.

- Reference: Abhijit Guha Roy, Nassir Navab, Christian Wachinger: Recalibrating Fully Convolutional Networks With Spatial and Channel "Squeeze and Excitation" Blocks. IEEE Trans. Med. Imaging 38(2): 540-549 (2019).

Parameters are given in the *params* dictionary, and should include the following fields:

**Parameters**

- **in_chns** – (int) Input channel number.
- **feature_chns** – (list) Feature channel for each resolution level. The length should be 5, such as [16, 32, 64, 128, 256].
- **dropout** – (list) The dropout ratio for each resolution level. The length should be the same as that of *feature_chns*.
- **class_num** – (int) The class number for segmentation task.
- **trilinear** – (bool) Using trilinear for up-sampling or not. If False, deconvolution will be used for up-sampling.

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training: bool**

**class** pymic.net.net3d.unet3d_scse.**UpBlock**(*in_channels1*, *in_channels2*, *out_channels*, *dropout_p*, *trilinear=True*)

Bases: `Module`

3D Up-sampling followed by *ConvScSEBlock3D* in UNet3D_ScSE.

**Parameters**

- **in_channels1** – (int) Input channel number for low-resolution feature map.
- **in_channels2** – (int) Input channel number for high-resolution feature map.
- **out_channels** – (int) Output channel number.
- **dropout_p** – (int) Dropout probability.
- **trilinear** – (bool) Use trilinear for up-sampling or not.

**forward**(*x1*, *x2*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**training: bool**

#### Module contents

### 3.4.2 Submodules

### 3.4.3 pymic.net.net_dict_cls module

Built-in networks for classification.

- resnet18 *pymic.net.cls.torch_pretrained_net.ResNet18*
- vgg16 *pymic.net.cls.torch_pretrained_net.VGG16*
- mobilenetv2 *pymic.net.cls.torch_pretrained_net.MobileNetV2*

### 3.4.4 pymic.net.net_dict_seg module

Built-in networks for segmentation.

- UNet2D *pymic.net.net2d.unet2d.UNet2D*
- UNet2D_DualBranch *pymic.net.net2d.unet2d_dual_branch.UNet2D_DualBranch*
- UNet2D_CCT *pymic.net.net2d.unet2d_cct.UNet2D_CCT*
- UNet2D_ScSE *pymic.net.net2d.unet2d_scse.UNet2D_ScSE*
- AttentionUNet2D *pymic.net.net2d.unet2d_attention.AttentionUNet2D*

- NestedUNet2D *pymic.net.net2d.unet2d_nest.NestedUNet2D*
- COPLENet *pymic.net.net2d.cople_net.COPLENet*
- UNet2D5 *pymic.net.net3d.unet2d5.UNet2D5*
- UNet3D *pymic.net.net3d.unet3d.UNet3D*
- UNet3D_ScSE *pymic.net.net3d.unet3d_scse.UNet3D_ScSE*

### 3.4.5 Module contents

## 3.5 pymic.net_run package

### 3.5.1 Subpackages

#### pymic.net_run.semi_sup package

#### Submodules

#### pymic.net_run.semi_sup.ssl_abstract module

**class** pymic.net_run.semi_sup.ssl_abstract.**SSLSegAgent**(*config*, *stage='train'*)

Bases: *SegmentationAgent*

Abstract class for semi-supervised segmentation.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

---

**Note:** In the configuration dictionary, in addition to the four sections (*dataset*, *network*, *training* and *inference*) used in fully supervised learning, an extra section *semi_supervised_learning* is needed. See *Semi-Supervised Learning* for details.

---

**create_dataset**()

Create datasets for training, validation or testing based on configuraiton.

**get_unlabeled_dataset_from_config**()

Create a dataset for the unlabeled images based on configuration.

**train_valid**()

Train and valid.

**write_scalars**(*train_scalars*, *valid_scalars*, *lr_value*, *glob_it*)

Write scalars using SummaryWriter.

**Parameters**

- **train_scalars** – (dictionary) Scalars for training set.
- **valid_scalars** – (dictionary) Scalars for validation set.
- **lr_value** – (float) Current learning rate.

- **glob_it** – (int) Current iteration number.

### pymic.net_run.semi_sup.ssl_cct module

class pymic.net_run.semi_sup.ssl_cct.**SSLCCT**(*config*, *stage='train'*)

Bases: *SSLSegAgent*

Cross-Consistency Training for semi-supervised segmentation. It requires a network with multiple decoders for learning, such as *pymic.net.net2d.unet2d_cct.UNet2D_CCT*.

- Reference: Yassine Ouali, Celine Hudelot and Myriam Tami: Semi-Supervised Semantic Segmentation With Cross-Consistency Training. CVPR 2020.

The Code is adapted from Github

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

---

**Note:** In the configuration dictionary, in addition to the four sections (*dataset*, *network*, *training* and *inference*) used in fully supervised learning, an extra section *semi_supervised_learning* is needed. See *Semi-Supervised Learning* for details.

---

**training**()

Train the network

pymic.net_run.semi_sup.ssl_cct.**softmax_js_loss**(*inputs*, *targets*, ***_*)

pymic.net_run.semi_sup.ssl_cct.**softmax_kl_loss**(*inputs*, *targets*, *conf_mask=False*, *threshold=None*, *use_softmax=False*)

pymic.net_run.semi_sup.ssl_cct.**softmax_mse_loss**(*inputs*, *targets*, *conf_mask=False*, *threshold=None*, *use_softmax=False*)

### pymic.net_run.semi_sup.ssl_cps module

class pymic.net_run.semi_sup.ssl_cps.**BiNet**(*params*)

Bases: `Module`

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**training**: bool

**class** pymic.net_run.semi_sup.ssl_cps.**SSLCPS**(*config*, *stage='train'*)

Bases: *SSLSegAgent*

Using cross pseudo supervision for semi-supervised segmentation.

- Reference: Xiaokang Chen, Yuhui Yuan, Gang Zeng, Jingdong Wang, Semi-Supervised Semantic Segmentation with Cross Pseudo Supervision, CVPR 2021, pp. 2613-2022.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

---

**Note:** In the configuration dictionary, in addition to the four sections (*dataset*, *network*, *training* and *inference*) used in fully supervised learning, an extra section *semi_supervised_learning* is needed. See *Semi-Supervised Learning* for details.

---

**create_network**()

Create network based on configuration.

**training**()

Train the network

**write_scalars**(*train_scalars*, *valid_scalars*, *lr_value*, *glob_it*)

Write scalars using SummaryWriter.

**Parameters**

- **train_scalars** – (dictionary) Scalars for training set.
- **valid_scalars** – (dictionary) Scalars for validation set.
- **lr_value** – (float) Current learning rate.
- **glob_it** – (int) Current iteration number.

### pymic.net_run.semi_sup.ssl_em module

**class** pymic.net_run.semi_sup.ssl_em.**SSLEntropyMinimization**(*config*, *stage='train'*)

Bases: *SSLSegAgent*

Using Entropy Minimization for semi-supervised segmentation.

- Reference: Yves Grandvalet and Yoshua Bengio: Semi-supervised Learningby Entropy Minimization. NeurIPS, 2005.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

---

**Note:** In the configuration dictionary, in addition to the four sections (*dataset*, *network*, *training* and *inference*) used in fully supervised learning, an extra section *semi_supervised_learning* is needed. See *Semi-Supervised Learning* for details.

---

**training()**
Train the network

### pymic.net_run.semi_sup.ssl_mt module

class pymic.net_run.semi_sup.ssl_mt.**SSLMeanTeacher**(*config*, *stage='train'*)
Bases: *SSLSegAgent*

Mean Teacher for semi-supervised segmentation.

- Reference: Antti Tarvainen, Harri Valpola: Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results. NeurIPS 2017.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

**Note:** In the configuration dictionary, in addition to the four sections (*dataset*, *network*, *training* and *inference*) used in fully supervised learning, an extra section *semi_supervised_learning* is needed. See *Semi-Supervised Learning* for details.

**create_network()**
Create network based on configuration.

**training()**
Train the network

### pymic.net_run.semi_sup.ssl_uamt module

class pymic.net_run.semi_sup.ssl_uamt.**SSLUncertaintyAwareMeanTeacher**(*config*, *stage='train'*)
Bases: *SSLMeanTeacher*

Uncertainty Aware Mean Teacher for semi-supervised segmentation.

