```python
SparrowLink/caculate_metric.py
from monai.transforms import (
    LoadImage,
    SaveImage,
)
from tqdm import tqdm
from monai.data.meta_tensor import MetaTensor
import monai
import numpy as np
from openpyxl import Workbook
from metric_zoo import clDice
import seg_metrics.seg_metrics as sg
import argparse
import pathlib

dice_metric = monai.metrics.DiceMetric(include_background=True, reduction="mean")  # only label channel
hausdorff_metric = monai.metrics.HausdorffDistanceMetric(include_background=True, reduction="mean")
asd_metric = monai.metrics.SurfaceDistanceMetric(include_background=True, symmetric=True)
iou_metric = monai.metrics.MeanIoU(include_background=True, reduction="mean")


def update(pbar, record, result):
    pbar.update()
    record.append(result)


def error_back(err):
    print(err)


def caculate_metric(label_root, seg_root):
    label = LoadImage(image_only=False)(str(label_root))[0]
    seg = LoadImage(image_only=False)(str(seg_root))[0]
    if label.sum() == 0:
        result = {"name": [label_root.name], "metric": [1000, 1000, 1000,
                                                        1000, 1000]}
        return result
    if seg.sum() == 0:
        result = {"name": [label_root.name], "metric": [0, 0, 0,
                                                        1000, 1000]}
        return result
    dice = dice_metric(y_pred=seg[None, None], y=label[None, None])
    cldice = clDice(seg, label)
    iou = iou_metric(y_pred=seg[None, None], y=label[None, None])
    spacing = label.meta["pixdim"][1:4].tolist()
    distance_metrics = sg.write_metrics(labels=[1],  # exclude background if needed
                                        gdth_img=np.array(label),
                                        pred_img=np.array(seg),
                                        csv_file=None,
                                        spacing=spacing,
                                        metrics=['msd', 'hd95'],
                                        verbose=False)
    result = {"name": [label_root.name], "metric":[dice.item(), cldice.item(), iou.item(),
                                                   distance_metrics[0]['msd'][0],distance_metrics[0]['hd95'][0]]}
    return result


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--seg path', type=str, default=None)
    parser.add_argument('--label_path', type=str, default=None)
    parser.add_argument('--metric_result_path', type=str, default=None)
    parser.add_argument('--label_find', type=str, default="*.nii.gz")
    parser.add_argument('--seg_find', type=str, default="*.nii.gz")
    parser.add_argument("--multiprocess", action='store_true', default=False)
    args = parser.parse_args()
    assert args.seg_path is not None, "seg path is None"
    assert args.label_path is not None, "label_path is None"
    if args.metric_result_path is None:
        metric_result_path = str(pathlib.Path(args.seg_path) / "metric_result.xlsx")
    else:
        metric_result_path = args.metric_result_path
        pathlib.Path(metric_result_path).parent.mkdir(parents=True, exist_ok=True)
    # caculate the dice metric of the segmentation result

    label_list = list(pathlib.Path(args.label_path).glob(args.label_find))
    seg_list = list(pathlib.Path(args.seg_path).glob(args.seg_find))
    # seg_list = [name for name in seg_list if 'auxiliary' not in name]
    label_list.sort()
    seg_list.sort()
    # save the image_name, dice metric in a excel file
    # new a excel file
    wb = Workbook()

    ws = wb.active
    ws.append(['image_name', 'dice_metric', "cldice", "mIoU", "MSD", "HD95"])
    metric_list = []

    print(f"\033[96m calculating metric for {args.seg_find} \033[00m")
    assert len(seg_list) > 0, f"no file found in {args.seg_path} {args.seg_find}"
    pbar = tqdm(total=len(label_list), colour='#87cefa')
    if not args.multiprocess:
        for label_root, seg_root in zip(label_list, seg_list):
            # print(label_root.split("/")[-1], seg_root.split("/")[-1])
            result_dict = caculate_metric(label_root, seg_root)
            ws.append(result_dict["name"] + result_dict["metric"])
            metric_list.append(result_dict["metric"])
            pbar.set_description(f'metric:{str(seg_root.name), result_dict["metric"][0]}')
            pbar.update()
    else:
        from multiprocessing import Pool
        pool = Pool(14)
        result_record = []
        for label_root, seg_root in zip(label_list, seg_list):
            pool.apply_async(func=caculate_metric,
                             args=(label_root, seg_root),
                             error_callback=error_back,
                             callback=lambda x: update(pbar, result_record, x))
        pool.close()
        pool.join()
        # sort the result by "name"
```

```
        # sort the result by "name"
        result_record.sort(key=lambda x: x.get("name"))
        for result in result_record:
            ws.append(result.get("name") + result.get("metric"))
            metric_list.append(result.get("metric"))

    metric_array = np.array(metric_list)
    mean_metric = metric_array.mean(axis=0).tolist()
    mean_metric.insert(0, 'mean')
    ws.append(mean_metric)
    wb.save(metric_result_path)
```

H:\git\Repo2PDF\dist\repo2pdfAPP\SparrowLink\data\loader.py

```
SparrowLink/data/loader.py
import json
import os
import glob
import random
import numpy as np
import pathlib


def five_fold_generator(L):
    """
    :param L: a list
    :return: five_fold: shape 5*1
    """
    length = len(L)
    index = list(range(length))
    random.shuffle(index)
    d = length // 5
    five_fold = []
    print(f"rand_index:{index}")
    for i in range(5):
        train_index = index[:(4 - i) * d] + index[(5 - i) * d:] if i > 0 else index[:(4 - i) * d]
        val_index = index[(4 - i) * d:(5 - i) * d] if i > 0 else index[(4 - i) * d:]
        five_fold.append({"train_files": [L[k] for k in train_index],
                          "val_files": [L[k] for k in val_index]})
    return five_fold
        # print(f"{i}: train: {five_fold_index[i]['train']}, val: {five_fold_index[i]['val']}")


def shuffle_generator(L):
    """
    :param L: a list
    :return: shuffle_fold: 20% randomly selected sample form origin dataset
    """
    length = len(L)
    index = list(range(length))
    random.shuffle(index)
    train_index, val_index = index[:int(0.8*length)], index[int(0.8*length):]
    shuffle_fold = [{"train files": [L[k] for k in train_index],
                     "val_files": [L[k] for k in val_index]}]

    return shuffle_fold


def generator_multi_dataset(L1, L2, split_mode='shuffle'):
    """
    :spilt_mode:
    :param L1, L2: a list, diast, syst
    :return: shuffle_fold: 20% randomly selected sample form origin dataset
    """
    length = len(L1)
    assert len(L1) == len(L2), 'length of two list are different '
    train_index, val_index = index_generator(length, split_mode=split_mode)
    fold = []
    for i in range(len(train_index)):
        fold_l1 = {"train_files": [L1[k] for k in train_index[i]],
                   "val files": [L1[k] for k in val_index[i]]}
        fold_l2 = {"train_files": [L2[k] for k in train_index[i]],
                   "val files": [L2[k] for k in val index[i]]}
        fold.append({"train_files": fold_l1["train_files"] + fold_l2["train_files"],
                     "val_files": fold_l1["val_files"] + fold_l2["val_files"], })
    return fold


def prepare_datalist(image_file="images", label_file="label", split_mode='order'):
    """
    :param image_file: the name of image file
    :param label_file: the name of label file
    :return: train_files, val_files. Now we separate them directly. It needs to be modified for 5-fold validation.
    """

    image_path = pathlib.Path(image_file)
    label_path = pathlib.Path(label_file)
    assert image_path.is_dir(), f"img path not exist: {image_path}"
    assert label_path.is_dir(), f"label path not exist: {label_path}"
    # get the image name list in image_path using pathlib
    train_images = [path.name for path in image_path.glob("*.nii.gz")]
    data_dicts = [
        {"image": str(pathlib.Path(image_path, image_name)), "label": str(pathlib.Path(label_path, image_name))}
        for image_name in train_images
    ]
    if split_mode == 'five_fold':
        return five_fold_generator(data_dicts)
    elif split_mode == 'order':
        length = len(data_dicts)
        index = int(0.2 * length)
        train_files, val_files = data_dicts[:-index], data_dicts[-index:]
        return [{"train_files": train_files, "val_files": val_files}]
    elif split_mode == 'shuffle':
        length = len(data_dicts)
        index = int(0.2 * length)
        train_files, val_files = data_dicts[:-index], data_dicts[-index:]
        return [{"train_files": train_files, "val_files": val_files}]
    elif split_mode == 'all':
        return data_dicts
    else:
        raise RuntimeError(f"{split_mode} is not supported")

```

```python

def prepare_datalist_with_file(image_file="images", label_file="label",img_name="", ):
    """
    :param image_file: the name of image file
    :param label_file: the name of label file
    :return: train_files, val_files. Now we separate them directly. It needs to be modified for 5-fold validation.
    """
    # use json load image name list form img_name, which is a json file
    with open(img_name, 'r') as f:
        name_list = json.load(f)
    image_path = pathlib.Path(image_file)
    label_path = pathlib.Path(label_file)
    assert os.path.isdir(image_path), f"img path not exist: {image_path}"
    assert os.path.isdir(label_path), f"label path not exist: {label_path}"
    data_dicts = [
        {"image": str(pathlib.Path(image_path, name)), "label": str(pathlib.Path(label_path, name))}
        for name in name_list
    ]
    return data_dicts


def prepare_multi_datalist_with_file(main_file, auxiliary_file, label_file, broken_file, broken_gt_file, img_name=None, select_file=None):
    """
    :param main_file:  the root of data file.
    :param auxiliary_file: the name of image file
    :param label_file: the name of label file
    :param broken_file: the name of broken image file
    :param broken_gt_file: the name of broken gt file
    :param select_file: the name of selected file, contains the name of selected image and the number broken part
    :return: train_files, val_files. Now we separate them directly. It needs to be modified for 5-fold validation.
    """
    # use json load image name list form img_name, which is a json file
    if img_name is not None:
        with open(img_name, 'r') as f:
            name_list = json.load(f)
    else:
        name_list = pathlib.Path(main_file).glob("*.nii.gz")
        name_list = [path.name for path in name_list]
        # sort the name list
        name_list.sort()
    if select_file is not None:
        with open(select_file, 'r') as f:
            select_list = json.load(f)
        select_list = [path["name"] for path in select_list if path["num"] > 0]
        name_list = [name for name in name_list if name.replace(".nii.gz", "") in select_list]

    print(f"num:{len(name_list)}")
    main_path = pathlib.Path(main_file)
    auxiliary_path = pathlib.Path(auxiliary_file)

    label_path = pathlib.Path(label_file)
    assert os.path.isdir(main_path), f"img path not exist: {main_path}"
    assert os.path.isdir(label_path), f"label path not exist: {label_path}"
    assert os.path.isdir(auxiliary_path), f"img path not exist: {auxiliary_path}"
    assert os.path.isdir(broken_file), f"img path not exist: {broken_file}"
    assert os.path.isdir(broken_gt_file), f"img path not exist: {broken_gt_file}"
    # because we do not have full data
    data_dicts = [
        {"main": str(pathlib.Path(main_path, name)),
         "auxiliary": str(pathlib.Path(auxiliary_path, name)),
         "label": str(pathlib.Path(label_path, name)),
         "broken": str(pathlib.Path(broken_file, name)),
         "broken_gt": str(pathlib.Path(broken_gt_file, name)),}
        for name in name_list if pathlib.Path(main_path, name).exists() and pathlib.Path(auxiliary_path, name).exists()
    ]
    return data_dicts


def prepare_main_auxiliary_with_img_datalist_with_file(main_file,
                                                       auxiliary_file,
                                                       main_img_file,
                                                       auxiliary_img_file,
                                                       broken_file,
                                                       broken_gt_file,
                                                       label_file=None,
                                                       img_name="",
                                                       select_file=None):
    """
    :param main_file: the path of coarse segmentation in main phase.
    :param auxiliary_file: the path of refined segmentation in auxiliary phase.
    :param main_img_file: the path of image in main phase.
    :param auxiliary_img_file: the path of image in auxiliary phase.
    :param broken_file: the path of broken spheres in main phase.
    :param broken_gt_file: the path of broken spheres generated with gt in main phase.
    :param label_file: the path of label file.
    :param img_name: the name of image file.
    :param select_file: record the number of discontinuity sphere.
    """
    # use json load image name list form img_name, which is a json file
    if img_name is not None:
        with open(img_name, 'r') as f:
            name_list = json.load(f)
    else:
        name_list = pathlib.Path(main_file).glob("*.nii.gz")
        name_list = [path.name for path in name_list]
        # sort the name list
        name_list.sort()

    if select_file is not None:
        with open(select_file, 'r') as f:
            select_list = json.load(f)
        select_list = [path["name"] for path in select_list if path["num"] > 0]
        name_list = [name for name in name_list if name.replace(".nii.gz", "") in select_list]

    main_path = pathlib.Path(main_file)
    auxiliary_path = pathlib.Path(auxiliary_file)
    main_image_path = pathlib.Path(main_img_file)
    auxiliary_image_path = pathlib.Path(auxiliary_img_file)
    assert os.path.isdir(main_path), f"main path not exist: {main_path}"
    assert os.path.isdir(main_image_path), f"main_image_path not exist: {main_image_path}"
    assert os.path.isdir(auxiliary_image_path), f"auxiliary image path not exist: {auxiliary_image_path}"
```

```
    assert os.path.isdir(auxiliary_path), f"auxiliary_path not exist: {auxiliary_path}"
    assert os.path.isdir(broken_file), f"broken_file not exist: {broken_file}"
    assert os.path.isdir(broken_gt_file), f"broken_gt_file not exist: {broken_gt_file}"
    if label_file is None:
        data_dicts = [
            {"CS_M": str(pathlib.Path(main_path, name)),
             "CS_A": str(pathlib.Path(auxiliary_path, name)),
             "I_M": str(pathlib.Path(main_image_path, name)),
             "I_A": str(pathlib.Path(auxiliary_image_path, name)),
             "CS_DL": str(pathlib.Path(broken_file, name)),
             "CS_DLGT": str(pathlib.Path(broken_gt_file, name)), }
            for name in name_list
        ]
        return data_dicts
    else:
        label_path = pathlib.Path(label_file)
        assert os.path.isdir(label_path), f"label path not exist: {label_path}"
        data_dicts = [
            {"CS_M": str(pathlib.Path(main_path, name)),
             "CS_A": str(pathlib.Path(auxiliary_path, name)),
             "I_M": str(pathlib.Path(main_image_path, name)),
             "I_A": str(pathlib.Path(auxiliary_image_path, name)),
             "label": str(pathlib.Path(label_path, name)),
             "CS_DL": str(pathlib.Path(broken_file, name)),
             "CS_DLGT": str(pathlib.Path(broken_gt_file, name)), }
            for name in name_list
        ]
        return data_dicts


def prepare_main_with_img_datalist_with_file(main_file,
                                             main_img_file,
                                             broken_file,
                                             broken_gt_file,
                                             label_file=None,
                                             img_name="",
                                             select_file=None):
    """
    :param main_file: the path of coarse segmentation in main phase.
    :param auxiliary_file: the path of refined segmentation in auxiliary phase.
    :param main_img_file: the path of image in main phase.
    :param auxiliary_img_file: the path of image in auxiliary phase.
    :param broken_file: the path of broken spheres in main phase.
    :param broken_gt_file: the path of broken spheres generated with gt in main phase.
    :param label_file: the path of label file.
    :param img_name: the name of image file.
    :param select_file: record the number of discontinuity sphere.