- Reference: Lequan Yu, Shujun Wang, Xiaomeng Li, Chi-Wing Fu, and Pheng-Ann Heng. Uncertainty-aware Self-ensembling Model for Semi-supervised 3D Left Atrium Segmentation, MICCAI 2019.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

**Note:** In the configuration dictionary, in addition to the four sections (*dataset*, *network*, *training* and *inference*) used in fully supervised learning, an extra section *semi_supervised_learning* is needed. See *Semi-Supervised Learning* for details.

**training()**
Train the network

### pymic.net_run.semi_sup.ssl_urpc module

**class** `pymic.net_run.semi_sup.ssl_urpc.SSLURPC`(*config*, *stage='train'*)

Bases: *SSLSegAgent*

Uncertainty-Rectified Pyramid Consistency for semi-supervised segmentation.

- Reference: Xiangde Luo, Guotai Wang*, Wenjun Liao, Jieneng Chen, Tao Song, Yinan Chen, Shichuan Zhang, Dimitris N. Metaxas, Shaoting Zhang. Semi-Supervised Medical Image Segmentation via Uncertainty Rectified Pyramid Consistency . Medical Image Analysis 2022.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

---

**Note:** In the configuration dictionary, in addition to the four sections (*dataset*, *network*, *training* and *inference*) used in fully supervised learning, an extra section *semi_supervised_learning* is needed. See *Semi-Supervised Learning* for details.

---

**training()**

Train the network

### Module contents

### pymic.net_run.weak_sup package

### Submodules

### pymic.net_run.weak_sup.wsl_abstract module

**class** `pymic.net_run.weak_sup.wsl_abstract.WSLSegAgent`(*config*, *stage='train'*)

Bases: *SegmentationAgent*

Abstract agent for weakly supervised segmentation.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

---

**Note:** In the configuration dictionary, in addition to the four sections (*dataset*, *network*, *training* and *inference*) used in fully supervised learning, an extra section *weakly_supervised_learning* is needed. See *Weakly-Supervised Learning* for details.

---

**write_scalars**(*train_scalars*, *valid_scalars*, *lr_value*, *glob_it*)

Write scalars using SummaryWriter.

**Parameters**

- **train_scalars** – (dictionary) Scalars for training set.

- **valid_scalars** – (dictionary) Scalars for validation set.
- **lr_value** – (float) Current learning rate.
- **glob_it** – (int) Current iteration number.

### pymic.net_run.weak_sup.wsl_dmpls module

`class pymic.net_run.weak_sup.wsl_dmpls.WSLDMPLS(config, stage='train')`

Bases: *WSLSegAgent*

Weakly supervised segmentation based on Dynamically Mixed Pseudo Labels Supervision.

- Reference: Xiangde Luo, Minhao Hu, Wenjun Liao, Shuwei Zhai, Tao Song, Guotai Wang, Shaoting Zhang. ScribblScribble-Supervised Medical Image Segmentation via Dual-Branch Network and Dynamically Mixed Pseudo Labels Supervision. MICCAI 2022.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

**Note:** In the configuration dictionary, in addition to the four sections (*dataset*, *network*, *training* and *inference*) used in fully supervised learning, an extra section *weakly_supervised_learning* is needed. See *Weakly-Supervised Learning* for details.

`training()`

Train the network

### pymic.net_run.weak_sup.wsl_em module

`class pymic.net_run.weak_sup.wsl_em.WSLEntropyMinimization(config, stage='train')`

Bases: *WSLSegAgent*

Weakly supervised segmentation based on Entropy Minimization.

- Reference: Yves Grandvalet and Yoshua Bengio: Semi-supervised Learningby Entropy Minimization. NeurIPS, 2005.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

**Note:** In the configuration dictionary, in addition to the four sections (*dataset*, *network*, *training* and *inference*) used in fully supervised learning, an extra section *weakly_supervised_learning* is needed. See *Weakly-Supervised Learning* for details.

`training()`

Train the network

### pymic.net_run.weak_sup.wsl_gatedcrf module

class pymic.net_run.weak_sup.wsl_gatedcrf.**WSLGatedCRF**(*config*, *stage='train'*)

Bases: *WSLSegAgent*

Implementation of the Gated CRF loss for weakly supervised segmentation.

- Reference: Anton Obukhov, Stamatios Georgoulis, Dengxin Dai, Luc Van Gool: Gated CRF Loss for Weakly Supervised Semantic Image Segmentation. CoRR, abs/1906.04651, 2019.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

---

**Note:** In the configuration dictionary, in addition to the four sections (*dataset*, *network*, *training* and *inference*) used in fully supervised learning, an extra section *weakly_supervised_learning* is needed. See *Weakly-Supervised Learning* for details.

---

**training**()

Train the network

### pymic.net_run.weak_sup.wsl_mumford_shah module

class pymic.net_run.weak_sup.wsl_mumford_shah.**WSLMumfordShah**(*config*, *stage='train'*)

Bases: *WSLSegAgent*

Weakly supervised learning with Mumford Shah Loss.

- Reference: Boah Kim and Jong Chul Ye: Mumford–Shah Loss Functional for Image Segmentation With Deep Learning. IEEE TIP, 2019.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

---

**Note:** In the configuration dictionary, in addition to the four sections (*dataset*, *network*, *training* and *inference*) used in fully supervised learning, an extra section *weakly_supervised_learning* is needed. See *Weakly-Supervised Learning* for details.

---

**training**()

Train the network

### pymic.net_run.weak_sup.wsl_tv module

class pymic.net_run.weak_sup.wsl_tv.**WSLTotalVariation**(*config*, *stage='train'*)

Bases: *WSLSegAgent*

Weakly suepervised segmentation with Total Variation regularization.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

**Note:** In the configuration dictionary, in addition to the four sections (*dataset*, *network*, *training* and *inference*) used in fully supervised learning, an extra section *weakly_supervised_learning* is needed. See *Weakly-Supervised Learning* for details.

**training**()

Train the network

### pymic.net_run.weak_sup.wsl_ustm module

class pymic.net_run.weak_sup.wsl_ustm.**WSLUSTM**(*config*, *stage='train'*)

Bases: *WSLSegAgent*

USTM for scribble-supervised segmentation.

- Reference: Xiaoming Liu, Quan Yuan, Yaozong Gao, Helei He, Shuo Wang, Xiao Tang, Jinshan Tang, Dinggang Shen: Weakly Supervised Segmentation of COVID19 Infection with Scribble Annotation on CT Images. Patter Recognition, 2022.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

**Note:** In the configuration dictionary, in addition to the four sections (*dataset*, *network*, *training* and *inference*) used in fully supervised learning, an extra section *weakly_supervised_learning* is needed. See *Weakly-Supervised Learning* for details.

**create_network**()

Create network based on configuration.

**training**()

Train the network

### Module contents

### pymic.net_run.noisy_label package

#### Submodules

#### pymic.net_run.noisy_label.nll_clslsr module

**class** pymic.net_run.noisy_label.nll_clslsr.**NLLCLSLSR**(*config*, *stage='test'*)

Bases: *SegmentationAgent*

An agent to estimatate the confidence of noisy labels during inference.

- Reference: Minqing Zhang et al., Characterizing Label Errors: Confident Learning for Noisy-Labeled Image Segmentation, MICCAI 2020.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

**infer_with_cl**()

Inference with confidence estimation.

pymic.net_run.noisy_label.nll_clslsr.**get_confidence_map**(*cfg_file*)

pymic.net_run.noisy_label.nll_clslsr.**get_confident_map**(*gt*, *pred*, *CL_type='both'*)

Get the confidence map based on the label and prediction.

**Parameters**

- **gt** – (tensor) One-hot label with shape of NXC.
- **pred** – (tensor) Digit prediction of network with shape of NXC.
- **CL_type** – (str) A string in {'both', 'Qij', 'Cij', 'intersection', 'union', 'prune_by_class', 'prune_by_noise_rate'}.

**Returns**

A tensor representing the noisiness of each pixel.