    """
    # use json load image name list form img_name, which is a json file
    if img_name is not None:
        with open(img_name, 'r') as f:
            name_list = json.load(f)
    else:
        name_list = pathlib.Path(main_file).glob("*.nii.gz")
        name_list = [path.name for path in name_list]
        # sort the name list
        name_list.sort()

    if select_file is not None:
        with open(select_file, 'r') as f:
            select_list = json.load(f)
        select_list = [path["name"] for path in select_list if path["num"] > 0]
        name_list = [name for name in name_list if name.replace(".nii.gz", "") in select_list]

    main_path = pathlib.Path(main_file)
    main_image_path = pathlib.Path(main img file)
    assert os.path.isdir(main_path), f"main_path not exist: {main_path}"
    assert os.path.isdir(main_image_path), f"main image path not exist: {main_image_path}"
    assert os.path.isdir(broken_file), f"broken_file not exist: {broken_file}"
    assert os.path.isdir(broken_gt_file), f"broken_gt_file not exist: {broken_gt_file}"
    if label_file is None:
        data_dicts = [
            {"CS_M": str(pathlib.Path(main_path, name)),
             "I_M": str(pathlib.Path(main_image_path, name)),
             "CS_DL": str(pathlib.Path(broken_file, name)),
             "CS_DLGT": str(pathlib.Path(broken_gt_file, name)), }
            for name in name_list
        ]
        return data_dicts
    else:
        label_path = pathlib.Path(label_file)
        assert os.path.isdir(label_path), f"label path not exist: {label_path}"
        data_dicts = [
            {"CS_M": str(pathlib.Path(main_path, name)),
             "I_M": str(pathlib.Path(main_image_path, name)),
             "label": str(pathlib.Path(label_path, name)),
             "CS_DL": str(pathlib.Path(broken_file, name)),
             "CS_DLGT": str(pathlib.Path(broken_gt_file, name)), }
            for name in name_list
        ]
        return data_dicts



def index_generator(length, split_mode='shuffle'):
    """
    :length: list length
    """
    train_index = []
    val_index = []
    index = list(range(length))
    if split_mode == 'shuffle':
        random.shuffle(index)
        train_index.append(index[:int(0.8 * length)])

        val_index.append(index[int(0.8 * length):])

    elif split_mode == 'five_fold':
        # random.shuffle(index)
        d = length // 5
        for i in range(5):
```

```
            train_index.append(index[:(4 - i) * d] + index[(5 - i) * d:] if i > 0 else index[:(4 - i) * d])
            val_index.append(index[(4 - i) * d:(5 - i) * d] if i > 0 else index[(4 - i) * d:])

    elif split_mode == 'order':
        train_index.append(index[:int(0.8 * length)])
        val_index.append(index[int(0.8 * length):])
    else:
        raise RuntimeError(f"{split_mode} is not supported")
    return train_index, val_index


def prepare_datalist_with_heart_label(data_dir, image_file="images", label_file="label", heart_file="heart"):
    """
    :param data_dir:  the root of data file.
    :param image_file: the name of image file
    :param label_file: the name of label file
    :return: train_files, val_files. Now we separate them directly. It needs to be modified for 5-fold validation.
    """

    image_path = os.path.join(data_dir, image_file)
    label_path = os.path.join(data_dir, label_file)
    heart_path = os.path.join(data_dir, heart_file)
    assert os.path.isdir(image_path), "img path not exist"
    assert os.path.isdir(label_path), "label path not exist"
    assert os.path.isdir(heart_path), "label path not exist"

    train_images = sorted(glob.glob(os.path.join(image_path, "*.nii.gz")))
    train_labels = sorted(glob.glob(os.path.join(label_path, "*.nii.gz")))
    train_heart = sorted(glob.glob(os.path.join(heart_path, "*.nii.gz")))

    data_dicts = [
        {"image": image_name, "label": label_name, "heart": heart_seg}
        for image_name, label_name, heart_seg in zip(train_images, train_labels, train_heart)
    ]
    return data_dicts


def write_data_reference(L, save_path):
    "a list contain val_files and train files"
    with open(os.path.join(save_path, 'train_set.txt'), 'w') as f:
        for dic in L['train_files']:
            for key, file in dic.items():
                f.write(os.path.split(file)[-1] + '\n')
                break

    with open(os.path.join(save path, 'val_set.txt'), 'w') as f:
        for dic in L['val_files']:
            for key, file in dic.items():
                f.write(os.path.split(file)[-1] + '\n')
                break


def save_json(L, save_path):
    with open(save_path, 'w') as f:
        json.dump(L, f)


def load_json(path):
    with open(path, 'r') as f:
        x = json.load(f)
    return x


def prepare_image_list(image_path):
    """
    generate image list in a dir
    """
    assert os.path.isdir(image_path), "img path not exist"
    train_images = sorted(glob.glob(os.path.join(image_path, "*.nii.gz")))
    data_dicts = [
        {"image": image_name}
        for image_name in train_images
    ]
    return data_dicts


if __name__ == '__main__':
    L = list(range(10))
    print(shuffle_generator(L))
```

```python
SparrowLink/loss_zoo/cldice.py
import torch
import torch.nn as nn
import torch.nn.functional as F
from .soft_skeleton import soft_skel
from monai.losses import DiceLoss
from monai.networks import one_hot


class soft_cldice(nn.Module):
    def __init__(self, iter_=3, smooth = 1.):
        super(soft_cldice, self).__init__()
        self.iter = iter_
        self.smooth = smooth

    def forward(self, y_true, y_pred):
        skel_pred = soft_skel(y_pred, self.iter)
        skel_true = soft_skel(y_true, self.iter)
        tprec = (torch.sum(torch.multiply(skel_pred, y_true)[:,1:,...])+self.smooth)/(torch.sum(skel_pred[:,1:,...])+smooth)
        tsens = (torch.sum(torch.multiply(skel_true, y_pred)[:,1:,...])+self.smooth)/(torch.sum(skel_true[:,1:,...])+smooth)
        cl_dice = 1.- 2.0*(tprec*tsens)/(tprec+tsens)
        return cl_dice


def soft_dice(y_true, y_pred):
    """[function to compute dice loss]
    Args:
        y_true ([float32]): [ground truth image]
        y_pred ([float32]): [predicted image]
    Returns:
        [float32]: [loss value]
    """
    smooth = 1
    intersection = torch.sum((y_true * y_pred)[:,1:,...])
    coeff = (2. * intersection + smooth) / (torch.sum(y_true[:,1:,...]) + torch.sum(y_pred[:,1:,...]) + smooth)
    return (1. - coeff)


class soft_dice_cldice(nn.Module):
    def __init__(self, iter_=3, alpha=0.5, smooth=1.):
        super(soft_dice_cldice, self).__init__()
        self.iter = iter_
        self.smooth = smooth
        self.alpha = alpha
        self.dice = DiceLoss(to_onehot_y=False, softmax=False)

    def forward(self, y_pred, y_true):
        y_true = one_hot(y_true, num_classes=2)
        y_pred = F.softmax(y_pred, dim=1)
        dice = self.dice(y_pred, y_true)
        skel_pred = soft_skel(y_pred, self.iter)
        skel_true = soft_skel(y_true, self.iter)
        tprec = (torch.sum(torch.multiply(skel_pred, y_true)[:,1:,...])+self.smooth)/(torch.sum(skel_pred[:,1:,...])+self.smooth)
        tsens = (torch.sum(torch.multiply(skel_true, y_pred)[:,1:,...])+self.smooth)/(torch.sum(skel_true[:,1:,...])+self.smooth)
        cl_dice = 1.- 2.0*(tprec*tsens)/(tprec+tsens)
        return (1.0-self.alpha)*dice+self.alpha*cl_dice


class soft_dice_cldice_weighted(nn.Module):
    def __init__(self, iter_=3, alpha=0.5, smooth = 1.):
        super(soft_dice_cldice_weighted, self).__init__()
        self.iter = iter_
        self.smooth = smooth
        self.alpha = alpha
        self.dice = DiceLoss(to_onehot_y=False, softmax=False, batch=True, reduction="none")

    def forward(self, y_pred, y_true, weight):
        # y_true = one_hot(y_true, num_classes=2)
        # y_pred = F.softmax(y_pred, dim=1)
        intersection = torch.sum((y_true * y_pred)[:, 1:, ...], dim=(1, 2, 3, 4), keepdim=True)
        coeff = (2. * intersection + self.smooth) / (torch.sum(y_true[:, 1:, ...], dim=(1, 2, 3, 4), keepdim=True) +
                                                     torch.sum(y_pred[:, 1:, ...], dim=(1, 2, 3, 4), keepdim=True) + self.smooth)
        dice = 1.0 - coeff
        skel_pred = soft_skel(y_pred, self.iter)
        skel_true = soft_skel(y_true, self.iter)
        tprec = (torch.sum(torch.multiply(skel_pred, y_true)[:,1:,...], dim=(1, 2, 3, 4), keepdim=True)+self.smooth)/(torch.sum(skel_pred[:,1:,...], dim=(1, 2, 3, 4), keepd
        tsens = (torch.sum(torch.multiply(skel_true, y_pred)[:,1:,...], dim=(1, 2, 3, 4), keepdim=True)+self.smooth)/(torch.sum(skel_true[:,1:,...], dim=(1, 2, 3, 4), keepd
        cl_dice = 1.- 2.0*(tprec*tsens)/(tprec+tsens)
        dice = torch.mean(dice * weight)
        cl_dice = torch.mean(cl_dice * weight)
        return (1.0-self.alpha)*dice+self.alpha*cl_dice
```

H:\git\Repo2PDF\dist\repo2pdfAPP\SparrowLink\loss_zoo\soft_skeleton.py

```python
SparrowLink/loss_zoo/soft_skeleton.py
import torch
import torch.nn as nn
import torch.nn.functional as F


def soft_erode(img):
    if len(img.shape)==4:
        p1 = -F.max_pool2d(-img, (3,1), (1,1), (1,0))
        p2 = -F.max_pool2d(-img, (1,3), (1,1), (0,1))
        return torch.min(p1,p2)
    elif len(img.shape)==5:
        p1 = -F.max_pool3d(-img,(3,1,1),(1,1,1),(1,0,0))
        p2 = -F.max_pool3d(-img,(1,3,1),(1,1,1),(0,1,0))
        p3 = -F.max_pool3d(-img,(1,1,3),(1,1,1),(0,0,1))
        return torch.min(torch.min(p1, p2), p3)


def soft_dilate(img):
    if len(img.shape) == 4:
        return F.max_pool2d(img, (3,3), (1,1), (1,1))
    elif len(img.shape) == 5:
        return F.max_pool3d(img,(3,3,3),(1,1,1),(1,1,1))


def soft_open(img):
    return soft_dilate(soft_erode(img))


def soft_skel(img, iter_):
    img1 = soft_open(img)
    skel = F.relu(img-img1)
    for j in range(iter_):
        img = soft_erode(img)
        img1 = soft_open(img)
        delta = F.relu(img-img1)
        skel = skel + F.relu(delta-skel*delta)
    return skel
```

```
SparrowLink/main.py
from monai.utils import first, set_determinism
from monai.data import CacheDataset, DataLoader, Dataset, decollate_batch, ITKReader, PersistentDataset
from monai.config import print_config
from monai.metrics.meandice import DiceMetric
import torch
import matplotlib.pyplot as plt
import os
import argparse
import monai
import logging
from utils.Config import Config
import sys
import time
from data.loader import prepare_datalist, prepare_datalist_with_file, prepare_image_list, save_json, write_data_reference, load_json
from transform.utils import get_transform
from utils.test import cardio_vessel_segmentation_test
from utils.trainer import cardio_vessel_segmentation_train
from utils.inferer import cardio_vessel_segmentation_infer
from pre_processing.checking import DatasetinformationExtractor
import pathlib
# print_config()


torch.multiprocessing.set_sharing_strategy('file_system')


if __name__ == "__main__":

    parser = argparse.ArgumentParser(description="Run a basic UNet segmentation baseline.")
    parser.add_argument("--configs",
                        dest="cfg",
                        help="The configs file.",
                        default='./configs/heart_server.yaml',
                        type=str)
    parser.add_argument('--iters', dest='iters', help='Iterations in training.', type=int, default=None)
    parser.add_argument('--batch_size', dest='batch_size', help='Mini batch size of one gpu or cpu.', type=int, default=None)
    parser.add_argument('--learning_rate', dest='learning_rate', help='Learning rate', type=float, default=None)
    parser.add_argument('--seed', dest='seed', help='seed', type=float, default=0)
    parser.add_argument('--mode', dest='mode', help='train, infer or both', type=str, default=None)

    parser.add_argument('--experiments_path', dest='experiments_path',
                        help='experiments_path, all output and checkpoint are saved here.', type=str, default=None)

    parser.add_argument('--pretrain_weight_path', help='pretrain weight for training, testing of inferring', type=str, default=None)
    parser.add_argument('--img_path', help='image path', type=str, default=None)
    parser.add_argument('--label_path', help='label path', type=str, default=None)
    parser.add_argument('--output_path', help='save path', type=str, default=None)
    parser.add_argument('--persist_path', help='persist path, the path used to save persist data in persist loader',
                        type=str, default=None)
    parser.add_argument('--val_set', help='val set, a txt path to select val data, only used if mode is train',
                        type=str, default=None)
    parser.add_argument('--train_set', help='train set, a txt path to select train data, only used if mode is train',
                        type=str, default=None)
    args = parser.parse_args()

    monai.config.print_config()
    logging.basicConfig(stream=sys.stdout, level=logging.INFO)
    cfg = Config(
        args.cfg,
        learning_rate=args.learning_rate,
        iters=args.iters,
        batch_size=args.batch_size,
        seed=args.seed,
        mode=args.mode,
        img_path=args.img_path,
        label_path=args.label_path,
        output_path=args.output_path,
        persist_path=args.persist_path,
        val_set=args.val_set,
        train_set=args.train_set,
        experiments_path=args.experiments_path,
        pretrain_weight_path=args.pretrain_weight_path,
        )
    set_determinism(seed=cfg.seed)
    if cfg.dic['mode'] == 'train':

        # ---------------------------- built transform sequence ---------------------------- #
        # ---------------------- Create Model, Loss, Optimizer in Config ------------------- #
        cfg.creat_training_require()
        train_transforms, val_transforms, save_transform = get_transform(cfg.dic['transform'])
        if cfg.dic["train"]["loader"].get("file_path"):
            files = [load_json(cfg.dic["train"]["loader"]["file_path"])]

        elif cfg.dic["train"]["loader"].get("val_set"):
            val_files = prepare_datalist_with_file(image_file=cfg.train_img_path,
                                                   label_file=cfg.train_label_path,
                                                   img_name=cfg.val_set, )
            train_files = prepare_datalist_with_file(image_file=cfg.train_img_path,
                                                     label_file=cfg.train_label_path,
                                                     img_name=cfg.train_set, )
            files = [{"train_files": train_files, "val_files": val_files}]
        else:
            files = prepare_datalist(image_file=cfg.train_img_path,
                                     label_file=cfg.train_label_path,
                                     split_mode=cfg.dic['train']['loader']['split_mode'], )
        # ---------------------------- create loss function ---------------------------------- #
        # ------------- you can crate your own loss or metric here amd replace cfg ------------ #

        # dice_metric = DiceMetric(include_background=False, reduction="mean")
        # loss_function = DiceLoss(to_onehot_y=True, softmax=True)

        # ---------------------------------- training ---------------------------------------- #
        metric_record = []
        experiment_path = os.path.join(cfg.dic['experiments_path'], time.strftime("%d_%m_%Y_%H_%M_%S"))\
            if cfg.dic.get('time_name', None) else cfg.dic['experiments_path']

        if not os.path.exists(experiment_path):
            os.makedirs(experiment_path)
        for i in range(len(files)):
            if cfg.dic['train']['loader'].get('split_mode') == "five_fold":
                experiment_path_fold = os.path.join(experiment_path, f"{i}_fold")
                if not os.path.exists(experiment_path_fold):
                    os.makedirs(experiment_path_fold)
            else:
                experiment_path_fold = experiment_path
            write_data_reference(files[i], experiment_path_fold)
            save_json(files[i], os.path.join(experiment_path_fold, 'files.txt'))
            # ---------------------------------- save config ------------------------------- #
```

```
            cfg.save_config(os.path.join(experiment_path_fold, 'configs.yaml'))
            if cfg.dic['train']['loader'].get('split_mode') == "five_fold":
                print(f"----------------fold{i} start!-----------------")
            else:
                print(f"----------------training start!-----------------")
            if cfg.dic['train']['loader'].get('persist'):
                print("---------------- using persist dataset -----------------")
                if cfg.persist_path == 'default':
                    persistent_cache = pathlib.Path(experiment_path_fold, "persistent_cache")
                else:
                    persistent_cache = pathlib.Path(cfg.persist_path)

                persistent_cache.mkdir(parents=True, exist_ok=True)
                train_ds = PersistentDataset(data=files[i]['train_files'], transform=train_transforms,
                                             cache_dir=persistent_cache)
                val_ds = PersistentDataset(data=files[i]['val_files'], transform=val_transforms,
                                           cache_dir=persistent_cache)
            else:
                print("---------------- using cache dataset -----------------")
                train_ds = CacheDataset(
                    data=files[i]['train_files'], transform=train_transforms,
                    cache_rate=cfg.dic['train']['loader']['cache'], num_workers=cfg.dic['train']['loader']['num_workers'])
                # train_ds = Dataset(data=train_files, transform=train_transforms)
                # use batch_size=2 to load images and use RandCropByPosNegLabeld
                val_ds = CacheDataset(
                    data=files[i]['val_files'], transform=val_transforms,
                    cache_rate=cfg.dic['train']['loader']['cache'], num_workers=cfg.dic['train']['loader']['num_workers'])
            out_channels = cfg.dic['model']['out_channels'] if cfg.dic['model'].get('out_channels',
                                                                                    None) else cfg.model.out_channels
            cardio_vessel_segmentation_train(cfg=cfg,
                                             model=cfg.model,
                                             num_class=out_channels,
                                             loss_function=cfg.train_loss,
                                             val_metric=cfg.val_metric,
                                             optimizer=cfg.optimizer_init,
                                             lr_scheduler=cfg.lr_scheduler_init,
                                             train_dataset=train_ds,
                                             val_dataset=val_ds,
                                             experiment_path=experiment_path_fold,
                                             device=cfg.device,
                                             metric_record=metric_record,
                                             start_epoch=cfg.start_epoch,
                                             mirror_axes=cfg.train_mirror_axes,
                                             sw_batch_size=cfg.train_sw_batch_size,
                                             overlap=cfg.train_sw_overlap,
                                             )

    elif cfg.dic['mode'] == 'test':
        # ---------------------------- built transform sequence ---------------------------- #
        train_transforms, val_transforms, save_transform = get_transform(cfg.dic['transform'])

        cfg.creat_test_require()
        cfg.save_config(os.path.join(cfg.test_output_path, 'config.yaml'))
        if cfg.dic["test"]["loader"].get("file_path"):
            files = load_json(cfg.dic["test"]["loader"]["file_path"])["val_files"]
        elif cfg.dic["test"]["loader"].get("val_set"):
            val_files = prepare_datalist_with_file(image_file=cfg.test_img_path,
                                                   label_file=cfg.test_label_path,
                                                   img_name=cfg.dic["test"]["loader"]["val_set"], )
            files = val_files
        else:
            files = prepare_datalist(image_file=cfg.train_img_path,
                                     label_file=cfg.test_label_path,
                                     split_mode=cfg.dic['test']['loader']['split_mode'], )

        total_ds = CacheDataset(
            data=files, transform=val_transforms, cache_rate=cfg.dic['test']['loader']['cache'], num_workers=2)

        cardio_vessel_segmentation_test(model=cfg.model,
                                        val_dataset=total_ds,
                                        device=cfg.device,
                                        output_path=cfg.test_output_path,
                                        window_size=cfg.dic['test']['test_windows_size'],
                                        save_data=cfg.dic['test']['save_data'])

    elif cfg.dic['mode'] == 'infer':
        # ---------------------------- built transform sequence ---------------------------- #
        infer_transforms = get_transform(cfg.dic['transform'], mode='infer')

        cfg.creat_infer_require()
        cfg.save_config(os.path.join(cfg.infer_output_path, 'config.yaml'))

        # files = prepare_image_list(image_path=cfg.dic['infer']['loader']['path'])

        image_path = pathlib.Path(cfg.infer_img_path)
        assert image_path.is_dir(), f"img path not exist: {image_path}"
        # get the image name list in image_path using pathlib
        train_images = [path.name for path in image_path.glob("*.nii.gz")]
        data_dicts = [
            {"image": str(pathlib.Path(image_path, image_name))}
            for image_name in train_images
        ]
        total_ds = CacheDataset(
            data=data_dicts, transform=infer_transforms, cache_rate=cfg.dic['infer']['loader']['cache'], num_workers=2)
        cardio_vessel_segmentation_infer(model=cfg.model,
                                         val_dataset=total_ds,
                                         device=cfg.device,
                                         output_path=cfg.infer_output_path,
                                         window_size=tuple(cfg.dic['transform']['patch_size']),
                                         overlap=cfg.infer_sw_overlap,
                                         origin_transforms=infer_transforms,
                                         mirror_axes=cfg.infer_mirror_axes,
                                         sw_batch_size=cfg.infer_sw_batch_size,
                                         mode="gaussian",
                                         )
    else:
        raise RuntimeError('Only train and infer mode are supported now')
```

H:\git\Repo2PDF\dist\repo2pdfAPP\SparrowLink\model\CS2net.py

```
SparrowLink/model/CS2net.py
"""
3D Channel and Spatial Attention Network (CSA-Net 3D).