#### pymic.net_run.noisy_label.nll_co_teaching module

**class** pymic.net_run.noisy_label.nll_co_teaching.**BiNet**(*params*)

Bases: Module

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training**: bool

class pymic.net_run.noisy_label.nll_co_teaching.**NLLCoTeaching**(*config*, *stage='train'*)

Bases: *SegmentationAgent*

Co-teaching for noisy-label learning.

- Reference: Bo Han, Quanming Yao, Xingrui Yu, Gang Niu, Miao Xu, Weihua Hu, Ivor Tsang, Masashi Sugiyama. Co-teaching: Robust Training of Deep Neural Networks with Extremely Noisy Labels. NeurIPS 201.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

---

**Note:** In the configuration dictionary, in addition to the four sections (*dataset*, *network*, *training* and *inference*) used in fully supervised learning, an extra section *noisy_label_learning* is needed. See *Noisy Label Learning* for details.

---

**create_network**()

Create network based on configuration.

**training**()

Train the network

**write_scalars**(*train_scalars*, *valid_scalars*, *lr_value*, *glob_it*)

Write scalars using SummaryWriter.

**Parameters**

- **train_scalars** – (dictionary) Scalars for training set.
- **valid_scalars** – (dictionary) Scalars for validation set.
- **lr_value** – (float) Current learning rate.
- **glob_it** – (int) Current iteration number.

## pymic.net_run.noisy_label.nll_dast module

class pymic.net_run.noisy_label.nll_dast.**ConsistLoss**

Bases: Module

**forward**(*input1*, *input2*, *size_average=True*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**kl_div_map**(*input*, *label*)

**kl_loss**(*input*, *target*, *size_average=True*)

**training:** bool

class pymic.net_run.noisy_label.nll_dast.**NLLDAST**(*config*, *stage='train'*)

Bases: *SegmentationAgent*

Divergence-Aware Selective Training for noisy label learning.

- Reference: Shuojue Yang, Guotai Wang, Hui Sun, Xiangde Luo, Peng Sun, Kang Li, Qijun Wang, Shaoting Zhang: Learning COVID-19 Pneumonia Lesion Segmentation from Imperfect Annotations via Divergence-Aware Selective Training. JBHI 2022.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

---

**Note:** In the configuration dictionary, in addition to the four sections (*dataset*, *network*, *training* and *inference*) used in fully supervised learning, an extra section *noisy_label_learning* is needed. See *Noisy Label Learning* for details.

---

**create_dataset**()

Create datasets for training, validation or testing based on configuraiton.

**get_noisy_dataset_from_config**()

Create a dataset for images with noisy labels based on configuraiton.

**train_valid**()

Train and valid.

**training**()

Train the network

class pymic.net_run.noisy_label.nll_dast.**Rank**(*quene_length=100*)

Bases: object

Dynamically rank the current training sample with specific metrics.

**Parameters**

**quene_length** – (int) The lenght for a quene.

**add_val**(*val*)

Update the quene and calculate the order of the input value.

**Parameters**

**val** – (float) a value adding to the quene.

**Returns**

rank of the input value with a range of (0, self.quene_length)

pymic.net_run.noisy_label.nll_dast.**get_ce**(*prob*, *soft_y*, *size_avg=True*)

pymic.net_run.noisy_label.nll_dast.**select_criterion**(*no_noisy_sample*, *cl_noisy_sample*, *label*)

Obtain the sample selection criterion score.

**Parameters**

- **no_noisy_sample** – noisy branch's output probability for noisy sample.
- **cl_noisy_sample** – clean branch's output probability for noisy sample.
- **label** – noisy label.

## pymic.net_run.noisy_label.nll_trinet module

class pymic.net_run.noisy_label.nll_trinet.**NLLTriNet**(*config*, *stage='train'*)

Bases: `SegmentationAgent`

Implementation of trinet for learning from noisy samples for segmentation tasks.

- Reference: Tianwei Zhang, Lequan Yu, Na Hu, Su Lv, Shi Gu: Robust Medical Image Segmentation from Non-expert Annotations with Tri-network. MICCAI 2020.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

---

**Note:** In the configuration dictionary, in addition to the four sections (*dataset*, *network*, *training* and *inference*) used in fully supervised learning, an extra section *noisy_label_learning* is needed. See *Noisy Label Learning* for details.

---

**create_network**()

Create network based on configuration.

**get_loss_and_confident_mask**(*pred*, *labels_prob*, *conf_ratio*)

**training**()

Train the network

**write_scalars**(*train_scalars*, *valid_scalars*, *lr_value*, *glob_it*)

Write scalars using SummaryWriter.

**Parameters**

- **train_scalars** – (dictionary) Scalars for training set.
- **valid_scalars** – (dictionary) Scalars for validation set.
- **lr_value** – (float) Current learning rate.
- **glob_it** – (int) Current iteration number.

class pymic.net_run.noisy_label.nll_trinet.**TriNet**(*params*)

Bases: `Module`

**forward**(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training:** `bool`

### Module contents

## 3.5.2 Submodules

## 3.5.3 pymic.net_run.agent_abstract module

class pymic.net_run.agent_abstract.**NetRunAgent**(*config*, *stage='train'*)

Bases: `object`

The abstract class for medical image segmentation.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

**Note:** The config dictionary should have at least four sections: *dataset*, *network*, *training* and *inference*. See *Quick Start* and *Fully Supervised Learning* for example.

**convert_tensor_type**(*input_tensor*)

Convert the type of an input tensor to float or double based on configuration.

**create_dataset**()

Create datasets for training, validation or testing based on configuraiton.

abstract **create_loss_calculator**()

Create loss function object.

abstract **create_network**()

Create network based on configuration.

**create_optimizer**(*params*, *checkpoint=None*)

Create optimizer based on configuration.

**Parameters**

- **params** – network parameters for optimization. Usually it is obtained by *self.get_parameters_to_update()*.
- **checkpoint** – A previous checkpoint to load. Default is *None*.

**get_checkpoint_name**()

Get the checkpoint name for inference based on config['testing']['ckpt_mode'].

**abstract get_loss_value**(*data*, *pred*, *gt*, *param=None*)

Get the loss value. Assume *pred* and *gt* has been sent to self.device. *data* is obtained by dataloader, and is a dictionary containing extra information, such as pixel-level weight. By default, such information is not used by standard loss functions such as Dice loss and cross entropy loss.

**Parameters**

- **data** – (dictionary) A data dictionary obtained by dataloader.
- **pred** – (tensor) Prediction result by the network.
- **gt** – (tensor) Ground truth.
- **param** – (dictionary) Other parameters if needed.

**abstract get_parameters_to_update**()

Get parameters for update.

**abstract get_stage_dataset_from_config**(*stage*)

Create dataset based on training, validation or inference stage.

**Parameters**

**stage** – (str) *train*, *valid* or *test*.

**abstract infer**()

Inference on testing set.

**run**()

Run the training or inference code according to configuration.

**set_datasets**(*train_set*, *valid_set*, *test_set*)

Set customized datasets for training and inference.

**Parameters**

- **train_set** – (torch.utils.data.Dataset) The training set.
- **valid_set** – (torch.utils.data.Dataset) The validation set.
- **test_set** – (torch.utils.data.Dataset) The testing set.

**set_inferer**(*inferer*)

Set the inferer.

**Parameters**

**inferer** – An inferer object.

**set_loss_dict**(*loss_dict*)

Set the available loss functions, including customized loss functions.

**Parameters**

**loss_dict** – (dictionary) A dictionary of available loss functions.

**set_net_dict**(*net_dict*)

Set the available networks, including customized networks.

**Parameters**

**net_dict** – (dictionary) A dictionary of available networks.

**set_network**(*net*)

Set the network.

**Parameters**
**net** – (nn.Module) A deep learning network.

**set_optimizer**(*optimizer*)
Set the optimizer.

**Parameters**
**optimizer** – An optimizer.

**set_scheduler**(*scheduler*)
Set the learning rate scheduler.

**Parameters**
**scheduler** – A learning rate scheduler.

**set_transform_dict**(*custom_transform_dict*)
Set the available Transforms, including customized Transforms.

**Parameters**
**custom_transform_dict** – (dictionary) A dictionary of available Transforms.

**abstract train_valid**()
Train and valid.

**abstract training**()
Train the network

**abstract validation**()
Evaluate the performance on the validation set.

**abstract write_scalars**(*train_scalars*, *valid_scalars*, *lr_value*, *glob_it*)
Write scalars using SummaryWriter.

**Parameters**

- **train_scalars** – (dictionary) Scalars for training set.
- **valid_scalars** – (dictionary) Scalars for validation set.
- **lr_value** – (float) Current learning rate.
- **glob_it** – (int) Current iteration number.

pymic.net_run.agent_abstract.**seed_torch**(*seed=1*)
Set random seed.

**Parameters**
**seed** – (int) the seed for random.

### 3.5.4 pymic.net_run.agent_cls module

**class** pymic.net_run.agent_cls.**ClassificationAgent**(*config*, *stage='train'*)
Bases: NetRunAgent

The agent for image classificaiton tasks.

**Parameters**

- **config** – (dict) A dictionary containing the configuration.
- **stage** – (str) One of the stage in *train* (default), *inference* or *test*.

**Note:** The config dictionary should have at least four sections: *dataset*, *network*, *training* and *inference*. See *Quick Start* and *Fully Supervised Learning* for example.

**create_loss_calculator**()

Create loss function object.

**create_network**()

Create network based on configuration.

**get_evaluation_score**(*outputs*, *labels*)

Get evaluation score for a prediction.

**Parameters**

- **outputs** – (tensor) Prediction obtained by a network with size N X C.
- **labels** – (tensor) The ground truth with size N X C.

**get_loss_value**(*data*, *pred*, *gt*, *param=None*)

Get the loss value. Assume *pred* and *gt* has been sent to self.device. *data* is obtained by dataloader, and is a dictionary containing extra information, such as pixel-level weight. By default, such information is not used by standard loss functions such as Dice loss and cross entropy loss.

**Parameters**

- **data** – (dictionary) A data dictionary obtained by dataloader.
- **pred** – (tensor) Prediction result by the network.
- **gt** – (tensor) Ground truth.
- **param** – (dictionary) Other parameters if needed.

**get_parameters_to_update**()

Get parameters for update.

**get_stage_dataset_from_config**(*stage*)

Create dataset based on training, validation or inference stage.

**Parameters**

**stage** – (str) *train*, *valid* or *test*.

**infer**()

Inference on testing set.

**train_valid**()

Train and valid.

**training**()

Train the network

**validation**()

Evaluate the performance on the validation set.

**write_scalars**(*train_scalars*, *valid_scalars*, *lr_value*, *glob_it*)

Write scalars using SummaryWriter.

**Parameters**

- **train_scalars** – (dictionary) Scalars for training set.

- **valid_scalars** – (dictionary) Scalars for validation set.
- **lr_value** – (float) Current learning rate.
- **glob_it** – (int) Current iteration number.

pymic.net_run.agent_cls.**random**() → x in the interval [0, 1).

### 3.5.5 pymic.net_run.agent_seg module

**class** pymic.net_run.agent_seg.**SegmentationAgent**(*config*, *stage='train'*)

Bases: *NetRunAgent*

**create_loss_calculator**()

Create loss function object.

**create_network**()

Create network based on configuration.

**get_loss_value**(*data*, *pred*, *gt*, *param=None*)

Get the loss value. Assume *pred* and *gt* has been sent to self.device. *data* is obtained by dataloader, and is a dictionary containing extra information, such as pixel-level weight. By default, such information is not used by standard loss functions such as Dice loss and cross entropy loss.

**Parameters**

- **data** – (dictionary) A data dictionary obtained by dataloader.
- **pred** – (tensor) Prediction result by the network.
- **gt** – (tensor) Ground truth.
- **param** – (dictionary) Other parameters if needed.

**get_parameters_to_update**()

Get parameters for update.

**get_stage_dataset_from_config**(*stage*)

Create dataset based on training, validation or inference stage.

**Parameters**

**stage** – (str) *train*, *valid* or *test*.

**infer**()

Inference on testing set.

**infer_with_multiple_checkpoints**()

Inference with ensemble of multilple check points.

**save_outputs**(*data*)

Save prediction output.

**Parameters**

**data** – (dictionary) A data dictionary with prediciton result and other information such as input image name.

**set_postprocessor**(*postprocessor*)

Set post processor after prediction.

**Parameters**
**postprocessor** – post processor, such as an instance of *pymic.util.post_process.PostProcess*.

**train_valid**()
Train and valid.

**training**()
Train the network

**validation**()
Evaluate the performance on the validation set.

**write_scalars**(*train_scalars*, *valid_scalars*, *lr_value*, *glob_it*)
Write scalars using SummaryWriter.

**Parameters**

- **train_scalars** – (dictionary) Scalars for training set.
- **valid_scalars** – (dictionary) Scalars for validation set.
- **lr_value** – (float) Current learning rate.
- **glob_it** – (int) Current iteration number.

pymic.net_run.agent_seg.**random**() → x in the interval [0, 1).

### 3.5.6 pymic.net_run.get_optimizer module

pymic.net_run.get_optimizer.**get_lr_scheduler**(*optimizer*, *sched_params*)
Create learning rate scheduler for an optimizer

**Parameters**

- **optimizer** – An optimizer instance.
- **sched_params** – (dict) The parameters required for the scheduler.

**Returns**
An instance of the target learning rate scheduler.

pymic.net_run.get_optimizer.**get_optimizer**(*name*, *net_params*, *optim_params*)
Create an optimizer for learnable parameters.

**Parameters**

- **name** – (string) Name of the optimizer. Should be one of {*SGD*, *Adam*, *SparseAdam*, *Adadelta*, *Adagrad*, *Adamax*, *ASGD*, *LBFGS*, *RMSprop*, *Rprop*}.
- **net_params** – Learnable parameters that need to be set for an optimizer.
- **optim_params** – (dict) The parameters required for the target optimizer.

**Returns**
An instance of the target optimizer.

### 3.5.7 pymic.net_run.infer_func module

**class** pymic.net_run.infer_func.**Inferer**(*config*)

Bases: `object`

The class for inference. The arguments should be written in the *config* dictionary, and it has the following fields:

**Parameters**

- **sliding_window_enable** – (optional, bool) Default is *False*.
- **sliding_window_size** – (optional, list) The sliding window size.
- **sliding_window_stride** – (optional, list) The sliding window stride.
- **tta_mode** – (optional, int) The test time augmentation mode. Default is 0 (no test time augmentation). The other option is 1 (augmentation with horinzontal and vertical flipping) and 2 (ensemble of inference in axial, sagittal and coronal views for 2D networks applied to 3D volumes)

**run**(*model, image*)

Using *model* for inference on *image*.

**Parameters**

- **model** – (nn.Module) a network.
- **image** – (tensor) An image.

### 3.5.8 Module contents

## 3.6 pymic.transform package

### 3.6.1 Submodules

### 3.6.2 pymic.transform.abstract_transform module

**class** pymic.transform.abstract_transform.**AbstractTransform**(*params*)

Bases: `object`

The abstract class for Transform.

**inverse_transform_for_prediction**(*sample*)

Inverse transform for the sample dictionary. Especially, it will update sample['predict'] obtained by a network's prediction based on the inverse transform. This function is only useful for spatial transforms.

### 3.6.3 pymic.transform.crop module

**class** pymic.transform.crop.**CenterCrop**(*params*)

Bases: *AbstractTransform*

Crop the given image at the center. Input shape should be [C, D, H, W] or [C, H, W].

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **CenterCrop_output_size** – (list or tuple) The output size. [D, H, W] for 3D images and [H, W] for 2D images. If D is None, then the z-axis is not cropped.
- **CenterCrop_inverse** – (optional, bool) Is inverse transform needed for inference. Default is *True*.

**inverse_transform_for_prediction**(*sample*)

Inverse transform for the sample dictionary. Especially, it will update sample['predict'] obtained by a network's prediction based on the inverse transform. This function is only useful for spatial transforms.

class pymic.transform.crop.**CropWithBoundingBox**(*params*)

Bases: *CenterCrop*

Crop the image (shape [C, D, H, W] or [C, H, W]) based on a bounding box. The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **CropWithBoundingBox_start** – (None, or list/tuple) The start index along each spatial axis. If None, calculate the start index automatically so that the cropped region is centered at the non-zero region.
- **CropWithBoundingBox_output_size** – (None or tuple/list): Desired spatial output size. If None, set it as the size of bounding box of non-zero region.
- **CropWithBoundingBox_inverse** – (optional, bool) Is inverse transform needed for inference. Default is *True*.

class pymic.transform.crop.**RandomCrop**(*params*)

Bases: *CenterCrop*

Randomly crop the input image (shape [C, D, H, W] or [C, H, W]).