"""
from __future__ import division
import torch
import torch.nn as nn
import torch.nn.functional as F


```

```python
def downsample():
    return nn.MaxPool3d(kernel_size=2, stride=2)


def deconv(in_channels, out_channels):
    return nn.ConvTranspose3d(in_channels, out_channels, kernel_size=2, stride=2)


def initialize_weights(*models):
    for model in models:
        for m in model.modules():
            if isinstance(m, nn.Conv3d) or isinstance(m, nn.Linear):
                nn.init.kaiming_normal(m.weight)
                if m.bias is not None:
                    m.bias.data.zero_()
            elif isinstance(m, nn.BatchNorm3d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()


class ResEncoder3d(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResEncoder3d, self).__init__()
        self.conv1 = nn.Conv3d(in_channels, out_channels, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm3d(out_channels)
        self.conv2 = nn.Conv3d(out_channels, out_channels, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm3d(out_channels)
        self.relu = nn.ReLU(inplace=False)
        self.conv1x1 = nn.Conv3d(in_channels, out_channels, kernel_size=1)

    def forward(self, x):
        residual = self.conv1x1(x)
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.relu(self.bn2(self.conv2(out)))
        out = residual + out
        out = self.relu(out)
        return out


class Decoder3d(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Decoder3d, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv3d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm3d(out_channels),
            nn.ReLU(inplace=False),
            nn.Conv3d(out_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm3d(out_channels),
            nn.ReLU(inplace=False)
        )

    def forward(self, x):
        out = self.conv(x)
        return out


class SpatialAttentionBlock3d(nn.Module):
    def __init__(self, in_channels):
        super(SpatialAttentionBlock3d, self).__init__()
        self.query = nn.Conv3d(in_channels, in_channels // 8, kernel_size=(1, 3, 1), padding=(0, 1, 0))
        self.key = nn.Conv3d(in_channels, in_channels // 8, kernel_size=(3, 1, 1), padding=(1, 0, 0))
        self.judge = nn.Conv3d(in_channels, in_channels // 8, kernel_size=(1, 1, 3), padding=(0, 0, 1))
        self.value = nn.Conv3d(in_channels, in_channels, kernel_size=1)
        self.gamma = nn.Parameter(torch.zeros(1))
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, x):
        """
        :param x: input( BxCxHxWxZ )
        :return: affinity value + x
        B: batch size
        C: channels
        H: height
        W: width
        D: slice number (depth)
        """
        B, C, H, W, D = x.size()
        # compress x: [B,C,H,W,Z]-->[B,H*W*Z,C], make a matrix transpose
        proj_query = self.query(x).view(B, -1, W * H * D).permute(0, 2, 1)  # -> [B,W*H*D,C]
        proj_key = self.key(x).view(B, -1, W * H * D)  # -> [B,H*W*D,C]
        proj_judge = self.judge(x).view(B, -1, W * H * D).permute(0, 2, 1)  # -> [B,C,H*W*D]

        affinity1 = torch.matmul(proj_query, proj_key)
        affinity2 = torch.matmul(proj_judge, proj_key)
        affinity = torch.matmul(affinity1, affinity2)
        affinity = self.softmax(affinity)

        proj_value = self.value(x).view(B, -1, H * W * D)  # -> C*N
        weights = torch.matmul(proj_value, affinity)
        weights = weights.view(B, C, H, W, D)
        out = self.gamma * weights + x
        return out


class ChannelAttentionBlock3d(nn.Module):
    def __init__(self, in_channels):
        super(ChannelAttentionBlock3d, self).__init__()
        self.gamma = nn.Parameter(torch.zeros(1))
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, x):
        """
        :param x: input( BxCxHxWxD )
        :return: affinity value + x
        """
        B, C, H, W, D = x.size()
        proj_query = x.view(B, C, -1).permute(0, 2, 1)
        proj_key = x.view(B, C, -1)
        proj_judge = x.view(B, C, -1).permute(0, 2, 1)
        affinity1 = torch.matmul(proj_key, proj_query)
        affinity2 = torch.matmul(proj_key, proj_judge)
        affinity = torch.matmul(affinity1, affinity2)
        affinity_new = torch.max(affinity, -1, keepdim=True)[0].expand_as(affinity) - affinity
        affinity_new = self.softmax(affinity_new)
        proj_value = x.view(B, C, -1)
        weights = torch.matmul(affinity_new, proj_value)
        weights = weights.view(B, C, H, W, D)
        out = self.gamma * weights + x
        return out
```

```python


class AffinityAttention3d(nn.Module):
    """ Affinity attention module """

    def __init__(self, in_channels):
        super(AffinityAttention3d, self).__init__()
        self.sab = SpatialAttentionBlock3d(in_channels)
        self.cab = ChannelAttentionBlock3d(in_channels)
        # self.conv1x1 = nn.Conv2d(in_channels * 2, in_channels, kernel_size=1)

    def forward(self, x):
        """
        sab: spatial attention block
        cab: channel attention block
        :param x: input tensor
        :return: sab + cab
        """
        sab = self.sab(x)
        cab = self.cab(x)
        out = sab + cab + x
        return out


class CSNet3D(nn.Module):
    def __init__(self, out_channels, in_channels):
        """
        :param classes: the object classes number.
        :param channels: the channels of the input image.
        """
        super(CSNet3D, self).__init__()
        self.out_channels = out_channels
        self.enc_input = ResEncoder3d(in_channels, 16)
        self.encoder1 = ResEncoder3d(16, 32)
        self.encoder2 = ResEncoder3d(32, 64)
        self.encoder3 = ResEncoder3d(64, 128)
        self.encoder4 = ResEncoder3d(128, 256)
        self.downsample = downsample()
        self.affinity_attention = AffinityAttention3d(256)
        self.attention_fuse = nn.Conv3d(256 * 2, 256, kernel_size=1)
        self.decoder4 = Decoder3d(256, 128)
        self.decoder3 = Decoder3d(128, 64)
        self.decoder2 = Decoder3d(64, 32)
        self.decoder1 = Decoder3d(32, 16)
        self.deconv4 = deconv(256, 128)
        self.deconv3 = deconv(128, 64)
        self.deconv2 = deconv(64, 32)
        self.deconv1 = deconv(32, 16)
        self.final = nn.Conv3d(16, out_channels, kernel_size=1)
        initialize_weights(self)

    def forward(self, x):
        enc_input = self.enc_input(x)
        down1 = self.downsample(enc_input)

        enc1 = self.encoder1(down1)
        down2 = self.downsample(enc1)

        enc2 = self.encoder2(down2)
        down3 = self.downsample(enc2)

        enc3 = self.encoder3(down3)
        down4 = self.downsample(enc3)

        input_feature = self.encoder4(down4)

        # Do Attenttion operations here
        attention = self.affinity_attention(input_feature)
        attention_fuse = input_feature + attention

        # Do decoder operations here
        up4 = self.deconv4(attention_fuse)
        up4 = torch.cat((enc3, up4), dim=1)
        dec4 = self.decoder4(up4)

        up3 = self.deconv3(dec4)
        up3 = torch.cat((enc2, up3), dim=1)
        dec3 = self.decoder3(up3)

        up2 = self.deconv2(dec3)
        up2 = torch.cat((enc1, up2), dim=1)
        dec2 = self.decoder2(up2)

        up1 = self.deconv1(dec2)
        up1 = torch.cat((enc_input, up1), dim=1)
        dec1 = self.decoder1(up1)

        final = self.final(dec1)
        final = F.sigmoid(final)
        return final


if __name__ == "__main__":
    from torch.distributions.normal import Normal
    model1 = CSNet3D(2, 1)
    model2 = CSNet3D(2, 4)


    # from torch.distributions.normal import Normal
    # d1 = model1.state_dict()
    # d2 = model2.state_dict()
    # for k2, v2 in d2.items():
    #     if d1[k2].shape != v2.shape:
    #         assert d1[k2].shape[1] != v2.shape[1] and len(d1[k2].shape) > 1 and len(v2.shape) > 1
    #         noise_shape = torch.tensor(v2.shape)
    #         noise_shape[1] = v2.shape[1] - d1[k2].shape[1]
    #         noise = nn.Parameter(Normal(0, 1e-5).sample(noise_shape)).to(d1[k2].device)
    #         d1[k2] = torch.cat((d1[k2], noise), dim=1)
    #         print(f"key: {k2}, shape1: {d1[k2].shape}, mean1: {d1[k2].mean()}, shape2: {v2.shape}, mean2: {v2.mean()}")

    d1 = torch.load('H:/Graduate_project/segment_server/experiments/Graduate_project/two_stage2/first_stage/CS2net/checkpoint/best_metric_model.pth')
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model1.to(device)
    model1.load_state_dict(d1)
    d2 = model2.state_dict()
    print("-----------------------------load checkpoint and modify-----------------------------------------")
    for k2, v2 in d2.items():
        if d1[k2].shape != v2.shape:
            assert d1[k2].shape[1] != v2.shape[1] and len(d1[k2].shape) > 1 and len(v2.shape) > 1
            noise_shape = torch.tensor(v2.shape)
```

```
            noise_shape[1] = v2.shape[1] - d1[k2].shape[1]
            noise = nn.Parameter(Normal(0, 1e-7).sample(noise_shape)).to(d1[k2].device)
            print(f"key: {k2}, shape1: {d1[k2].shape}, mean1: {d1[k2].mean()}, shape2: {v2.shape}, mean2: {v2.mean()}")
            d1[k2] = torch.cat((d1[k2], noise), dim=1)
    model2.to(device)
    model2.load_state_dict(d1)
    x1 = torch.ones(1, 1, 64, 64, 64).to(device)
    x2 = torch.ones(1, 4, 64, 64, 64).to(device)
    y1 = model1(x1)
    y2 = model2(x2)
    print(((y1 - y2)**2).max(), ((y1 - y2)**2).mean())
    pass
```

```python
SparrowLink/model/denseunet_skip.py
import warnings
from typing import Optional, Sequence, Tuple, Union

import torch
import torch.nn as nn
import torch
import torch.nn as nn
import torch.nn.functional as F


from monai.networks.blocks.convolutions import Convolution, ResidualUnit
from monai.networks.layers.factories import Act, Norm
from monai.networks.layers.simplelayers import SkipConnection
from monai.utils import alias, deprecated_arg, export
from monai.networks.layers.convutils import same_padding

@export("monai.networks.nets")
@alias("Unet")
class SkipDenseUNet(nn.Module):

    @deprecated_arg(
        name="dimensions", new_name="spatial_dims", since="0.6", msg_suffix="Please use `spatial_dims` instead."
    )
    def __init__(
            self,
            spatial_dims: int,
            in_channels: int,
            out_channels: int,
            channels: Sequence[int],
            strides: Sequence[int],
            kernel_size: Union[Sequence[int], int] = 3,
            up_kernel_size: Union[Sequence[int], int] = 3,
            num_res_units: int = 0,
            act: Union[Tuple, str] = Act.PRELU,
            norm: Union[Tuple, str] = Norm.INSTANCE,
            dropout: float = 0.0,
            bias: bool = True,
            adn_ordering: str = "NDA",
            dimensions: Optional[int] = None,
    ) -> None:

        super().__init__()

        if len(channels) < 2:
            raise ValueError("the length of `channels` should be no less than 2.")
        delta = len(strides) - (len(channels) - 1)
        if delta < 0:
            raise ValueError("the length of `strides` should equal to `len(channels) - 1`.")
        if delta > 0:
            warnings.warn(f"`len(strides) > len(channels) - 1`, the last {delta} values of strides will not be used.")
        if dimensions is not None:
            spatial_dims = dimensions
        if isinstance(kernel_size, Sequence):
            if len(kernel_size) != spatial_dims:
                raise ValueError("the length of `kernel_size` should equal to `dimensions`.")
        if isinstance(up_kernel_size, Sequence):
            if len(up_kernel_size) != spatial_dims:
                raise ValueError("the length of `up_kernel_size` should equal to `dimensions`.")

        self.dimensions = spatial_dims
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.channels = channels
        self.strides = strides
        self.kernel_size = kernel_size
        self.up_kernel_size = up_kernel_size
        self.num_res_units = num_res_units
        self.act = act
        self.norm = norm
        self.dropout = dropout
        self.bias = bias
        self.adn_ordering = adn_ordering

        def _create_block(
                inc: int, outc: int, channels: Sequence[int], strides: Sequence[int], is_top: bool
        ) -> nn.Module:
            """
            Builds the UNet structure from the bottom up by recursing down to the bottom block, then creating sequential
            blocks containing the downsample path, a skip connection around the previous block, and the upsample path.

            Args:
                inc: number of input channels.
                outc: number of output channels.
                channels: sequence of channels. Top block first.
                strides: convolution stride.
                is_top: True if this is the top block.
            """
            c = channels[0]
            s = strides[0]

            subblock: nn.Module

            if len(channels) > 2:
                subblock = _create_block(c, c, channels[1:], strides[1:], False)  # continue recursion down
                upc = c * 2
            else:
                # the next layer is the bottom so stop recursion, create the bottom layer as the sublock for this layer
                subblock = self._get_bottom_layer(c, channels[1])
                upc = c + channels[1]

            down = self._get_down_layer(inc, c, s, is_top)  # create layer in downsampling path
            up = self._get_up_layer(upc, outc, s, is_top)  # create layer in upsampling path

            return self._get_connection_block(down, up, subblock)

        self.model = _create_block(in_channels, out_channels, self.channels, self.strides, True)

    def _get_connection_block(self, down_path: nn.Module, up_path: nn.Module, subblock: nn.Module) -> nn.Module:
        """
        Returns the block object defining a layer of the UNet structure including the implementation of the skip
        between encoding (down) and decoding (up) sides of the network.

        Args:
            down_path: encoding half of the layer
            up_path: decoding half of the layer
            subblock: block defining the next layer in the network.
        Returns: block for this layer: `nn.Sequential(down_path, SkipConnection(subblock), up_path)`
        """
        return nn.Sequential(down_path, SkipConnection(subblock), up_path)
```

```
120
121     def _get_down_layer(self, in_channels: int, out_channels: int, strides: int, is_top: bool) -> nn.Module:
122         """
123         Returns the encoding (down) part of a layer of the network. This typically will downsample data at some point
124         in its structure. Its output is used as input to the next layer down and is concatenated with output from the
125         next layer to form the input for the decode (up) part of the layer.
126
127         Args:
128             in_channels: number of input channels.
129             out_channels: number of output channels.
130             strides: convolution stride.
131             is_top: True if this is the top block.
132         """
133         mod: nn.Module
134         mod = DenseUnit(
135             in_channels=in_channels,
136             out_channels=out_channels,
137             strides=strides,
138         )
139         return mod
140
141     def _get_bottom_layer(self, in_channels: int, out_channels: int) -> nn.Module:
142         """
143         Returns the bottom or bottleneck layer at the bottom of the network linking encode to decode halves.
144
145         Args:
146             in_channels: number of input channels.
147             out_channels: number of output channels.
148         """
149         return self._get_down_layer(in_channels, out_channels, 1, False)
150
151     def _get_up_layer(self, in_channels: int, out_channels: int, strides: int, is_top: bool) -> nn.Module:
152         """
153         Returns the decoding (up) part of a layer of the network. This typically will upsample data at some point
154         in its structure. Its output is used as input to the next layer up.
155
156         Args:
157             in_channels: number of input channels.
158             out_channels: number of output channels.
159             strides: convolution stride.
160             is_top: True if this is the top block.
161         """
162         conv: Union[Convolution, nn.Sequential]
163
164         conv = Convolution(
165             self.dimensions,
166             in_channels,
167             out_channels,
168             strides=strides,
169             kernel_size=self.up_kernel_size,
170             act=self.act,
171             norm=self.norm,
172             dropout=self.dropout,
173             bias=self.bias,
174             conv_only=is_top and self.num_res_units == 0,
175             is_transposed=True,
176             adn_ordering=self.adn_ordering,
177         )
178
179         if self.num_res_units > 0:
180             ru = ResidualUnit(
181                 self.dimensions,
182                 out_channels,
183                 out_channels,
184                 strides=1,
185                 kernel_size=self.kernel_size,
186                 subunits=1,
187                 act=self.act,
188                 norm=self.norm,
189                 dropout=self.dropout,
190                 bias=self.bias,
191                 last_conv_only=is_top,
192                 adn_ordering=self.adn_ordering,
193             )
194             conv = nn.Sequential(conv, ru)
195
196         return conv
197
198     def forward(self, x: torch.Tensor) -> torch.Tensor:
199         x = self.model(x)
200         return x
201
202
203 class DenseUnit(nn.Module):
204     """
205     Residual module with multiple convolutions and a residual connection.
206
207     For example:
208
209     .. code-block:: python
210
211         from monai.networks.blocks import ResidualUnit
212
213         convs = ResidualUnit(
214             spatial_dims=3,
215             in_channels=1,
216             out_channels=1,
217             adn_ordering="AN",
218             act=("prelu", {"init": 0.2}),
219             norm=("layer", {"normalized_shape": (10, 10, 10)}),
220         )
221         print(convs)
222
223     output::
224
225         DenseUnit(
226           (conv): Sequential(
227             (unit0): Convolution(
228               (conv): Conv3d(1, 1, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
229               (adn): ADN(
230                 (A): PReLU(num_parameters=1)
231                 (N): LayerNorm((10, 10, 10), eps=1e-05, elementwise_affine=True)
232               )
233             )
234             (unit1): Convolution(
235               (conv): Conv3d(1, 1, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
236               (adn): ADN(
237                 (A): PReLU(num_parameters=1)
238                 (N): LayerNorm((10, 10, 10), eps=1e-05, elementwise_affine=True)
239               )
240             )
241           )
242           (residual): Identity()
```

```
            (residual): Identity()
        )

    """

    def __init__(
        self,
        in_channels: int,
        out_channels: int,
        strides: Union[Sequence[int], int] = 1,
        kernel_size: Union[Sequence[int], int] = 3,
        subunits: int = 2,
        adn_ordering: str = "NDA",
        act: Optional[Union[Tuple, str]] = "PRELU",
        norm: Optional[Union[Tuple, str]] = "INSTANCE",
        dropout: Optional[Union[Tuple, str, float]] = None,
        dropout_dim: Optional[int] = 1,
        dilation: Union[Sequence[int], int] = 1,
        bias: bool = True,
        last_conv_only: bool = False,
        padding: Optional[Union[Sequence[int], int]] = None,
    ) -> None:
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.strides = strides
        self.dense_blocks = DenseBlock(in_channels=self.in_channels,
                                       out_channels=self.out_channels,
                                       nb_layers=subunits)
        self.transition = TransitionLayer(in_channels=self.out_channels,
                                          out_channels=self.out_channels,
                                          stride=self.strides)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.dense_blocks(x)
        x = self.transition(x)
        return x


class BottleneckLayer(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(BottleneckLayer, self).__init__()
        self.conv1 = nn.Conv3d(in_channels, out_channels, kernel_size=1, stride=1, padding=0)
        self.bn1 = nn.BatchNorm3d(out_channels)
        self.relu1 = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv3d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm3d(out_channels)
        self.relu2 = nn.ReLU(inplace=True)

    def forward(self, x):
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu1(out)
        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu2(out)
        return out


class TransitionLayer(nn.Module):
    def __init__(self, in_channels, out_channels, stride=2):
        super(TransitionLayer, self).__init__()
        self.conv = nn.Conv3d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1)
        self.bn = nn.BatchNorm3d(out_channels)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        out = self.conv(x)
        out = self.bn(out)
        out = self.relu(out)
        return out


class DenseBlock(nn.Module):
    def __init__(self, in_channels, out_channels, nb_layers=4):
        super(DenseBlock, self).__init__()
        self.layers = nn.ModuleList()
        for i in range(nb_layers):
            self.layers.append(BottleneckLayer(in_channels + i * out_channels, out_channels))
        self.merge = nn.Sequential(
            nn.Conv3d(in_channels + nb_layers * out_channels, out_channels, kernel_size=1, stride=1),
            nn.BatchNorm3d(out_channels),
            nn.ReLU(inplace=True),
        )

    def forward(self, x):
        layers_concat = [x]
        for layer in self.layers:
            out = layer(Concatenation(layers_concat))
            layers_concat.append(out)
        return self.merge(Concatenation(layers_concat))


def Concatenation(layers):
    return torch.cat(layers, dim=1)


if __name__ == "__main__":
    import torch

    model1 = SkipDenseUNet(
        spatial_dims=3,
        in_channels=1,
        out_channels=2,
        channels=(16, 32, 64, 128, 256),
        strides=(2, 2, 2, 2),
        num_res_units=2,
        norm=Norm.BATCH,)

    model2 = SkipDenseUNet(
        spatial_dims=3,
        in_channels=4,
        out_channels=2,
        channels=(16, 32, 64, 128, 256),
        strides=(2, 2, 2, 2),
        num_res_units=2,
        norm=Norm.BATCH, )

    from torch.distributions.normal import Normal
    d1 = torch.load('H:/Graduate_project/segment_server/experiments/Graduate_project/two_stage2/first_stage/DenseUnet/checkpoint/best_metric_model.pth')
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model1.to(device)
```

```
    model1.load_state_dict(d1)
    d2 = model2.state_dict()
    in_channels = 4
    print("----------------------------load checkpoint and modify-----------------------------------------")
    for k2, v2 in d2.items():
        if d1[k2].shape != v2.shape:
            assert d1[k2].shape[1] != v2.shape[1] and len(d1[k2].shape) > 1 and len(v2.shape) > 1
            noise_shape = torch.tensor(v2.shape)
            noise_shape[1] = v2.shape[1] - d1[k2].shape[1]
            noise = nn.Parameter(Normal(0, 1e-10).sample(noise_shape)).to(d1[k2].device)
            print(f"key: {k2}, shape1: {d1[k2].shape}, mean1: {d1[k2].mean()}, shape2: {v2.shape}, mean2: {v2.mean()}")
            d1[k2] = torch.cat((d1[k2][:, :1, ...], noise, d1[k2][:, 1:, ...]), dim=1)
    model2.to(device)
    model2.load_state_dict(d1)
    x1 = torch.ones(1, 1, 64, 64, 64).to(device)
    x2 = torch.ones(1, 4, 64, 64, 64).to(device)
    y1 = model1(x1)
    y2 = model2(x2)
    print(((y1 - y2)**2).max(), ((y1 - y2)**2).mean())
    pass
```

H:\git\Repo2PDF\dist\repo2pdfAPP\SparrowLink\model\swin_unet.py

```
SparrowLink/model/swin_unet.py
from functools import reduce, lru_cache
from operator import mul

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.checkpoint as checkpoint
from einops import rearrange
# from mmcv.runner import load_checkpoint
from timm.models.layers import DropPath, trunc_normal_
from IPython import embed

class Mlp(nn.Module):
    """ Multilayer perceptron."""

    def __init__(self, in_features, hidden_features=None, out_features=None, act_layer=nn.GELU, drop=0.):
        super().__init__()
        out_features = out_features or in_features
        hidden_features = hidden_features or in_features
        self.fc1 = nn.Linear(in_features, hidden_features)
        self.act = act_layer()
        self.fc2 = nn.Linear(hidden_features, out_features)
        self.drop = nn.Dropout(drop)

    def forward(self, x):
        x = self.fc1(x)
        x = self.act(x)
        x = self.drop(x)
        x = self.fc2(x)
        x = self.drop(x)
        return x


def img2windows(img, H_sp, W_sp):
    """
    img: B C D H W
    """
    B, C, D, H, W = img.shape
    img_reshape = img.view(B, C, D, H // H_sp, H_sp, W // W_sp, W_sp)
    img_perm = img_reshape.permute(0, 2, 3, 5, 4, 6, 1).contiguous().reshape(-1, D * H_sp * W_sp, C)
    return img_perm


def windows2img(img_splits_hw, H_sp, W_sp, D, H, W):
    """
    img_splits_hw: B' D H W C
    """
    B = int(img_splits_hw.shape[0] / (D * H * W / H_sp / W_sp))

    img = img_splits_hw.view(B, D, H // H_sp, W // W_sp, H_sp, W_sp, -1)
    img = img.permute(0, 1, 2, 4, 3, 5, 6).contiguous().view(B, D, H, W, -1)

    return img


class Merge_Block(nn.Module):
    def __init__(self, dim, dim_out, norm_layer=nn.LayerNorm):
        super().__init__()
        self.conv = nn.Conv3d(dim, dim_out, 3, 2, 1)
        self.norm = norm_layer(dim_out)

    def forward(self, x):
        B, new_HW, C = x.shape
        D = 32
        H = W = int(np.sqrt(new_HW // D))
        x = x.transpose(-2, -1).contiguous().view(B, C, D, H, W)
        x = self.conv(x)
        B, C = x.shape[:2]
        x = x.view(B, C, -1).transpose(-2, -1).contiguous()
        x = self.norm(x)

        return x


def window_partition(x, window_size):
    """
    Args:
        x: (B, D, H, W, C)
        window_size (tuple[int]): window size

    Returns:
        windows: (B*num_windows, window_size*window_size, C)
    """
    B, D, H, W, C = x.shape
    x = x.view(B, D // window_size[0], window_size[0], H // window_size[1], window_size[1], W // window_size[2],
               window_size[2], C)
    windows = x.permute(0, 1, 3, 5, 2, 4, 6, 7).contiguous().view(-1, reduce(mul, window_size), C)
    return windows


def window_reverse(windows, window_size, B, D, H, W):
    """
    Args:
        windows: (B*num_windows, window_size, window_size, C)
        window_size (tuple[int]): Window size
        H (int): Height of image
        W (int): Width of image
```

```
    Returns:
        x: (B, D, H, W, C)
    """
    x = windows.view(B, D // window_size[0], H // window_size[1], W // window_size[2], window_size[0], window_size[1],
                     window_size[2], -1)
    x = x.permute(0, 1, 4, 2, 5, 3, 6, 7).contiguous().view(B, D, H, W, -1)
    return x


def get_window_size(x_size, window_size, shift_size=None):
    use_window_size = list(window_size)
    if shift_size is not None:
        use_shift_size = list(shift_size)
    for i in range(len(x_size)):
        if x_size[i] <= window_size[i]:
            use_window_size[i] = x_size[i]
            if shift_size is not None:
                use_shift_size[i] = 0

    if shift_size is None:
        return tuple(use_window_size)
    else:
        return tuple(use_window_size), tuple(use_shift_size)


class WindowAttention3D(nn.Module):
    """ Window based multi-head self attention (W-MSA) module with relative position bias.
    It supports both of shifted and non-shifted window.
    Args:
        dim (int): Number of input channels.
        window_size (tuple[int]): The temporal length, height and width of the window.
        num_heads (int): Number of attention heads.
        qkv_bias (bool, optional):  If True, add a learnable bias to query, key, value. Default: True
        qk_scale (float | None, optional): Override default qk scale of head_dim ** -0.5 if set
        attn_drop (float, optional): Dropout ratio of attention weight. Default: 0.0
        proj_drop (float, optional): Dropout ratio of output. Default: 0.0
    """

    def __init__(self, dim, window_size, num_heads, qkv_bias=False, qk_scale=None, attn_drop=0., proj_drop=0.):

        super().__init__()
        self.dim = dim
        self.window_size = window_size  # Wd, Wh, Ww
        self.num_heads = num_heads
        head_dim = dim // num_heads
        self.scale = qk_scale or head_dim ** -0.5

        # define a parameter table of relative position bias
        self.relative_position_bias_table = nn.Parameter(
            torch.zeros((2 * window_size[0] - 1) * (2 * window_size[1] - 1) * (2 * window_size[2] - 1),
                        num_heads))  # 2*Wd-1 * 2*Wh-1 * 2*Ww-1, nH

        # get pair-wise relative position index for each token inside the window
        coords_d = torch.arange(self.window_size[0])
        coords_h = torch.arange(self.window_size[1])
        coords_w = torch.arange(self.window_size[2])
        coords = torch.stack(torch.meshgrid(coords_d, coords_h, coords_w))  # 3, Wd, Wh, Ww
        coords_flatten = torch.flatten(coords, 1)  # 3, Wd*Wh*Ww
        relative_coords = coords_flatten[:, :, None] - coords_flatten[:, None, :]  # 3, Wd*Wh*Ww, Wd*Wh*Ww
        relative_coords = relative_coords.permute(1, 2, 0).contiguous()  # Wd*Wh*Ww, Wd*Wh*Ww, 3
        relative_coords[:, :, 0] += self.window_size[0] - 1  # shift to start from 0
        relative_coords[:, :, 1] += self.window_size[1] - 1
        relative_coords[:, :, 2] += self.window_size[2] - 1

        relative_coords[:, :, 0] *= (2 * self.window_size[1] - 1) * (2 * self.window_size[2] - 1)
        relative_coords[:, :, 1] *= (2 * self.window_size[2] - 1)
        relative_position_index = relative_coords.sum(-1)  # Wd*Wh*Ww, Wd*Wh*Ww
        self.register_buffer("relative_position_index", relative_position_index)

        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_drop)

        trunc_normal_(self.relative_position_bias_table, std=.02)
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, x, mask=None, prev_v=None, prev_k=None, prev_q=None, is_decoder=False):
        """ Forward function.