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **RandomCrop_output_size** – (list/tuple) Desired output size [D, H, W] or [H, W]. The output channel is the same as the input channel. If D is None for 3D images, the z-axis is not cropped.
- **RandomCrop_foreground_focus** – (optional, bool) If true, allow crop around the foreground. Default is False.
- **RandomCrop_foreground_ratio** – (optional, float) Specifying the probability of foreground focus cropping when *RandomCrop_foreground_focus* is True.
- **RandomCrop_mask_label** – (optional, None, or list/tuple) Specifying the foreground labels for foreground focus cropping when *RandomCrop_foreground_focus* is True.
- **RandomCrop_inverse** – (optional, bool) Is inverse transform needed for inference. Default is *True*.

class pymic.transform.crop.**RandomResizedCrop**(*params*)

Bases: *CenterCrop*

Randomly crop the input image (shape [C, H, W]). Only 2D images are supported.

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **RandomResizedCrop_output_size** – (list/tuple) Desired output size [H, W]. The output channel is the same as the input channel.

- **RandomResizedCrop_scale** – (list/tuple) Range of scale, e.g. (0.08, 1.0).
- **RandomResizedCrop_ratio** – (list/tuple) Range of aspect ratio, e.g. (0.75, 1.33).
- **RandomResizedCrop_inverse** – (optional, bool) Is inverse transform needed for inference. Default is *False*. Currently, the inverse transform is not supported, and this transform is assumed to be used only during training stage.

### 3.6.4 pymic.transform.flip module

**class** pymic.transform.flip.**RandomFlip**(*params*)

Bases: *AbstractTransform*

Random flip the image. The shape is [C, D, H, W] or [C, H, W].

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **RandomFlip_flip_depth** – (bool) Random flip along depth axis or not, only used for 3D images.
- **RandomFlip_flip_height** – (bool) Random flip along height axis or not.
- **RandomFlip_flip_width** – (bool) Random flip along width axis or not.
- **RandomFlip_inverse** – (optional, bool) Is inverse transform needed for inference. Default is *True*.

**inverse_transform_for_prediction**(*sample*)

Inverse transform for the sample dictionary. Especially, it will update sample['predict'] obtained by a network's prediction based on the inverse transform. This function is only useful for spatial transforms.

### 3.6.5 pymic.transform.intensity module

**class** pymic.transform.intensity.**GammaCorrection**(*params*)

Bases: *AbstractTransform*

Apply random gamma correction to given channels.

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **GammaCorrection_channels** – (list) A list of int for specifying the channels.
- **GammaCorrection_gamma_min** – (float) The minimal gamma value.
- **GammaCorrection_gamma_max** – (float) The maximal gamma value.
- **GammaCorrection_probability** – (optional, float) The probability of applying GammaCorrection. Default is 0.5.
- **GammaCorrection_inverse** – (optional, bool) Is inverse transform needed for inference. Default is *False*.

**class** pymic.transform.intensity.**GaussianNoise**(*params*)

Bases: *AbstractTransform*

Add Gaussian Noise to given channels.

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **GaussianNoise_channels** – (list) A list of int for specifying the channels.
- **GaussianNoise_mean** – (float) The mean value of noise.
- **GaussianNoise_std** – (float) The std of noise.
- **GaussianNoise_probability** – (optional, float) The probability of applying GaussianNoise. Default is 0.5.
- **GaussianNoise_inverse** – (optional, bool) Is inverse transform needed for inference. Default is *False*.

**class** pymic.transform.intensity.**GrayscaleToRGB**(*params*)

Bases: *AbstractTransform*

Convert gray scale images to RGB by copying channels.

**class** pymic.transform.intensity.**InOutPainting**(*params*)

Bases: *AbstractTransform*

Apply in-painting or out-patining randomly. They are mutually exclusive.

**class** pymic.transform.intensity.**InPainting**(*params*)

Bases: *AbstractTransform*

In-painting of an input image, used for self-supervised learning

**class** pymic.transform.intensity.**LocalShuffling**(*params*)

Bases: *AbstractTransform*

local pixel shuffling of an input image, used for self-supervised learning

**class** pymic.transform.intensity.**NonLinearTransform**(*params*)

Bases: *AbstractTransform*

**class** pymic.transform.intensity.**OutPainting**(*params*)

Bases: *AbstractTransform*

Out-painting of an input image, used for self-supervised learning

pymic.transform.intensity.**bernstein_poly**(*i*, *n*, *t*)

The Bernstein polynomial of n, i as a function of t.

pymic.transform.intensity.**bezier_curve**(*points*, *nTimes=1000*)

Given a set of control points, return the bezier curve defined by the control points. Control points should be a list of lists, or list of tuples such as [ [1,1], [2,3], [4,5], ..[Xn, Yn] ].

*nTimes* is the number of time steps, defaults to 1000. See http://processingjs.nihongoresources.com/bezierinfo/

### 3.6.6 pymic.transform.label_convert module

**class** pymic.transform.label_convert.**LabelConvert**(*params*)

Bases: *AbstractTransform*

Convert the label based on a source list and target list.

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **LabelConvert_source_list** – (list) A list of labels to be converted.
- **LabelConvert_target_list** – (list) The target label list.
- **LabelConvert_inverse** – (optional, bool) Is inverse transform needed for inference. Default is *False*.

class pymic.transform.label_convert.**LabelConvertNonzero**(*params*)

Bases: *AbstractTransform*

Convert label into binary, i.e., setting nonzero labels as 1.

class pymic.transform.label_convert.**LabelToProbability**(*params*)

Bases: *AbstractTransform*

Convert one-channel label map to one-hot multi-channel probability map.

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **LabelToProbability_class_num** – (int) The class number in the label map.
- **LabelToProbability_inverse** – (optional, bool) Is inverse transform needed for inference. Default is *False*.

class pymic.transform.label_convert.**PartialLabelToProbability**(*params*)

Bases: *AbstractTransform*

Convert one-channel partial label map to one-hot multi-channel probability map. This is used for segmentation tasks only. In the input label map, 0 represents the background class, 1 to C-1 represent the foreground classes, and C represents unlabeled pixels. In the output dictionary, *label_prob* is the one-hot probability map, and *pixel_weight* represents a weighting map, where the weight for a pixel is 0 if the label is unkown.

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **PartialLabelToProbability_class_num** – (int) The class number for the segmentation task.
- **PartialLabelToProbability_inverse** – (optional, bool) Is inverse transform needed for inference. Default is *False*.

class pymic.transform.label_convert.**ReduceLabelDim**(*params*)

Bases: *AbstractTransform*

Remove the first dimension of label tensor.

### 3.6.7 pymic.transform.normalize module

class pymic.transform.normalize.**NormalizeWithMeanStd**(*params*)

Bases: *AbstractTransform*

Normalize the image based on mean and std. The image should have a shape of [C, D, H, W] or [C, H, W].

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **NormalizeWithMeanStd_channels** – (list/tuple or None) A list or tuple of int for specifying the channels. If None, the transform operates on all the channels.

- `NormalizeWithMeanStd_mean` – (list/tuple or None) The mean values along each specified channel. If None, the mean values are calculated automatically.
- `NormalizeWithMeanStd_std` – (list/tuple or None) The std values along each specified channel. If None, the std values are calculated automatically.
- `NormalizeWithMeanStd_ignore_non_positive` – (optional, bool) Only used when mean and std are not given. Default is False. If True, calculate mean and std in the positive region for normalization, and set non-positive region to random. If False, calculate the mean and std values in the entire image region.
- `NormalizeWithMeanStd_inverse` – (optional, bool) Is inverse transform needed for inference. Default is *False*.

**class** `pymic.transform.normalize.NormalizeWithMinMax`(*params*)

Bases: *AbstractTransform*

Nomralize the image to [0, 1]. The shape should be [C, D, H, W] or [C, H, W].