        Args:
            x: input features with shape of (num_windows*B, N, C)
            mask: (0/-inf) mask with shape of (num_windows, N, N) or None
        """
        B_, N, C = x.shape
        qkv = self.qkv(x).reshape(B_, N, 3, self.num_heads, C // self.num_heads).permute(2, 0, 3, 1, 4)
        q, k, v = qkv[0], qkv[1], qkv[2]  # B_, nH, N, C

        q = q * self.scale
        attn = q @ k.transpose(-2, -1)

        relative_position_bias = self.relative_position_bias_table[
            self.relative_position_index[:N, :N].reshape(-1)].reshape(
            N, N, -1)  # Wd*Wh*Ww,Wd*Wh*Ww,nH
        relative_position_bias = relative_position_bias.permute(2, 0, 1).contiguous()  # nH, Wd*Wh*Ww, Wd*Wh*Ww
        attn = attn + relative_position_bias.unsqueeze(0)  # B_, nH, N, N

        if mask is not None:
            nW = mask.shape[0]
            attn = attn.view(B_ // nW, nW, self.num_heads, N, N) + mask.unsqueeze(1).unsqueeze(0)
            attn = attn.view(-1, self.num_heads, N, N)
            attn = self.softmax(attn)
        else:
            attn = self.softmax(attn)

        attn = self.attn_drop(attn)

        x = (attn @ v).transpose(1, 2).reshape(B_, N, C)
        x = self.proj(x)
        x = self.proj_drop(x)
        x2 = None

        if is_decoder:
            q = q * self.scale
            attn2 = q @ prev_k.transpose(-2, -1)
            attn2 = attn2 + relative_position_bias.unsqueeze(0)

            if mask is not None:
                nW = mask.shape[0]
                attn2 = attn2.view(B_ // nW, nW, self.num_heads, N, N) + mask.unsqueeze(1).unsqueeze(0)
                attn2 = attn2.view(-1, self.num_heads, N, N)
                attn2 = self.softmax(attn2)
```

```
            else:
                attn2 = self.softmax(attn2)

            attn2 = self.attn_drop(attn2)

            x2 = (attn2 @ prev_v).transpose(1, 2).reshape(B_, N, C)
            x2 = self.proj(x2)
            x2 = self.proj_drop(x2)

        return x, x2, v, k, q


class PositionalEncoding3D(nn.Module):
    def __init__(self, channels):
        """
        :param channels: The last dimension of the tensor you want to apply pos emb to.
        """
        super(PositionalEncoding3D, self).__init__()
        channels = int(np.ceil(channels / 6) * 2)
        if channels % 2:
            channels += 1
        self.channels = channels
        inv_freq = 1. / (10000 ** (torch.arange(0, channels, 2).float() / channels))
        self.register_buffer('inv_freq', inv_freq)

    def forward(self, tensor):
        """
        :param tensor: A 5d tensor of size (batch_size, x, y, z, ch)
        :return: Positional Encoding Matrix of size (batch_size, x, y, z, ch)
        """
        if len(tensor.shape) != 5:
            raise RuntimeError("The input tensor has to be 5d!")
        batch_size, x, y, z, orig_ch = tensor.shape
        pos_x = torch.arange(x, device=tensor.device).type(self.inv_freq.type())
        pos_y = torch.arange(y, device=tensor.device).type(self.inv_freq.type())
        pos_z = torch.arange(z, device=tensor.device).type(self.inv_freq.type())
        sin_inp_x = torch.einsum("i,j->ij", pos_x, self.inv_freq)
        sin_inp_y = torch.einsum("i,j->ij", pos_y, self.inv_freq)
        sin_inp_z = torch.einsum("i,j->ij", pos_z, self.inv_freq)
        emb_x = torch.cat((sin_inp_x.sin(), sin_inp_x.cos()), dim=-1).unsqueeze(1).unsqueeze(1)
        emb_y = torch.cat((sin_inp_y.sin(), sin_inp_y.cos()), dim=-1).unsqueeze(1)
        emb_z = torch.cat((sin_inp_z.sin(), sin_inp_z.cos()), dim=-1)
        emb = torch.zeros((x, y, z, self.channels * 3), device=tensor.device).type(tensor.type())
        emb[:, :, :, :self.channels] = emb_x
        emb[:, :, :, self.channels:2 * self.channels] = emb_y
        emb[:, :, :, 2 * self.channels:] = emb_z

        return emb[None, :, :, :, :orig_ch].repeat(batch_size, 1, 1, 1, 1)


class SwinTransformerBlock3D(nn.Module):
    """ Swin Transformer Block.

    Args:
        dim (int): Number of input channels.
        num_heads (int): Number of attention heads.
        window_size (tuple[int]): Window size.
        shift_size (tuple[int]): Shift size for SW-MSA.
        mlp_ratio (float): Ratio of mlp hidden dim to embedding dim.
        qkv_bias (bool, optional): If True, add a learnable bias to query, key, value. Default: True
        qk_scale (float | None, optional): Override default qk scale of head_dim ** -0.5 if set.
        drop (float, optional): Dropout rate. Default: 0.0
        attn_drop (float, optional): Attention dropout rate. Default: 0.0
        drop_path (float, optional): Stochastic depth rate. Default: 0.0
        act_layer (nn.Module, optional): Activation layer. Default: nn.GELU
        norm_layer (nn.Module, optional): Normalization layer.  Default: nn.LayerNorm
    """

    def __init__(self, dim, num_heads, window_size=(7, 7, 7), shift_size=(0, 0, 0),
                 mlp_ratio=4., qkv_bias=True, qk_scale=None, drop=0., attn_drop=0., drop_path=0.,
                 act_layer=nn.GELU, norm_layer=nn.LayerNorm, use_checkpoint=False):
        super().__init__()
        self.dim = dim
        self.num_heads = num_heads
        self.window_size = window_size
        self.shift_size = shift_size
        self.mlp_ratio = mlp_ratio
        self.use_checkpoint = use_checkpoint

        assert 0 <= self.shift_size[0] < self.window_size[0], "shift_size must in 0-window_size"
        assert 0 <= self.shift_size[1] < self.window_size[1], "shift_size must in 0-window_size"
        assert 0 <= self.shift_size[2] < self.window_size[2], "shift_size must in 0-window_size"

        self.norm1 = norm_layer(dim)
        self.attn = WindowAttention3D(
            dim, window_size=self.window_size, num_heads=num_heads,
            qkv_bias=qkv_bias, qk_scale=qk_scale, attn_drop=attn_drop, proj_drop=drop)

        self.drop_path = DropPath(drop_path) if drop_path > 0. else nn.Identity()
        self.norm2 = norm_layer(dim)
        mlp_hidden_dim = int(dim * mlp_ratio)
        self.mlp = Mlp(in_features=dim, hidden_features=mlp_hidden_dim, act_layer=act_layer, drop=drop)

    def forward_part1(self, x, mask_matrix, prev_v, prev_k, prev_q, is_decoder):
        B, D, H, W, C = x.shape
        window_size, shift_size = get_window_size((D, H, W), self.window_size, self.shift_size)

        x = self.norm1(x)
        # pad feature maps to multiples of window size
        pad_l = pad_t = pad_d0 = 0
        pad_d1 = (window_size[0] - D % window_size[0]) % window_size[0]
        pad_b = (window_size[1] - H % window_size[1]) % window_size[1]
        pad_r = (window_size[2] - W % window_size[2]) % window_size[2]
        x = F.pad(x, (0, 0, pad_l, pad_r, pad_t, pad_b, pad_d0, pad_d1))
        _, Dp, Hp, Wp, _ = x.shape
        # cyclic shift
        if any(i > 0 for i in shift_size):
            shifted_x = torch.roll(x, shifts=(-shift_size[0], -shift_size[1], -shift_size[2]), dims=(1, 2, 3))
            attn_mask = mask_matrix
        else:
            shifted_x = x
            attn_mask = None
        # partition windows
        x_windows = window_partition(shifted_x, window_size)  # B*nW, Wd*Wh*Ww, C
        # W-MSA/SW-MSA
        attn_windows, cross_attn_windows, v, k, q = self.attn(x_windows, mask=attn_mask, prev_v=prev_v, prev_k=prev_k,
                                                              prev_q=prev_q, is_decoder=is_decoder)  # B*nW, Wd*Wh*Ww, C

        # merge windows
        attn_windows = attn_windows.view(-1, *(window_size + (C,)))
        shifted_x = window_reverse(attn_windows, window_size, B, Dp, Hp, Wp)  # B D' H' W' C
        # reverse cyclic shift
```

```
        if any(i > 0 for i in shift_size):
            x = torch.roll(shifted_x, shifts=(shift_size[0], shift_size[1], shift_size[2]), dims=(1, 2, 3))
        else:
            x = shifted_x

        x2 = None
        if pad_d1 > 0 or pad_r > 0 or pad_b > 0:
            x = x[:, :D, :H, :W, :].contiguous()

        if cross_attn_windows is not None:
            # merge windows
            cross_attn_windows = cross_attn_windows.view(-1, *(window_size + (C,)))
            cross_shifted_x = window_reverse(cross_attn_windows, window_size, B, Dp, Hp, Wp)  # B D' H' W' C
            # reverse cyclic shift
            if any(i > 0 for i in shift_size):
                x2 = torch.roll(cross_shifted_x, shifts=(shift_size[0], shift_size[1], shift_size[2]), dims=(1, 2, 3))
            else:
                x2 = cross_shifted_x

            if pad_d1 > 0 or pad_r > 0 or pad_b > 0:
                x2 = x2[:, :D, :H, :W, :].contiguous()

        return x, x2, v, k, q

    def forward_part2(self, x):
        return self.drop_path(self.mlp(self.norm2(x)))

    def forward_part3(self, x):
        return self.mlp(self.norm2(x))

    def forward(self, x, mask_matrix, prev_v, prev_k, prev_q, is_decoder=False):
        """ Forward function.

        Args:
            x: Input feature, tensor size (B, D, H, W, C).
            mask_matrix: Attention mask for cyclic shift.
        """

        alpha = 0.5
        shortcut = x
        x2, v, k, q = None, None, None, None

        if self.use_checkpoint:
            x = checkpoint.checkpoint(self.forward_part1, x, mask_matrix)
        else:
            x, x2, v, k, q = self.forward_part1(x, mask_matrix, prev_v, prev_k, prev_q, is_decoder)

        x = shortcut + self.drop_path(x)

        if self.use_checkpoint:
            x = x + checkpoint.checkpoint(self.forward_part2, x)
        else:
            x = x + self.forward_part2(x)

        if x2 is not None:
            x2 = shortcut + self.drop_path(x2)
            if self.use_checkpoint:
                x2 = x2 + checkpoint.checkpoint(self.forward_part2, x2)
            else:
                x2 = x2 + self.forward_part2(x2)

            FPE = PositionalEncoding3D(x.shape[4])

            x = torch.add((1 - alpha) * x, alpha * x2) + self.forward_part3(FPE(x))

        return x, v, k, q


class PatchMerging(nn.Module):
    """ Patch Merging Layer

    Args:
        dim (int): Number of input channels.
        norm_layer (nn.Module, optional): Normalization layer.  Default: nn.LayerNorm
    """

    def __init__(self, dim, norm_layer=nn.LayerNorm):
        super().__init__()
        self.dim = dim
        self.reduction = nn.Linear(4 * dim, 2 * dim, bias=False)
        self.norm = norm_layer(4 * dim)

    def forward(self, x):
        """ Forward function.

        Args:
            x: Input feature, tensor size (B, D, H, W, C).
        """
        B, D, H, W, C = x.shape

        # padding
        pad_input = (H % 2 == 1) or (W % 2 == 1)
        if pad_input:
            x = F.pad(x, (0, 0, 0, W % 2, 0, H % 2))

        x0 = x[:, :, 0::2, 0::2, :]  # B D H/2 W/2 C
        x1 = x[:, :, 1::2, 0::2, :]  # B D H/2 W/2 C
        x2 = x[:, :, 0::2, 1::2, :]  # B D H/2 W/2 C
        x3 = x[:, :, 1::2, 1::2, :]  # B D H/2 W/2 C
        x = torch.cat([x0, x1, x2, x3], -1)  # B D H/2 W/2 4*C

        x = self.norm(x)
        x = self.reduction(x)

        return x


class PatchExpand_Up(nn.Module):
    def __init__(self, input_resolution, dim, dim_scale=2, norm_layer=nn.LayerNorm):
        super().__init__()
        self.input_resolution = input_resolution
        self.dim_scale = dim_scale
        self.dim = dim
        self.expand = nn.Linear(dim, 2 * dim, bias=False) if dim_scale == 2 else nn.Identity()
        self.norm = norm_layer(dim // dim_scale)

    def forward(self, x):
        """
        x: B, H*W, C
        """
        D, H, W = self.input_resolution
        x = x.flatten(2).transpose(1, 2)
```

```
        x = self.expand(x)
        B, L, C = x.shape
        # assert L == D * H * W, "input feature has wrong size"

        x = x.view(B, 32, H, W, C)
        x = rearrange(x, 'b d h w (p1 p2 c)-> b d (h p1) (w p2) c', p1=self.dim_scale, p2=self.dim_scale, c=C // 4)

        x = self.norm(x)
        x = x.permute(0, 4, 1, 2, 3)

        return x


class PatchExpand(nn.Module):
    def __init__(self, input_resolution, dim, dim_scale=2, norm_layer=nn.LayerNorm):
        super().__init__()
        self.input_resolution = input_resolution
        self.dim_scale = dim_scale
        self.dim = dim
        # self.expand = nn.Linear(dim, 2 * dim, bias=False) if dim_scale == 2 else nn.Identity()
        self.expand = nn.Linear(dim, 2 * dim, bias=False) if dim_scale == 2 else nn.Identity()
        self.norm = norm_layer(dim // dim_scale)

    def forward(self, x):
        """
        x: B, H*W, C
        """
        D, H, W = self.input_resolution
        x = x.flatten(2).transpose(1, 2)
        x = self.expand(x)
        B, L, C = x.shape
        # assert L == D * H * W, "input feature has wrong size"

        x = x.view(B, D * 8, H, W, C)
        x = rearrange(x, 'b d h w (p1 p2 c)-> b d (h p1) (w p2) c', p1=self.dim_scale, p2=self.dim_scale, c=C // 4)

        x = self.norm(x)
        x = x.permute(0, 4, 1, 2, 3)

        return x


class FinalPatchExpand_X4(nn.Module):
    def __init__(self, input_resolution, dim, dim_scale=4, norm_layer=nn.LayerNorm):
        super().__init__()
        self.input_resolution = input_resolution
        self.dim = dim
        self.dim_scale = dim_scale
        self.expand = nn.Linear(dim, 4 * 16 * dim, bias=False)
        self.output_dim = dim
        self.norm = norm_layer(self.output_dim)

    def forward(self, x):
        """
        x: B, H*W, C
        """
        D, H, W = self.input_resolution
        x = x.permute(0, 4, 1, 2, 3)
        x = x.flatten(2).transpose(1, 2)
        x = self.expand(x)
        B, L, C = x.shape

        x = x.view(B, D, H, W, C)
        x = rearrange(x, 'b d h w (p1 p2 p3 c)-> b (d p1) (h p2) (w p3) c', p1=self.dim_scale, p2=self.dim_scale,
                      p3=self.dim_scale,
                      c=C // (self.dim_scale ** 3))
        # x = x.view(B, -1, self.output_dim)
        x = self.norm(x)

        return x


class BasicLayer_up(nn.Module):
    """ A basic Swin Transformer layer for one stage.

    Args:
        dim (int): Number of input channels.
        input_resolution (tuple[int]): Input resolution.
        depth (int): Number of blocks.
        num_heads (int): Number of attention heads.
        window_size tuple(int): Local window size.
        mlp_ratio (float): Ratio of mlp hidden dim to embedding dim.
        qkv_bias (bool, optional): If True, add a learnable bias to query, key, value. Default: True
        qk_scale (float | None, optional): Override default qk scale of head_dim ** -0.5 if set.
        drop (float, optional): Dropout rate. Default: 0.0
        attn_drop (float, optional): Attention dropout rate. Default: 0.0
        drop_path (float | tuple[float], optional): Stochastic depth rate. Default: 0.0
        norm_layer (nn.Module, optional): Normalization layer. Default: nn.LayerNorm
        downsample (nn.Module | None, optional): Downsample layer at the end of the layer. Default: None
        use_checkpoint (bool): Whether to use checkpointing to save memory. Default: False.
    """

    def __init__(self, dim, input_resolution, depth, num_heads, window_size=(7, 7, 7),
                 mlp_ratio=4., qkv_bias=True, qk_scale=None, drop=0., attn_drop=0.,
                 drop_path=0., norm_layer=nn.LayerNorm, upsample=None, use_checkpoint=False):

        super().__init__()
        self.dim = dim
        self.input_resolution = input_resolution
        self.window_size = window_size
        self.shift_size = tuple(i // 2 for i in window_size)
        self.depth = depth
        self.use_checkpoint = use_checkpoint

        # build blocks
        self.blocks = nn.ModuleList([
            SwinTransformerBlock3D(
                dim=dim,
                num_heads=num_heads,
                window_size=window_size,
                shift_size=(0, 0, 0) if (i % 2 == 0) else self.shift_size,
                mlp_ratio=mlp_ratio,
                qkv_bias=qkv_bias,
                qk_scale=qk_scale,
                drop=drop,
                attn_drop=attn_drop,
                drop_path=drop_path[i] if isinstance(drop_path, list) else drop_path,
                norm_layer=norm_layer,
                use_checkpoint=use_checkpoint,
            )
            for i in range(depth)])

```

```
        # patch merging layer
        if upsample is not None:
            self.upsample = PatchExpand_Up(input_resolution, dim=dim, dim_scale=2, norm_layer=norm_layer)
        else:
            self.upsample = None

    def forward(self, x, prev_v1, prev_k1, prev_q1, prev_v2, prev_k2, prev_q2):
        """ Forward function.