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- `NormalizeWithMinMax_channels` – (list/tuple or None) A list or tuple of int for specifying the channels. If None, the transform operates on all the channels.
- `NormalizeWithMinMax_threshold_lower` – (list/tuple or None) The min values along each specified channel. If None, the min values are calculated automatically.
- `NormalizeWithMinMax_threshold_upper` – (list/tuple or None) The max values along each specified channel. If None, the max values are calculated automatically.
- `NormalizeWithMinMax_inverse` – (optional, bool) Is inverse transform needed for inference. Default is *False*.

**class** `pymic.transform.normalize.NormalizeWithPercentiles`(*params*)

Bases: *AbstractTransform*

Nomralize the image to [0, 1] with percentiles for given channels. The shape should be [C, D, H, W] or [C, H, W].

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- `NormalizeWithPercentiles_channels` – (list/tuple or None) A list or tuple of int for specifying the channels. If None, the transform operates on all the channels.
- `NormalizeWithPercentiles_percentile_lower` – (float) The min percentile, which must be between 0 and 100 inclusive.
- `NormalizeWithPercentiles_percentile_upper` – (float) The max percentile, which must be between 0 and 100 inclusive.
- `NormalizeWithMinMax_inverse` – (optional, bool) Is inverse transform needed for inference. Default is *False*.

### 3.6.8 pymic.transform.pad module

class pymic.transform.pad.**Pad**(*params*)

Bases: *AbstractTransform*

Pad an image to an new spatial shape. The image has a shape of [C, D, H, W] or [C, H, W]. The real output size will be max(image_size, output_size).

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **Pad_output_size** – (list/tuple) The output size along each spatial axis.
- **Pad_ceil_mode** – (optional, bool) If true (by default), the real output size will be the minimal integer multiples of output_size higher than the input size. For example, the input image has a shape of [3, 100, 100], *Pad_output_size* = [32, 32], and the real output size will be [3, 128, 128] if *Pad_ceil_mode* = True.
- **Pad_inverse** – (optional, bool) Is inverse transform needed for inference. Default is *True*.

**inverse_transform_for_prediction**(*sample*)

Inverse transform for the sample dictionary. Especially, it will update sample['predict'] obtained by a network's prediction based on the inverse transform. This function is only useful for spatial transforms.

### 3.6.9 pymic.transform.rescale module

class pymic.transform.rescale.**RandomRescale**(*params*)

Bases: *AbstractTransform*

Rescale the input image randomly along each spatial axis.

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **RandomRescale_lower_bound** – (list/tuple or float) Desired minimal rescale ratio. If tuple/list, the length should be 3 or 2.
- **RandomRescale_upper_bound** – (list/tuple or float) Desired maximal rescale ratio. If tuple/list, the length should be 3 or 2.
- **RandomRescale_probability** – (optional, float) The probability of applying RandomRescale. Default is 0.5.
- **RandomRescale_inverse** – (optional, bool) Is inverse transform needed for inference. Default is *True*.

**inverse_transform_for_prediction**(*sample*)

Inverse transform for the sample dictionary. Especially, it will update sample['predict'] obtained by a network's prediction based on the inverse transform. This function is only useful for spatial transforms.

class pymic.transform.rescale.**Rescale**(*params*)

Bases: *AbstractTransform*

Rescale the image to a given size.

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **Rescale_output_size** – (list/tuple or int) The output size along each spatial axis, such as [D, H, W] or [H, W]. If D is None, the input image is only reslcaled in 2D. If int, the smallest axis is matched to output_size keeping aspect ratio the same as the input.
- **Rescale_inverse** – (optional, bool) Is inverse transform needed for inference. Default is *True*.

**inverse_transform_for_prediction**(*sample*)

Inverse transform for the sample dictionary. Especially, it will update sample['predict'] obtained by a network's prediction based on the inverse transform. This function is only useful for spatial transforms.

### 3.6.10 pymic.transform.rotate module

class pymic.transform.rotate.**RandomRotate**(*params*)

Bases: *AbstractTransform*

Random rotate an image, wiht a shape of [C, D, H, W] or [C, H, W].

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **RandomRotate_angle_range_d** – (list/tuple or None) Rotation angle (degree) range along depth axis (x-y plane), e.g., (-90, 90). If None, no rotation along this axis.
- **RandomRotate_angle_range_h** – (list/tuple or None) Rotation angle (degree) range along height axis (x-z plane), e.g., (-90, 90). If None, no rotation along this axis. Only used for 3D images.
- **RandomRotate_angle_range_w** – (list/tuple or None) Rotation angle (degree) range along width axis (y-z plane), e.g., (-90, 90). If None, no rotation along this axis. Only used for 3D images.
- **RandomRotate_probability** – (optional, float) The probability of applying RandomRotate. Default is 0.5.
- **RandomRotate_inverse** – (optional, bool) Is inverse transform needed for inference. Default is *True*.

**inverse_transform_for_prediction**(*sample*)

Inverse transform for the sample dictionary. Especially, it will update sample['predict'] obtained by a network's prediction based on the inverse transform. This function is only useful for spatial transforms.

### 3.6.11 pymic.transform.threshold module

class pymic.transform.threshold.**ChannelWiseThreshold**(*params*)

Bases: *AbstractTransform*

Thresholding the image for given channels.

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **ChannelWiseThreshold_channels** – (list/tuple or None) A list of specified channels for thresholding. If None (by default), all the channels will be thresholded.
- **ChannelWiseThreshold_threshold_lower** – (list/tuple or None) The lower threshold for the given channels.

- **ChannelWiseThreshold_threshold_upper** – (list/tuple or None) The upper threshold for the given channels.
- **ChannelWiseThreshold_replace_lower** – (list/tuple or None) The output value for pixels with an input value lower than the threshold_lower.
- **ChannelWiseThreshold_replace_upper** – (list/tuple or None) The output value for pixels with an input value higher than the threshold_upper.
- **ChannelWiseThreshold_inverse** – (optional, bool) Is inverse transform needed for inference. Default is *False*.

class pymic.transform.threshold.**ChannelWiseThresholdWithNormalize**(*params*)

Bases: *AbstractTransform*

Apply thresholding and normalization for given channels. Pixel intensity will be truncated to the range of (lower, upper) and then normalized. If mean_std_mode is True, the mean and std values for pixel in the target range is calculated for normalization, and input intensity outside that range will be replaced by random values. Otherwise, the intensity will be normalized to [0, 1].

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

- **ChannelWiseThresholdWithNormalize_channels** – (list/tuple or None) A list of specified channels for thresholding. If None (by default), all the channels will be affected by this transform.
- **ChannelWiseThresholdWithNormalize_threshold_lower** – (list/tuple or None) The lower threshold for the given channels.
- **ChannelWiseThresholdWithNormalize_threshold_upper** – (list/tuple or None) The upper threshold for the given channels.
- **ChannelWiseThresholdWithNormalize_mean_std_mode** – (bool) If True, using mean and std for normalization. If False, using min and max values for normalization.
- **ChannelWiseThresholdWithNormalize_inverse** – (optional, bool) Is inverse transform needed for inference. Default is *False*.

### 3.6.12 pymic.transform.trans_dict module

The built-in transforms in PyMIC are:

```
'ChannelWiseThreshold': ChannelWiseThreshold,
'ChannelWiseThresholdWithNormalize': ChannelWiseThresholdWithNormalize,
'CropWithBoundingBox': CropWithBoundingBox,
'CenterCrop': CenterCrop,
'GrayscaleToRGB': GrayscaleToRGB,
'GammaCorrection': GammaCorrection,
'GaussianNoise': GaussianNoise,
'LabelConvert': LabelConvert,
'LabelConvertNonzero': LabelConvertNonzero,
'LabelToProbability': LabelToProbability,
'NormalizeWithMeanStd': NormalizeWithMeanStd,
'NormalizeWithMinMax': NormalizeWithMinMax,
'NormalizeWithPercentiles': NormalizeWithPercentiles,
'PartialLabelToProbability':PartialLabelToProbability,
```

(continues on next page)

(continued from previous page)

```
'RandomCrop': RandomCrop,
'RandomResizedCrop': RandomResizedCrop,
'RandomRescale': RandomRescale,
'RandomFlip': RandomFlip,
'RandomRotate': RandomRotate,
'ReduceLabelDim': ReduceLabelDim,
'Rescale': Rescale,
'Pad': Pad.
```

### 3.6.13 Module contents

## 3.7 pymic.util package

### 3.7.1 Submodules

### 3.7.2 pymic.util.evaluation_cls module

Evaluation module for classification tasks.

`pymic.util.evaluation_cls.`**`accuracy`**(*gt_label*, *pred_label*)

Calculate the accuracy.

`pymic.util.evaluation_cls.`**`binary_evaluation`**(*config*)

Evaluation of binary classification performance. The arguments are given in the *config* dictionary. It should have the following fields:

**Parameters**

- **`metric_list`** – (list) A list of evaluation metrics. The supported metrics are {*accuracy*, *recall*, *sensitivity*, *specificity*, *precision*, *auc*}.
- **`ground_truth_csv`** – (str) The csv file for ground truth.
- **`predict_prob_csv`** – (str) The csv file for prediction probability.

`pymic.util.evaluation_cls.`**`get_evaluation_score`**(*gt_label*, *pred_prob*, *metric*)

Get an evaluation score for binary classification.

**Parameters**

- **`gt_label`** – (array) Ground truth label.
- **`pred_prob`** – (array) Predicted positive probability.
- **`metric`** – (str) One of the evaluation metrics in {*accuracy*, *recall*, *sensitivity*, *specificity*, *precision*, *auc*}.

`pymic.util.evaluation_cls.`**`main`**()

Main function for evaluation of classification results. A configuration file is needed for runing. e.g.,

```
pymic_evaluate_cls config.cfg
```

The configuration file should have an *evaluation* section with the following fields:

**Parameters**

- **task_type** – (str) *cls* or *cls_nexcl*.
- **metric_list** – (list) A list of evaluation metrics. The supported metrics are {*accuracy*, *recall*, *sensitivity*, *specificity*, *precision*, *auc*}.
- **ground_truth_csv** – (str) The csv file for ground truth.
- **predict_prob_csv** – (str) The csv file for prediction probability.

pymic.util.evaluation_cls.**nexcl_evaluation**(*config*)

Evaluation of non-exclusive binary classification performance. The arguments are given in the *config* dictionary. It should have the following fields:

**Parameters**

- **metric_list** – (list) A list of evaluation metrics. The supported metrics are {*accuracy*, *recall*, *sensitivity*, *specificity*, *precision*, *auc*}.
- **ground_truth_csv** – (str) The csv file for ground truth.
- **predict_prob_csv** – (str) The csv file for prediction probability.

pymic.util.evaluation_cls.**sensitivity**(*gt_label*, *pred_label*)

Calculate the sensitivity for binary prediction.

pymic.util.evaluation_cls.**specificity**(*gt_label*, *pred_label*)

Calculate the specificity for binary prediction.

### 3.7.3 pymic.util.evaluation_seg module

Evaluation module for segmenation tasks.

pymic.util.evaluation_seg.**binary_assd**(*s*, *g*, *spacing=None*)

Get the Average Symetric Surface Distance (ASSD) between a binary segmentation and the ground truth.

**Parameters**

- **s** – (numpy.array) a 2D or 3D binary image for segmentation.
- **g** – (numpy.array) a 2D or 2D binary image for ground truth.
- **spacing** – (list) A list for image spacing, length should be 2 or 3.

**Returns**

The ASSD value.

pymic.util.evaluation_seg.**binary_dice**(*s*, *g*, *resize=False*)

Calculate the Dice score of two N-d volumes for binary segmentation.

**Parameters**

- **s** – The segmentation volume of numpy array.
- **g** – the ground truth volume of numpy array.
- **resize** – (optional, bool) If s and g have different shapes, resize s to match g. Default is *True*.

**Returns**

The Dice value.

`pymic.util.evaluation_seg.binary_hd95(s, g, spacing=None)`

Get the 95 percentile of hausdorff distance between a binary segmentation and the ground truth.

**Parameters**

- **s** – (numpy.array) a 2D or 3D binary image for segmentation.
- **g** – (numpy.array) a 2D or 2D binary image for ground truth.
- **spacing** – (list) A list for image spacing, length should be 2 or 3.

**Returns**
The HD95 value.

`pymic.util.evaluation_seg.binary_iou(s, g)`

Calculate the IoU score of two N-d volumes for binary segmentation.

**Parameters**

- **s** – The segmentation volume of numpy array.
- **g** – the ground truth volume of numpy array.

**Returns**
The IoU value.

`pymic.util.evaluation_seg.binary_relative_volume_error(s, g)`

Get the Relative Volume Error (RVE) between a binary segmentation and the ground truth.

**Parameters**

- **s** – (numpy.array) a 2D or 3D binary image for segmentation.
- **g** – (numpy.array) a 2D or 2D binary image for ground truth.

**Returns**
The RVE value.

`pymic.util.evaluation_seg.dice_of_images(s_name, g_name)`

Calculate the Dice score given the image names of binary segmentation and ground truth, respectively.

**Parameters**

- **s_name** – (str) The filename of segmentation result.
- **g_name** – (str) The filename of ground truth.

**Returns**
The Dice value.

`pymic.util.evaluation_seg.evaluation(config)`

Run evaluation of segmentation results based on a configuration dictionary *config*. The following fields should be provided in *config*:

**Parameters**

- **metric_list** – (list) The list of metrics for evaluation. The metric options are {*dice*, *iou*, *assd*, *hd95*, *rve*, *volume*}.
- **label_list** – (list) The list of labels for evaluation.
- **label_fuse** – (option, bool) If true, fuse the labels in the *label_list* as the foreground, and other labels as the background. Default is False.
- **organ_name** – (str) The name of the organ for segmentation.

- **ground_truth_folder_root** – (str) The root dir of ground truth images.
- **segmentation_folder_root** – (str or list) The root dir of segmentation images. When a list is given, each list element should be the root dir of the results of one method.
- **evaluation_image_pair** – (str) The csv file that provide the segmentation images and the corresponding ground truth images.
- **ground_truth_label_convert_source** – (optional, list) The list of source labels for label conversion in the ground truth.
- **ground_truth_label_convert_target** – (optional, list) The list of target labels for label conversion in the ground truth.
- **segmentation_label_convert_source** – (optional, list) The list of source labels for label conversion in the segmentation.
- **segmentation_label_convert_target** – (optional, list) The list of target labels for label conversion in the segmentation.

pymic.util.evaluation_seg.**get_binary_evaluation_score**(*s_volume*, *g_volume*, *spacing*, *metric*)

Evaluate the performance of binary segmentation using a specified metric. The metric options are {*dice*, *iou*, *assd*, *hd95*, *rve*, *volume*}.

**Parameters**

- **s_volume** – (numpy.array) a 2D or 3D binary image for segmentation.
- **g_volume** – (numpy.array) a 2D or 2D binary image for ground truth.
- **spacing** – (list) A list for image spacing, length should be 2 or 3.
- **metric** – (str) The metric name.

**Returns**

The metric value.

pymic.util.evaluation_seg.**get_edge_points**(*img*)

Get edge points of a binary segmentation result.

**Parameters**

**img** – (numpy.array) a 2D or 3D array of binary segmentation.

**Returns**

an edge map.

pymic.util.evaluation_seg.**get_multi_class_evaluation_score**(*s_volume*, *g_volume*, *label_list*, *fuse_label*, *spacing*, *metric*)

Evaluate the segmentation performance using a specified metric for a list of labels. The metric options are {*dice*, *iou*, *assd*, *hd95*, *rve*, *volume*}. If *fuse_label* is *True*, the labels in *label_list* will be merged as foreground and other labels will be merged as background as a binary segmentation result.

**Parameters**

- **s_volume** – (numpy.array) A 2D or 3D image for segmentation.
- **g_volume** – (numpy.array) A 2D or 2D image for ground truth.
- **label_list** – (list) A list of target labels.
- **fuse_label** – (bool) Fuse the labels in *label_list* or not.
- **spacing** – (list) A list for image spacing, length should be 2 or 3.
- **metric** – (str) The metric name.

**Returns**
The metric value list.

pymic.util.evaluation_seg.**main**()

Main function for evaluation of segmentation results. A configuration file is needed for runing. e.g.,

```
pymic_evaluate_cls config.cfg
```

The configuration file should have an *evaluation* section. See *pymic.util.evaluation_seg.evaluation* for details of the configuration required.