        Args:
            x: Input feature, tensor size (B, C, D, H, W).
        """
        # calculate attention mask for SW-MSA
        B, C, D, H, W = x.shape
        window_size, shift_size = get_window_size((D, H, W), self.window_size, self.shift_size)
        x = rearrange(x, 'b c d h w -> b d h w c')
        Dp = int(np.ceil(D / window_size[0])) * window_size[0]
        Hp = int(np.ceil(H / window_size[1])) * window_size[1]
        Wp = int(np.ceil(W / window_size[2])) * window_size[2]
        attn_mask = compute_mask(Dp, Hp, Wp, window_size, shift_size, x.device)

        for idx, blk in enumerate(self.blocks):
            if idx % 2 == 0:
                x, _, _, _ = blk(x, attn_mask, prev_v1, prev_k1, prev_q1, True)
            else:
                x, _, _, _ = blk(x, attn_mask, prev_v2, prev_k2, prev_q2, True)

        # x = x.view(B, D, H, W, -1)

        if self.upsample is not None:
            x = x.permute(0, 4, 1, 2, 3)
            x = self.upsample(x)
        # x = rearrange(x, 'b d h w c -> b c d h w')
        return x


# cache each stage results
@lru_cache()
def compute_mask(D, H, W, window_size, shift_size, device):
    img_mask = torch.zeros((1, D, H, W, 1), device=device)  # 1 Dp Hp Wp 1
    cnt = 0
    for d in slice(-window_size[0]), slice(-window_size[0], -shift_size[0]), slice(-shift_size[0], None):
        for h in slice(-window_size[1]), slice(-window_size[1], -shift_size[1]), slice(-shift_size[1], None):
            for w in slice(-window_size[2]), slice(-window_size[2], -shift_size[2]), slice(-shift_size[2], None):
                img_mask[:, d, h, w, :] = cnt
                cnt += 1
    mask_windows = window_partition(img_mask, window_size)  # nW, ws[0]*ws[1]*ws[2], 1
    mask_windows = mask_windows.squeeze(-1)  # nW, ws[0]*ws[1]*ws[2]
    attn_mask = mask_windows.unsqueeze(1) - mask_windows.unsqueeze(2)
    attn_mask = attn_mask.masked_fill(attn_mask != 0, float(-100.0)).masked_fill(attn_mask == 0, float(0.0))
    return attn_mask


class BasicLayer(nn.Module):
    """ A basic Swin Transformer layer for one stage.

    Args:
        dim (int): Number of feature channels
        depth (int): Depths of this stage.
        num_heads (int): Number of attention head.
        window_size (tuple[int]): Local window size. Default: (1,7,7).
        mlp_ratio (float): Ratio of mlp hidden dim to embedding dim. Default: 4.
        qkv_bias (bool, optional): If True, add a learnable bias to query, key, value. Default: True
        qk_scale (float | None, optional): Override default qk scale of head_dim ** -0.5 if set.
        drop (float, optional): Dropout rate. Default: 0.0
        attn_drop (float, optional): Attention dropout rate. Default: 0.0
        drop_path (float | tuple[float], optional): Stochastic depth rate. Default: 0.0
        norm_layer (nn.Module, optional): Normalization layer. Default: nn.LayerNorm
        downsample (nn.Module | None, optional): Downsample layer at the end of the layer. Default: None
    """

    def __init__(self,
                 dim,
                 depth,
                 depths,
                 num_heads,
                 window_size=(1, 7, 7),
                 mlp_ratio=4.,
                 qkv_bias=False,
                 qk_scale=None,
                 drop=0.,
                 attn_drop=0.,
                 drop_path=0.,
                 drop_path_rate=0.,
                 norm_layer=nn.LayerNorm,
                 downsample=None,
                 use_checkpoint=False):
        super().__init__()
        self.window_size = window_size
        self.shift_size = tuple(i // 2 for i in window_size)
        self.depth = depth
        self.use_checkpoint = use_checkpoint

        # build blocks
        self.blocks = nn.ModuleList([
            SwinTransformerBlock3D(
                dim=dim,
                num_heads=num_heads,
                window_size=window_size,
                shift_size=(0, 0, 0) if (i % 2 == 0) else self.shift_size,
                mlp_ratio=mlp_ratio,
                qkv_bias=qkv_bias,
                qk_scale=qk_scale,
                drop=drop,
                attn_drop=attn_drop,
                drop_path=drop_path[i] if isinstance(drop_path, list) else drop_path,
                norm_layer=norm_layer,
                use_checkpoint=use_checkpoint,
            )
            for i in range(depth)])

        self.downsample = downsample
        if self.downsample is not None:
            self.downsample = downsample(dim=dim, norm_layer=norm_layer)

    def forward(self, x, block_num):
        """ Forward function.

        Args:
            x: Input feature, tensor size (B, C, D, H, W).
        """
        # calculate attention mask for SW-MSA
```

```
        B, C, D, H, W = x.shape
        window_size, shift_size = get_window_size((D, H, W), self.window_size, self.shift_size)
        x = rearrange(x, 'b c d h w -> b d h w c')

        Dp = int(np.ceil(D / window_size[0])) * window_size[0]
        Hp = int(np.ceil(H / window_size[1])) * window_size[1]
        Wp = int(np.ceil(W / window_size[2])) * window_size[2]
        attn_mask = compute_mask(Dp, Hp, Wp, window_size, shift_size, x.device)

        v1, k1, q1, v2, k2, q2 = None, None, None, None, None, None

        for idx, blk in enumerate(self.blocks):
            if idx % 2 == 0:
                x, v1, k1, q1 = blk(x, attn_mask, None, None, None)
            else:
                x, v2, k2, q2 = blk(x, attn_mask, None, None, None)

        x = x.reshape(B, D, H, W, -1)

        if self.downsample is not None:
            x = self.downsample(x)
        x = rearrange(x, 'b d h w c -> b c d h w')

        return x, v1, k1, q1, v2, k2, q2


class PatchEmbed3D(nn.Module):
    """ Video to Patch Embedding.

    Args:
        patch_size (int): Patch token size. Default: (2,4,4).
        in_chans (int): Number of input video channels. Default: 3.
        embed_dim (int): Number of linear projection output channels. Default: 96.
        norm_layer (nn.Module, optional): Normalization layer. Default: None
    """

    def __init__(self, img_size=(128, 128, 128), patch_size=(4, 4, 4), in_chans=3, embed_dim=96, norm_layer=None):
        super().__init__()
        self.patch_size = patch_size

        self.in_chans = in_chans
        self.embed_dim = embed_dim
        patches_resolution = [img_size[0] // patch_size[0], img_size[1] // patch_size[1], img_size[1] // patch_size[1]]
        self.patches_resolution = patches_resolution

        self.proj = nn.Conv3d(in_chans, embed_dim, kernel_size=patch_size, stride=patch_size)
        if norm_layer is not None:
            self.norm = norm_layer(embed_dim)
        else:
            self.norm = None

    def forward(self, x):
        """Forward function."""
        # padding
        _, _, D, H, W = x.size()
        if W % self.patch_size[2] != 0:
            x = F.pad(x, (0, self.patch_size[2] - W % self.patch_size[2]))
        if H % self.patch_size[1] != 0:
            x = F.pad(x, (0, 0, 0, self.patch_size[1] - H % self.patch_size[1]))
        if D % self.patch_size[0] != 0:
            x = F.pad(x, (0, 0, 0, 0, 0, self.patch_size[0] - D % self.patch_size[0]))

        x = self.proj(x)  # B C D Wh Ww
        if self.norm is not None:
            D, Wh, Ww = x.size(2), x.size(3), x.size(4)
            x = x.flatten(2).transpose(1, 2)
            x = self.norm(x)
            x = x.transpose(1, 2).view(-1, self.embed_dim, D, Wh, Ww)

        return x


class SwinTransformerSys3D(nn.Module):
    r""" Swin Transformer
        A PyTorch impl of : `Swin Transformer: Hierarchical Vision Transformer using Shifted Windows`  -
          https://arxiv.org/pdf/2103.14030

    Args:
        img_size (int | tuple(int)): Input image size. Default 224
        patch_size (int | tuple(int)): Patch size. Default: 4
        in_chans (int): Number of input image channels. Default: 3
        num_classes (int): Number of classes for classification head. Default: 1000
        embed_dim (int): Patch embedding dimension. Default: 96
        depths (tuple(int)): Depth of each Swin Transformer layer.
        num_heads (tuple(int)): Number of attention heads in different layers.
        window_size (tuple(int)): Window size. Default: (7,7,7)
        mlp_ratio (float): Ratio of mlp hidden dim to embedding dim. Default: 4
        qkv_bias (bool): If True, add a learnable bias to query, key, value. Default: True
        qk_scale (float): Override default qk scale of head_dim ** -0.5 if set. Default: None
        drop_rate (float): Dropout rate. Default: 0
        attn_drop_rate (float): Attention dropout rate. Default: 0
        drop_path_rate (float): Stochastic depth rate. Default: 0.1
        norm_layer (nn.Module): Normalization layer. Default: nn.LayerNorm.
        ape (bool): If True, add absolute position embedding to the patch embedding. Default: False
        patch_norm (bool): If True, add normalization after patch embedding. Default: True
        use_checkpoint (bool): Whether to use checkpointing to save memory. Default: False
    """

    def __init__(self, pretrained=None,
                 pretrained2d=True,
                 img_size=(128, 128, 128),
                 patch_size=(4, 4, 4),
                 in_chans=4,
                 num_classes=3,
                 embed_dim=96,
                 depths=[2, 2, 2, 1],
                 depths_decoder=[1, 2, 2, 2],
                 num_heads=[3, 6, 12, 24],
                 window_size=(7, 7, 7),
                 mlp_ratio=4.,
                 qkv_bias=True,
                 qk_scale=None,
                 drop_rate=0.,
                 attn_drop_rate=0.,
                 drop_path_rate=0.1,
                 norm_layer=nn.LayerNorm,
                 patch_norm=True,
                 use_checkpoint=False,
                 frozen_stages=-1,
                 final_upsample="expand_first", **kwargs):
        super().__init__()
```

```
        print("SwinTransformerSys3D expand initial----depths:{};depths_decoder:{};drop_path_rate:{};num_classes:{};embed_dims:{};window:{}".format(
                depths,
                depths_decoder, drop_path_rate, num_classes, embed_dim, window_size))

        self.pretrained = pretrained # None
        self.pretrained2d = pretrained2d # True
        self.num_classes = num_classes # 3
        self.num_layers = len(depths) # number of swin transformer block
        self.embed_dim = embed_dim # number of patch embedding size (default: 96)
        self.patch_norm = patch_norm
        self.num_features = int(embed_dim * 2 ** (self.num_layers - 1))
        self.num_features_up = int(embed_dim * 2)
        self.mlp_ratio = mlp_ratio
        self.final_upsample = final_upsample
        self.frozen_stages = frozen_stages

        # split image into non-overlapping patches
        self.patch_embed = PatchEmbed3D(img_size=img_size,
                                        patch_size=patch_size, in_chans=in_chans, embed_dim=embed_dim,
                                        norm_layer=norm_layer if self.patch_norm else None)

        patches_resolution = self.patch_embed.patches_resolution
        self.patches_resolution = patches_resolution

        self.pos_drop = nn.Dropout(p=drop_rate)

        # stochastic depth
        dpr = [x.item() for x in torch.linspace(0, drop_path_rate, sum(depths))]  # stochastic depth decay rule

        # build encoder and bottleneck layers
        self.layers = nn.ModuleList()
        for i_layer in range(self.num_layers):
            layer = BasicLayer(
                dim=int(embed_dim * 2 ** i_layer),
                depth=depths[i_layer],
                depths=depths,
                num_heads=num_heads[i_layer],
                window_size=window_size,
                mlp_ratio=mlp_ratio,
                qkv_bias=qkv_bias,
                qk_scale=qk_scale,
                drop=drop_rate,
                attn_drop=attn_drop_rate,
                drop_path=dpr[sum(depths[:i_layer]):sum(depths[:i_layer + 1])],
                drop_path_rate=drop_path_rate,
                norm_layer=norm_layer,
                downsample=PatchMerging if i_layer < self.num_layers - 1 else None,
                use_checkpoint=use_checkpoint)
            self.layers.append(layer)

        # build decoder layers
        self.layers_up = nn.ModuleList()
        self.concat_back_dim = nn.ModuleList()
        for i_layer in range(self.num_layers):
            concat_linear = nn.Linear(2 * int(embed_dim * 2 ** (self.num_layers - 1 - i_layer)),
                                      int(embed_dim * 2 ** (
                                              self.num_layers - 1 - i_layer)),
                                      bias=False) if i_layer > 0 else nn.Identity()
            if i_layer == 0:
                layer_up = PatchExpand(
                    input_resolution=(patches_resolution[0] // (2 ** (self.num_layers - 1 - i_layer)),
                                      patches_resolution[1] // (2 ** (self.num_layers - 1 - i_layer)),
                                      patches_resolution[2] // (2 ** (self.num_layers - 1 - i_layer))),
                    dim=int(embed_dim * 2 ** (self.num_layers - 1 - i_layer)), dim_scale=2, norm_layer=norm_layer)
            else:
                layer_up = BasicLayer_up(
                    dim=int(embed_dim * 2 ** (self.num_layers - 1 - i_layer)),
                    input_resolution=(patches_resolution[0] // (2 ** (self.num_layers - 1 - i_layer)),
                                      patches_resolution[1] // (2 ** (self.num_layers - 1 - i_layer)),
                                      patches_resolution[2] // (2 ** (self.num_layers - 1 - i_layer))),
                    depth=depths[(self.num_layers - 1 - i_layer)],
                    num_heads=num_heads[(self.num_layers - 1 - i_layer)],
                    window_size=window_size,
                    mlp_ratio=mlp_ratio,
                    qkv_bias=qkv_bias,
                    qk_scale=qk_scale,
                    drop=drop_rate,
                    attn_drop=attn_drop_rate,
                    drop_path=dpr[sum(depths[:(self.num_layers - 1 - i_layer)]):sum(
                        depths[:(self.num_layers - 1 - i_layer) + 1])],
                    norm_layer=norm_layer,
                    upsample=PatchExpand if (i_layer < self.num_layers - 1) else None,
                    use_checkpoint=use_checkpoint)

                self.layers_up.append(layer_up)
                self.concat_back_dim.append(concat_linear)

        self.norm = norm_layer(self.num_features)
        self.norm_up = norm_layer(self.embed_dim)

        if self.final_upsample == "expand_first":
            print("---final upsample expand_first---")
            self.up = FinalPatchExpand_X4(input_resolution=(
                img_size[0] // patch_size[0], img_size[1] // patch_size[1], img_size[2] // patch_size[2]),
                dim_scale=4, dim=embed_dim)
            self.output = nn.Conv3d(in_channels=embed_dim, out_channels=self.num_classes, kernel_size=1, bias=False)
            self.softmax = nn.Sigmoid()

        self._freeze_stages()

    @torch.jit.ignore
    def no_weight_decay(self):
        return {'absolute_pos_embed'}

    @torch.jit.ignore
    def no_weight_decay_keywords(self):
        return {'relative_position_bias_table'}

    # Encoder and Bottleneck
    def forward_features(self, x):
        x = self.patch_embed(x)
        x = self.pos_drop(x)
        x_downsample = []
        v_values_1 = []
        k_values_1 = []
        q_values_1 = []
        v_values_2 = []
        k_values_2 = []
        q_values_2 = []

        for i, layer in enumerate(self.layers):
            x_downsample.append(x)
```

```
            x, v1, k1, q1, v2, k2, q2 = layer(x, i)
            v_values_1.append(v1)
            k_values_1.append(k1)
            q_values_1.append(q1)
            v_values_2.append(v2)
            k_values_2.append(k2)
            q_values_2.append(q2)

        x = rearrange(x, 'n c d h w -> n d h w c')
        x = self.norm(x)
        x = rearrange(x, 'n d h w c -> n c d h w')

        return x, x_downsample, v_values_1, k_values_1, q_values_1, v_values_2, k_values_2, q_values_2

    # Dencoder and Skip connection
    def forward_up_features(self, x, x_downsample, v_values_1, k_values_1, q_values_1, v_values_2, k_values_2,
                            q_values_2):

        for inx, layer_up in enumerate(self.layers_up):
            if inx == 0:
                x = layer_up(x)
            else:
                x = torch.cat([x, x_downsample[3 - inx]], 1)
                B, C, D, H, W = x.shape
                x = x.flatten(2).transpose(1, 2)
                x = self.concat_back_dim[inx](x)
                _, _, C = x.shape
                x = x.view(B, D, H, W, C)

                x = x.permute(0, 4, 1, 2, 3)
                x = layer_up(x, v_values_1[3 - inx], k_values_1[3 - inx], q_values_1[3 - inx], v_values_2[3 - inx],
                             k_values_2[3 - inx], q_values_2[3 - inx])

        x = self.norm_up(x)

        return x

    def up_x4(self, x):
        D, H, W = self.patches_resolution
        B, _, _, _, C = x.shape

        if self.final_upsample == "expand_first":
            x = self.up(x)
            x = x.view(B, 4 * D, 4 * H, 4 * W, -1)
            x = x.permute(0, 4, 1, 2, 3)  # B,C,D,H,W
            x = self.output(x)
            x = self.softmax(x)
        return x

    def _freeze_stages(self):
        if self.frozen_stages >= 0:
            self.patch_embed.eval()
            for param in self.patch_embed.parameters():
                param.requires_grad = False

        if self.frozen_stages >= 1:
            self.pos_drop.eval()
            for i in range(0, self.frozen_stages):
                m = self.layers[i]
                m.eval()
                for param in m.parameters():
                    param.requires_grad = False

    def inflate_weights(self):
        """Inflate the swin2d parameters to swin3d.

        The differences between swin3d and swin2d mainly lie in an extra
        axis. To utilize the pretrained parameters in 2d model,
        the weight of swin2d models should be inflated to fit in the shapes of
        the 3d counterpart.

        Args:
            logger (logging.Logger): The logger used to print
                debugging infomation.
        """
        checkpoint = torch.load(self.pretrained, map_location='cpu')
        state_dict = checkpoint['model']

        # delete relative_position_index since we always re-init it
        relative_position_index_keys = [k for k in state_dict.keys() if "relative_position_index" in k]
        for k in relative_position_index_keys:
            del state_dict[k]

        # delete attn_mask since we always re-init it
        attn_mask_keys = [k for k in state_dict.keys() if "attn_mask" in k]
        for k in attn_mask_keys:
            del state_dict[k]

        state_dict['patch_embed.proj.weight'] = state_dict['patch_embed.proj.weight'].unsqueeze(2).repeat(1, 1,
                                                                                                          self.patch_size[
                                                                                                              0], 1,
                                                                                                          1) / \
                                                self.patch_size[0]

        # bicubic interpolate relative_position_bias_table if not match
        relative_position_bias_table_keys = [k for k in state_dict.keys() if "relative_position_bias_table" in k]
        for k in relative_position_bias_table_keys:
            relative_position_bias_table_pretrained = state_dict[k]
            relative_position_bias_table_current = self.state_dict()[k]
            L1, nH1 = relative_position_bias_table_pretrained.size()
            L2, nH2 = relative_position_bias_table_current.size()
            L2 = (2 * self.window_size[1] - 1) * (2 * self.window_size[2] - 1)
            wd = self.window_size[0]

            if nH1 != nH2:
                print(f"Error in loading {k}, passing")
            else:
                if L1 != L2:
                    S1 = int(L1 ** 0.5)
                    relative_position_bias_table_pretrained_resized = torch.nn.functional.interpolate(
                        relative_position_bias_table_pretrained.permute(1, 0).view(1, nH1, S1, S1),
                        size=(2 * self.window_size[1] - 1, 2 * self.window_size[2] - 1),
                        mode='bicubic')
                    relative_position_bias_table_pretrained = relative_position_bias_table_pretrained_resized.view(nH2,
                                                                                                                   L2).permute(
                        1, 0)
            state_dict[k] = relative_position_bias_table_pretrained.repeat(2 * wd - 1, 1)

        msg = self.load_state_dict(state_dict, strict=False)
        print(msg)
        print(f"=> loaded successfully '{self.pretrained}'")
        del checkpoint
        torch.cuda.empty_cache()
```

```python
    def init_weights(self, pretrained=None):
        """Initialize the weights in backbone.

        Args:
            pretrained (str, optional): Path to pre-trained weights.
                Defaults to None.
        """

        def _init_weights(m):
            if isinstance(m, nn.Linear):
                trunc_normal_(m.weight, std=.02)
                if isinstance(m, nn.Linear) and m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.LayerNorm):
                nn.init.constant_(m.bias, 0)
                nn.init.constant_(m.weight, 1.0)

        if pretrained:
            self.pretrained = pretrained
        if isinstance(self.pretrained, str):
            self.apply(_init_weights)

            print(f'load model from: {self.pretrained}')

            if self.pretrained2d:
                # Inflate 2D model into 3D model.
                self.inflate_weights()
            else:
                # Directly load 3D model.
                pass
                # load_checkpoint(self, self.pretrained, strict=False)
        elif self.pretrained is None:
            self.apply(_init_weights)
        else:
            raise TypeError('pretrained must be a str or None')

    def forward(self, x):
        x, x_downsample, v_values_1, k_values_1, q_values_1, v_values_2, k_values_2, q_values_2 = self.forward_features(
            x)

        x = self.forward_up_features(x, x_downsample, v_values_1, k_values_1, q_values_1, v_values_2, k_values_2,
                                     q_values_2)
        x = self.up_x4(x)

        return x


if __name__ == '__main__':

    swin_unet = SwinTransformerSys3D(img_size=(128, 128, 128),
                                     patch_size=(4, 4, 4),
                                     in_chans=1,
                                     num_classes=2,
                                     )
    x1 = torch.randn(1, 4, 128, 128, 128)

    x2 = torch.randn(1, 4, 128, 128, 128)

    y = swin_unet(x1)
    embed()
```

H:\git\Repo2PDF\dist\repo2pdfAPP\SparrowLink\modify_key_in_config.py

```
SparrowLink/modify_key_in_config.py
import yaml
import argparse
import pathlib

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--config_path", type=str, default=None)

    args = parser.parse_args()
    assert args.config_path is not None, "config_path is None"
    name = pathlib.Path(args.config_path).name
    parent = pathlib.Path(args.config_path).parent
    with open(args.config_path, 'r') as f:
        config = yaml.load(f, Loader=yaml.FullLoader)
    # full_key = ['I_M', 'I_A', 'CS_M', 'CS_A', 'CS_DL']
    # config['infer']['key'] = ['I_M', 'I_A', 'CS_M', 'CS_A', 'CS_DL']
    # config['train']['key'] = ['I_M', 'I_A', 'CS_M', 'CS_A', 'CS_DL']
    # config['model']['in_channels'] = len(config['infer']['key'])
    # config['model']['model_parameter']['num_input_channels'] = len(config['infer']['key'])
    # with open(parent / name.replace(".yaml", f"_new_0.yaml"), 'w') as f:
    #     yaml.dump(config, f, encoding='utf-8', allow_unicode=True)
    #
    # config['infer']['key'] = ['I_M', 'CS_M', 'CS_A', 'CS_DL']
    # config['train']['key'] = ['I_M', 'CS_M', 'CS_A', 'CS_DL']
    # config['model']['in_channels'] = len(config['infer']['key'])
    # config['model']['model_parameter']['num_input_channels'] = len(config['infer']['key'])
    # with open(parent / name.replace(".yaml", f"_new_1.yaml"), 'w') as f:
    #     yaml.dump(config, f, encoding='utf-8', allow_unicode=True)
    #
    # config['infer']['key'] = ['I_M', 'I_A', 'CS_A', 'CS_DL']
    # config['train']['key'] = ['I_M', 'I_A', 'CS_A', 'CS_DL']
    # config['model']['in_channels'] = len(config['infer']['key'])
    # config['model']['model_parameter']['num_input_channels'] = len(config['infer']['key'])
    # with open(parent / name.replace(".yaml", f"_new_2.yaml"), 'w') as f:
    #     yaml.dump(config, f, encoding='utf-8', allow_unicode=True)
    #
    # config['infer']['key'] = ['I_M', 'I_A', 'CS_M', 'CS_DL']
    # config['train']['key'] = ['I_M', 'I_A', 'CS_M', 'CS_DL']
    # config['model']['in_channels'] = len(config['infer']['key'])
    # config['model']['model_parameter']['num_input_channels'] = len(config['infer']['key'])
    # with open(parent / name.replace(".yaml", f"_new_3.yaml"), 'w') as f:
    #     yaml.dump(config, f, encoding='utf-8', allow_unicode=True)

    # config['infer']['key'] = ['I_M', 'I_A', 'CS_M', 'CS_A']
    # config['train']['key'] = ['I_M', 'I_A', 'CS_M', 'CS_A']
    # config['model']['in_channels'] = len(config['infer']['key'])
    # config['model']['model_parameter']['num_input_channels'] = len(config['infer']['key'])
    # with open(parent / name.replace(".yaml", f"_new_4.yaml"), 'w') as f:
    #     yaml.dump(config, f, encoding='utf-8', allow_unicode=True)

    # config['infer']['key'] = ['I_M']
    # config['train']['key'] = ['I_M']
    # config['model']['in_channels'] = len(config['infer']['key'])
    # config['model']['model_parameter']['num_input_channels'] = len(config['infer']['key'])
    # with open(parent / name.replace(".yaml", f"_new_I_M.yaml"), 'w') as f:
    #     yaml.dump(config, f, encoding='utf-8', allow_unicode=True)

    config['infer']['key'] = ['I_M', 'CS_M', 'CS_DL']
    config['train']['key'] = ['I_M', 'CS_M', 'CS_DL']
    config['model']['in_channels'] = len(config['infer']['key'])
    config['model']['model_parameter']['num_input_channels'] = len(config['infer']['key'])
    with open(parent / name.replace(".yaml", f"_new_no_a.yaml"), 'w') as f:
        yaml.dump(config, f, encoding='utf-8', allow_unicode=True)

    config['infer']['key'] = ['I_M', 'I_A', 'CS_A', 'CS_DL']
    config['train']['key'] = ['I_M', 'CS_M', 'CS_A', 'CS_DL']
    config['model']['in_channels'] = len(config['infer']['key'])
    config['model']['model_parameter']['num_input_channels'] = len(config['infer']['key'])
    with open(parent / name.replace(".yaml", f"_new_MMAD.yaml"), 'w') as f:
        yaml.dump(config, f, encoding='utf-8', allow_unicode=True)
```

H:\git\Repo2PDF\dist\repo2pdfAPP\SparrowLink\post_processing\fracture_detection.py

```
H:\git\Repo2PDF\dist\repo2pdfAPP\SparrowLink\post_processing\fracture_detection.py
Empty File
```

```python
SparrowLink/post_processing/move_file.py
import pathlib
import argparse
import shutil
from tqdm import tqdm

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('--data_path',
                        type=str,
                        help='move discontinuity detection file to sphere or cube file for easiler dataloading',
                        default='test')
    parser.add_argument('--save_path',
                        type=str,
                        help='save path',
                        default='test')
    parser.add_argument('--data_postfix',
                        type=str,
                        help='sphere or cube',
                        default='')
    parser.add_argument('--hierarchical',
                        type=str,
                        help='sphere or cube',
                        default='')
    parser.add_argument('--save_postfix',
                        type=str,
                        help='discontinuity detection file',
                        default='')
    parser.add_argument('--separate folder',
                        action="store_true",
                        default=False,
                        )

    args = parser.parse_args()
    data_path = pathlib.Path(args.data_path)
    save_path = pathlib.Path(args.save_path)
    save_path.mkdir(exist_ok=True, parents=True)
    data_list = list(data_path.glob(f'*{args.hierarchical}{args.data_postfix}.nii.gz'))
    assert len(data list) > 0, f"no file found in {data_path} {args.data_postfix}"
    print(f"move file for better viewing from \n"
          f"{args.data_path} to \n"
          f"{args.save_path}, \n"
          f"finding={args.hierarchical}{args.data_postfix}, save_postfix={args.save_postfix}",)
    pbar = tqdm(total=len(data list))
    pbar.set_description('copy file')
    for file in data_list:
        file_name = file.name
        new_file_name = file_name.replace(f'{args.data_postfix}.nii.gz', f'{args.save_postfix}.nii.gz')
        if args.separate_folder:
            patient_id = file_name[:9]
            (save_path / patient_id).mkdir(exist_ok=True, parents=True)
            new_file = save_path / patient_id / new_file_name
        else:
            new_file = save_path / new_file_name
        shutil.copy(file, new_file)
        pbar.update()
```

```
0  SparrowLink/post_processing/select_two_region.py
1  import cc3d
2  import numpy as np
3
4  def sort_region(x, num=2):
5      """x:3D, select six the most large region"""
6      max_label = x.max()
7      sum_list = [(x == index).sum() for index in range(1, int(max_label.item())+1)]
8      # print(sum_list)
9      # sort sum_list, return index, from large to small
10     index_list = np.argsort(sum_list)[::-1]
11     region_reserved = x == (index_list[0] + 1)
12     for index in index_list[1:num]:
13         region_reserved = region_reserved | (x == (index+1))
14     return np.array(region_reserved, dtype=bool)
15
16
17 def select_two_biggest_connected_region(region, num=2):
18     region_mask = cc3d.connected_components(region > 0, connectivity=6)
19     region_two = region * sort_region(region_mask, num=num)
20     return region_two
```

0 SparrowLink/README.md
1 # SparrowLink: a two-Stage, two-Phase And special-Region-aware segmentation Optimization Workflow for discontinuity-Link