### 3.7.4 pymic.util.general module

pymic.util.general.**get_one_hot_seg**(*label*, *class_num*)

Convert a segmentation label to one-hot.

**Parameters**

- **label** – A tensor with a shape of [N, 1, D, H, W] or [N, 1, H, W]
- **class_num** – Class number.

**Returns**
a one-hot tensor with a shape of [N, C, D, H, W] or [N, C, H, W].

pymic.util.general.**keyword_match**(*a*, *b*)

Test if two string are the same when converted to lower case.

pymic.util.general.**mixup**(*inputs*, *labels*)

Shuffle a minibatch and do linear interpolation between images and labels. Both classification and segmentation labels are supported. The targets should be one-hot labels.

**Parameters**

- **inputs** – a tensor of input images with size N X C0 x H x W.
- **labels** – a tensor of one-hot labels. The shape is N X C for classification tasks, and N X C X H X W for segmentation tasks.

pymic.util.general.**tensor_shape_match**(*a*, *b*)

Test if two tensors have the same shape

### 3.7.5 pymic.util.image_process module

pymic.util.image_process.**convert_label**(*label*, *source_list*, *target_list*)

Convert a label map based a source list and a target list of labels

**Parameters**

- **label** – (numpy.array) The input label map.
- **source_list** – A list of labels that will be converted, e.g. [0, 1, 2, 4]
- **target_list** – A list of target labels, e.g. [0, 1, 2, 3]

`pymic.util.image_process.crop_ND_volume_with_bounding_box`(*volume*, *bb_min*, *bb_max*)

Extract a subregion form an ND image.

**Parameters**

- **volume** – The input ND array.
- **bb_min** – (list) The lower bound of the bounding box for each axis.
- **bb_max** – (list) The upper bound of the bounding box for each axis.

**Returns**
A croped ND image.

`pymic.util.image_process.crop_and_pad_ND_array_to_desired_shape`(*image*, *out_shape*, *pad_mod*)

Crop and pad an image to a given shape.

**Parameters**

- **image** – The input ND array.
- **out_shape** – (list) The desired output shape.
- **pad_mod** – (str) See numpy.pad

`pymic.util.image_process.get_ND_bounding_box`(*volume*, *margin=None*)

Get the bounding box of nonzero region in an ND volume.

**Parameters**

- **volume** – An ND numpy array.
- **margin** – (list) The margin of bounding box along each axis.

**Return bb_min**
(list) A list for the minimal value of each axis of the bounding box.

**Return bb_max**
(list) A list for the maximal value of each axis of the bounding box.

`pymic.util.image_process.get_euclidean_distance`(*image*, *dim=3*, *spacing=[1.0, 1.0, 1.0]*)

Get euclidean distance transform of 3D binary images. The output distance map is unsigned.

**Parameters**

- **image** – The input 3D array.
- **dim** – (int) Using 2D (dim = 2) or 3D (dim = 3) distance transforms.
- **spacing** – (list) The spacing along each axis.

`pymic.util.image_process.get_largest_k_components`(*image*, *k=1*)

Get the largest K components from 2D or 3D binary image.

**Parameters**

- **image** – The input ND array for binary segmentation.
- **k** – (int) The value of k.

**Returns**
An output array with only the largest K components of the input.

pymic.util.image_process.**resample_sitk_image_to_given_spacing**(*image*, *spacing*, *order*)

Resample an sitk image objct to a given spacing.

**Parameters**

- **image** – The input sitk image object.
- **spacing** – (list/tuple) Target spacing along x, y, z direction.
- **order** – (int) Order for interpolation.

**Returns**

A resampled sitk image object.

pymic.util.image_process.**set_ND_volume_roi_with_bounding_box_range**(*volume*, *bb_min*, *bb_max*, *sub_volume*, *addition=True*)

Set the subregion of an ND image. If *addition* is *True*, the original volume is added by the given sub volume.

**Parameters**

- **volume** – The input ND volume.
- **bb_min** – (list) The lower bound of the bounding box for each axis.
- **bb_max** – (list) The upper bound of the bounding box for each axis.
- **sub_volume** – The sub volume to replace the target region of the orginal volume.
- **addition** – (optional, bool) If True, the sub volume will be added to the target region of the input volume.

### 3.7.6 pymic.util.parse_config module

pymic.util.parse_config.**is_bool**(*var_str*)

pymic.util.parse_config.**is_float**(*val_str*)

pymic.util.parse_config.**is_int**(*val_str*)

pymic.util.parse_config.**is_list**(*val_str*)

pymic.util.parse_config.**logging_config**(*config*)

pymic.util.parse_config.**parse_bool**(*var_str*)

pymic.util.parse_config.**parse_config**(*filename*)

pymic.util.parse_config.**parse_list**(*val_str*)

pymic.util.parse_config.**parse_value_from_string**(*val_str*)

pymic.util.parse_config.**synchronize_config**(*config*)

### 3.7.7 pymic.util.post_process module

class pymic.util.post_process.**PostKeepLargestComponent**(*params*)

Bases: *PostProcess*

Post process by keeping the largest component.

The arguments should be written in the *params* dictionary, and it has the following fields:

**Parameters**

**KeepLargestComponent_mode** – (int) *1* means keep the largest component of the union of foreground classes. *2* means keep the largest component for each foreground class.

class pymic.util.post_process.**PostProcess**(*params*)

Bases: object

The abastract class for post processing.

### 3.7.8 pymic.util.preprocess module

pymic.util.preprocess.**get_transform_list**(*trans_config_file*)

Create a list of transforms given a configuration file.

pymic.util.preprocess.**preprocess_with_transform**(*transforms*, *img_in_name*, *img_out_name*, *lab_in_name=None*, *lab_out_name=None*)

Using a list of data transforms for preprocessing, such as image normalization, cropping, etc. TODO: support multip-modality preprocessing.

**Parameters**

- **transforms** – (list) A list of transform objects.
- **img_in_name** – (str) Input file name.
- **img_out_name** – (str) Output file name.
- **lab_in_name** – (optional, str) If None, load the image's corresponding label for preprocessing as well.
- **lab_out_name** – (optional, str) The output label name.

### 3.7.9 pymic.util.ramps module

Functions for ramping hyperparameters up or down.

Each function takes the current training step or epoch, and the ramp length (start and end step or epoch), and returns a multiplier between 0 and 1.

pymic.util.ramps.**get_rampdown_ratio**(*i*, *start*, *end*, *mode='linear'*)

Obtain the rampdown ratio.

**Parameters**

- **i** – (int) The current iteration.
- **start** – (int) The start iteration.
- **end** – (int) The end itertation.
- **mode** – (str) Valid values are {*linear*, *sigmoid*, *cosine*}.

`pymic.util.ramps.get_rampup_ratio`(*i*, *start*, *end*, *mode='linear'*)

Obtain the rampup ratio.

**Parameters**

- **i** – (int) The current iteration.
- **start** – (int) The start iteration.
- **end** – (int) The end itertation.
- **mode** – (str) Valid values are {*linear*, *sigmoid*, *cosine*}.

### 3.7.10 Module contents

CHAPTER

FOUR

# CITATION

If you use PyMIC for your research, please acknowledge it accordingly by citing our paper:

G. Wang, X. Luo, R. Gu, S. Yang, Y. Qu, S. Zhai, Q. Zhao, K. Li, S. Zhang. PyMIC: A deep learning toolkit for annotation-efficient medical image segmentation. Computer Methods and Programs in Biomedicine (CMPB). 231 (2023): 107398.

BibTeX entry:

```
@article{Wang2022pymic,
author = {Guotai Wang and Xiangde Luo and Ran Gu and Shuojue Yang and Yijie Qu and␣
↪Shuwei Zhai and Qianfei Zhao and Kang Li and Shaoting Zhang},
title = {{PyMIC: A deep learning toolkit for annotation-efficient medical image␣
↪segmentation}},
year = {2022},
url = {https://doi.org/10.1016/j.cmpb.2023.107398},
journal = {Computer Methods and Programs in Biomedicine},
volume = {231},
pages = {107398},
}
```

# PYTHON MODULE INDEX

## p

# INDEX

## A



## B



## C

# G



# H

## I



## K



## L



## M

## N



## O



## P

# R

# S



# T

## U



## V



## W