2 The name is given by chatgpt3.5. The framework is based on MONAI and nnUnet.
3 Still in modifying.
4 This method can work when only one phase is available.

```python
SparrowLink/registration/basic_registration.py
import pathlib
import ants
import time
import argparse
from multiprocessing import Pool


def Reg(fixed_path, moving_path, apply_path, save_path, save_path_apply, mode='SyNRA', reg_interpolator='linear',
        apply_interpolator='nearestNeighbor'):
    tic = time.time()
    fixed = ants.image_read(str(fixed_path))
    moving = ants.image_read(str(moving_path))
    apply = ants.image_read(str(apply_path))
    reg = ants.registration(fixed=fixed, moving=moving, type_of_transform=mode)
    moving_reg = ants.apply_transforms(fixed=fixed, moving=moving, transformlist=reg['fwdtransforms'],
                                       interpolator=reg_interpolator)
    apply_reg = ants.apply_transforms(fixed=fixed, moving=apply, transformlist=reg['fwdtransforms'],
                                      interpolator=apply_interpolator)

    ants.image_write(moving_reg, str(save_path / moving_path.name))
    ants.image_write(apply_reg, str(save_path_apply / moving_path.name))

    return {"name": moving_path.name, "time": time.time()-tic}


def update(pbar, result):
    pbar.update()
    # print(result)


def call_fun(result):
    print(result)


def errorback(err):
    print(err)


if __name__ == "__main__":
    arg = argparse.ArgumentParser()
    arg.add_argument('--reg_algorithm', type=str, default='SyNRA')  # SyNRA, Rigid
    arg.add_argument('--fixed_dir', type=str,
                     default="/public/home/v-xiongxx/Graduate_project/Cardio_vessel_segmentaion_based_on_monai/data/ZhangX/CT_Img/")
    arg.add_argument('--moving_dir', type=str,
                     default="/public/home/v-xiongxx/Graduate_project/Cardio_vessel_segmentaion_based_on_monai/data/ZhangX/CTA_Img/")
    arg.add_argument('--apply_dir', type=str,
                     default="/public/home/v-xiongxx/Graduate_project/Cardio_vessel_segmentaion_based_on_monai/data/ZhangX/CTA_infer/")
    arg.add_argument('--save_dir', type=str,
                     default="/public/home/v-xiongxx/Graduate_project/Cardio_vessel_segmentaion_based_on_monai/data/ZhangX/CTA_Img_reg/")
    arg.add_argument('--save_dir apply', type=str,
                     default="/public/home/v-xiongxx/Graduate_project/Cardio_vessel_segmentaion_based_on_monai/data/ZhangX/CTA_infer_reg/")


    args = arg.parse_args()
    fixed_dir = pathlib.Path(args.fixed_dir)
    moving_dir = pathlib.Path(args.moving_dir)
    apply_dir = pathlib.Path(args.apply_dir)
    save_dir = pathlib.Path(args.save_dir)
    save_dir_apply = pathlib.Path(args.save_dir_apply)
    save_dir.mkdir(exist_ok=True)
    save_dir_apply.mkdir(exist_ok=True)
    print(f"---------{args.reg_algorithm}_registration---------")
    pool = Pool(4)
    for file in fixed_dir.glob('*.nii.gz'):
        name = file.name
        moving_path = moving_dir / name.replace('CT', 'CTA')
        apply_path = apply_dir / name.replace('CT', 'CTA')
        pool.apply_async(Reg,
                         args=(file, moving_path, apply_path, save_dir, save_dir_apply, args.reg_algorithm),
                         callback=call_fun,
                         error_callback=errorback)

    pool.close()
    pool.join()
```

```python
SparrowLink/registration/label_registration.py
import pathlib
import ants
import time
import argparse
from multiprocessing import Pool


def Reg(target_path, moving_path, target_path_1, moving_path_1, save_path, save_path_1, mode='SyNRA', t=1):
    tic = time.time()
    target = ants.image_read(str(target_path))
    moving = ants.image_read(str(moving_path))
    target_1 = ants.image_read(str(target_path_1))
    moving_1 = ants.image_read(str(moving_path_1))
    moving_loop = moving
    moving_1_loop = moving_1
    target_loop = target
    target_1_loop = target_1
    for i in range(t):
        reg = ants.registration(fixed=target, moving=moving_loop, type_of_transform=mode)
        moving_loop = ants.apply_transforms(fixed=target, moving=moving_loop, transformlist=reg['fwdtransforms'], interpolator='nearestNeighbor')
        moving_1_loop = ants.apply_transforms(fixed=target_1, moving=moving_1_loop, transformlist=reg['fwdtransforms'])
        target_loop = ants.apply_transforms(fixed=moving, moving=target_loop, transformlist=reg['invtransforms'], interpolator='nearestNeighbor')
        target_1_loop = ants.apply_transforms(fixed=moving_1_loop, moving=target_1_loop, transformlist=reg['invtransforms'])


    ants.image_write(moving_loop, str(save_path))
    ants.image_write(moving_1_loop, str(save_path_1))

    save_path_2 = save_path.parent.parent / save_path.parent.name.replace('auxiliary', 'main')
    save_path_2.mkdir(exist_ok=True, parents=True) if not save_path_2.exists() else None
    save_path_2 = save_path_2 / save_path.name
    save_path_3 = save_path_1.parent.parent / save_path_1.parent.name.replace('auxiliary', 'main')
    save_path_3.mkdir(exist_ok=True, parents=True)
    save_path_3 = save_path_3 / save_path_1.name

    ants.image_write(target_loop, str(save_path_2))
    ants.image_write(target_1_loop, str(save_path_3))

    return {"name": moving_path.name, "time": time.time()-tic}


def update(pbar, result):
    pbar.update()
    # print(result)


def call_fun(result):
    print(result)


def errorback(err):
    print(err)


if __name__ == "__main__":
    arg = argparse.ArgumentParser()
    arg.add_argument('--reg_algorithm', type=str, default='SyNRA')  # SyNRA, Rigid
    arg.add_argument('--mode', type=str, help='used in save name', default='test')
    arg.add_argument('--time', type=int, help='time for registration loop',default=1)
    arg.add_argument('--net', type=str, default="used in save name")
    arg.add_argument('--save_root', type=str, help='save path for img and mask after registration',
                     default="/public/home/v-xiongxx/Graduate_project/"
                             "Cardio_vessel_segmentaion_based_on_monai/"
                             "experiments/Graduate_project/multi_phase/pretrain")
    arg.add_argument('--main_mask_path', type=str, default=None)
    arg.add_argument('--auxiliary_mask_path', type=str, default=None)
    arg.add_argument('--main_img_path', type=str, default=None)
    arg.add_argument('--auxiliary_img_path', type=str, default=None)


    args = arg.parse_args()
    main_label_path = pathlib.Path(args.main_mask_path)
    auxiliary_label_path = pathlib.Path(args.auxiliary_mask_path)
    main_img_path = pathlib.Path(args.main_img_path)
    auxiliary_img_path = pathlib.Path(args.auxiliary_img_path)
    save_auxiliary_label_regitration = pathlib.Path(args.save_root) / f"auxiliary_{args.mode}_infer_{args.reg_algorithm}_reg_{args.time}"
    save_auxiliary_img_regitration = pathlib.Path(args.save_root) / f"auxiliary_{args.mode}_img_{args.reg_algorithm}_reg_{args.time}"
    save_auxiliary_img_regitration.mkdir(exist_ok=True, parents=True)
    save_auxiliary_label_regitration.mkdir(exist_ok=True, parents=True)
    # find the corresponding file in segment_syst
    main_list = list(main_img_path.glob('*.nii.gz'))
    # sort the file name

    main_list.sort(key=lambda x: str(x.stem))
    # q: what is x.stem?

    pool = Pool(4)
    print(f"---------{args.mode}_registration---------")
    for i, img in enumerate(main_list):
        img_name = img.name

        main_img = img
        main_label = main_label_path / img_name
        auxiliary_img = auxiliary_img_path / img_name
        auxiliary_label = auxiliary_label_path / img_name
        save_auxiliary_label = save_auxiliary_label_regitration / img_name
        save_auxiliary_img = save_auxiliary_img_regitration / img_name

        assert main_label.exists(), f"{str(main_label)} not exist"
        assert main_img.exists(), f"{str(main_img)} not exist"
        assert auxiliary_img.exists(), f"{str(auxiliary_img)} not exist"
        assert auxiliary_label.exists(), f"{str(auxiliary_label)} not exist"
        # Reg(target_path=main_label,
        #     moving_path=auxiliary_label,
        #     target_path_1=main_img,
        #     moving_path_1=auxiliary_img,
        #     save_path=save_auxiliary_img,
        #     save_path_1=save_auxiliary_label,
        #     mode=args.mode)
        pool.apply_async(Reg,
                         args=(main_label, auxiliary_label, main_img, auxiliary_img, save_auxiliary_label, save_auxiliary_img, args.reg_algorithm, args.time),
                         callback=call_fun,
                         error_callback=errorback)

    pool.close()
    pool.join()
```

```python
SparrowLink/second_stage_main.py
from monai.utils import first, set_determinism
from monai.data import CacheDataset, DataLoader, Dataset, decollate_batch, ITKReader, PersistentDataset
from monai.config import print_config
from monai.metrics.meandice import DiceMetric
import torch
import matplotlib.pyplot as plt
import os
import argparse
import monai
import logging
from utils.Config import Config
import sys
import time
from data.loader import prepare_datalist, prepare_datalist_with_file, \
    prepare_image_list, save_json, write_data_reference, load_json, prepare_main_auxiliary_with_img_datalist_with_file
from transform.utils import get_transform, get_multi_phase_transform_with_image
from utils.test import cardio_vessel_segmentation_test, cardio_vessel_segmentation_multi_phase_with_image_test
from utils.trainer import cardio_vessel_segmentation_multi_phase_with_image_train
from utils.inferer import cardio_vessel_segmentation_infer, cardio_vessel_segmentation_multi_phase_with_image_infer
import pathlib
import torch.nn as nn
from torch.distributions.normal import Normal
# print_config()


torch.multiprocessing.set_sharing_strategy('file_system')


def load_weight_from_coarse_segmentation(in_channels, model, weight_path, net_architecture):
    """
    it will change
    """
    in_channels = model.in_channels if hasattr(model, "in_channels") else in_channels
    if net_architecture == 'UNet':
        # d = torch.load(
        #     './experiments/Graduate_project/multi_phase/pretrain/ResUnet/checkpoint/best_metric_model.pth')
        d = torch.load(weight_path)
        shape = list(d['model.0.conv.unit0.conv.weight'].shape)
        shape[1] = in_channels - 1
        if d.get('model.0.residual.weight') is not None:
            noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['model.0.residual.weight'].device)
            d['model.0.residual.weight'] = torch.cat((d['model.0.residual.weight'], noise), dim=1)
        noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['model.0.conv.unit0.conv.weight'].device)
        d['model.0.conv.unit0.conv.weight'] = torch.cat((d['model.0.conv.unit0.conv.weight'], noise), dim=1)
        model.load_state_dict(d)
        return model

    elif net_architecture == "CSNet3D":
        d1 = torch.load(weight_path)
        d2 = model.state_dict()
        print("------------------------------load checkpoint and modify-------------------------------------------")
        for k2, v2 in d2.items():
            if d1[k2].shape != v2.shape:
                assert d1[k2].shape[1] != v2.shape[1] and len(d1[k2].shape) > 1 and len(v2.shape) > 1
                noise_shape = torch.tensor(v2.shape)
                noise_shape[1] = v2.shape[1] - d1[k2].shape[1]
                noise = nn.Parameter(Normal(0, 1e-7).sample(noise_shape)).to(d1[k2].device)
                print(
                    f"key: {k2}, shape1: {d1[k2].shape}, mean1: {d1[k2].mean()}, shape2: {v2.shape}, mean2: {v2.mean()}")
                d1[k2] = torch.cat((d1[k2], noise), dim=1)
        model.load_state_dict(d1)
        return model

    elif net_architecture == "SkipDenseUNet":
        d1 = torch.load(weight_path)
        d2 = model.state_dict()
        print("------------------------------load checkpoint and modify-------------------------------------------")
        for k2, v2 in d2.items():
            if d1[k2].shape != v2.shape:
                assert d1[k2].shape[1] != v2.shape[1] and len(d1[k2].shape) > 1 and len(v2.shape) > 1
                noise_shape = torch.tensor(v2.shape)
                noise_shape[1] = v2.shape[1] - d1[k2].shape[1]
                noise = nn.Parameter(Normal(0, 1e-10).sample(noise_shape)).to(d1[k2].device)
                print(
                    f"key: {k2}, shape1: {d1[k2].shape}, mean1: {d1[k2].mean()}, shape2: {v2.shape}, mean2: {v2.mean()}")
                d1[k2] = torch.cat((d1[k2][:, :1, ...], noise, d1[k2][:, 1:, ...]), dim=1)
        model.load_state_dict(d1)
        return model
    elif net_architecture == "SwinTransformerSys3D":
        d = torch.load(weight_path)
        shape = list(d['patch_embed.proj.weight'].shape)
        shape[1] = in_channels - 1

        noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['patch_embed.proj.weight'].device)
        d['patch_embed.proj.weight'] = torch.cat((d['patch_embed.proj.weight'], noise), dim=1)
        model.load_state_dict(d)
        return model

    elif net_architecture == "nnunetv2":
        all_information = torch.load(weight_path)
        d = all_information['network weights']
        shape = list(d['encoder.stages.0.0.convs.0.conv.weight'].shape)
        shape[1] = in_channels - 1
        noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['encoder.stages.0.0.convs.0.conv.weight'].device)
        d['encoder.stages.0.0.convs.0.conv.weight'] = torch.cat((d['encoder.stages.0.0.convs.0.conv.weight'], noise), dim=1)
        noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['encoder.stages.0.0.convs.0.all_modules.0.weight'].device)
        d['encoder.stages.0.0.convs.0.all_modules.0.weight'] = torch.cat((d['encoder.stages.0.0.convs.0.all_modules.0.weight'], noise), dim=1)
        noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['decoder.encoder.stages.0.0.convs.0.conv.weight'].device)
        d['decoder.encoder.stages.0.0.convs.0.conv.weight'] = torch.cat((d['decoder.encoder.stages.0.0.convs.0.conv.weight'], noise), dim=1)
        noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['decoder.encoder.stages.0.0.convs.0.all_modules.0.weight'].device)
        d['decoder.encoder.stages.0.0.convs.0.all_modules.0.weight'] = torch.cat((d['decoder.encoder.stages.0.0.convs.0.all_modules.0.weight'], noise), dim=1)
        model.load_state_dict(d)
        return model

    else:
        raise NotImplementedError(f"not support {net_architecture} yet!")


if __name__ == "__main__":

    parser = argparse.ArgumentParser(description="cardio vessel segmentation")
    parser.add_argument("--configs", dest="cfg",
                        help="The configs file.",
                        default='./configs/heart_server.yaml',
                        type=str)
    parser.add_argument('--iters', dest='iters', help='Iterations in training.', type=int, default=None)
    parser.add_argument('--batch_size', dest='batch_size', help='Mini batch size of one gpu or cpu.', type=int, default=None)
    parser.add_argument('--learning_rate', dest='learning_rate', help='Learning rate', type=float, default=None)
    parser.add_argument('--seed', dest='seed', help='seed', type=float, default=0)
    parser.add_argument('--mode', dest='mode', help='train, infer or both', type=str, default=None)

    parser.add_argument('--experiments_path', dest='experiments_path',
                        help='experiments_path, all output and checkpoint are saved here.', type=str, default=None)

    parser.add_argument('--pretrain weight path', help='pretrain weight for training, testing of inferring', type=str, default=None)
```

```
    parser.add_argument('--img_path', help='image path', type=str, default=None)

    parser.add_argument('--label_path', help='label path', type=str, default=None)
    parser.add_argument('--output_path', help='save path', type=str, default=None)
    parser.add_argument('--persist_path', help='persist path, the path used to save persist data in persist loader',
                        type=str, default=None)
    parser.add_argument('--val_set', help='val set, a txt path to select val data, only used if mode is train',
                        type=str, default=None)
    parser.add_argument('--train_set', help='train set, a txt path to select train data, only used if mode is train',
                        type=str, default=None)
    parser.add_argument('--dataset_information', help='dataset information',
                        type=str, default=None)

    # ---------------------------------------- multi information ---------------------------------------- #
    parser.add_argument('--CS_W',
                        help='pretrain weight from coarse segmentation, testing of inferring', type=str, default=None)
    parser.add_argument('--CS_M', help='coarse segmentation in main phase', type=str, default=None)
    parser.add_argument('--CS_A', help='coarse segmentation in auxiliary phase', type=str, default=None)
    parser.add_argument('--CS_DL', help='discontinuity label generated in main coarse segmentation', type=str,
                        default=None)
    parser.add_argument('--CS_DLGT', help='discontinuity label generated in auxiliary coarse segmentation', type=str,
                        default=None)
    parser.add_argument('--I_M', help='main image', type=str, default=None)
    parser.add_argument('--I_A', help='auxiliary image', type=str, default=None)
    parser.add_argument('--select_file', help='use discontinuity label to determine what data are used', type=str,
                        default=None)
    parser.add_argument('--delete_persist', help='pretrain weight from coarse segmentation, testing of inferring', type=str,)
    parser.add_argument('--view', action='store_true', default=False, help='delete persist cache, because it is too large')
    # ---------------------------------------- multi information ---------------------------------------- #

    args = parser.parse_args()
    monai.config.print_config()

    logging.basicConfig(stream=sys.stdout, level=logging.INFO)
    cfg = Config(
        args.cfg,
        learning_rate=args.learning_rate,
        iters=args.iters,
        batch_size=args.batch_size,
        seed=args.seed,
        mode=args.mode,
        img_path=args.img_path,
        I_M=args.I_M,
        I_A=args.I_A,
        CS_M=args.CS_M,
        CS_A=args.CS_A,
        CS_DL=args.CS_DL,
        CS_DLGT=args.CS_DLGT,
        CS_W=args.CS_W,
        select_file=args.select_file,
        label_path=args.label_path,
        output_path=args.output_path,
        persist_path=args.persist_path,
        val_set=args.val_set,
        train_set=args.train_set,
        experiments_path=args.experiments_path,
        pretrain_weight_path=args.pretrain_weight_path,
        dataset_information=args.dataset_information
        )
    set_determinism(seed=cfg.seed)

    if cfg.dic['mode'] == 'train':

        # --------------------------- built transform sequence ---------------------------- #
        # ------------- take a look at val_transform on dataset and save a case ------------ #
        cfg.creat_training_require()
        load_weight_from_coarse_segmentation(in_channels=cfg.dic['model'].get("in_channels"), model=cfg.model,
                                             net_architecture=cfg.dic['model']['name'], weight_path=cfg.CS_W)
        train_transforms, val_transforms, save_transform = get_multi_phase_transform_with_image(cfg.dic['transform'])
        if cfg.dic["train"]["loader"].get("file_path"):
            files = [load_json(cfg.dic["train"]["loader"]["file_path"])]
        elif cfg.dic["train"]["loader"].get("val_set"):
            val_files = prepare_main_auxiliary_with_img_datalist_with_file(main_file=cfg.CS_M,
                                                                           auxiliary_file=cfg.CS_A,
                                                                           main_img_file=cfg.I_M,
                                                                           auxiliary_img_file=cfg.I_A,
                                                                           label_file=cfg.train_label_path,
                                                                           broken_file=cfg.CS_DL,
                                                                           broken_gt_file=cfg.CS_DLGT,
                                                                           img_name=cfg.val_set,
                                                                           select_file=cfg.select_file, )

            train_files = prepare_main_auxiliary_with_img_datalist_with_file(main_file=cfg.CS_M,
                                                                             auxiliary_file=cfg.CS_A,
                                                                             main_img_file=cfg.I_M,
                                                                             auxiliary_img_file=cfg.I_A,
                                                                             label_file=cfg.train_label_path,
                                                                             broken_file=cfg.CS_DL,
                                                                             broken_gt_file=cfg.CS_DLGT,
                                                                             img_name=cfg.train_set,
                                                                             select_file=cfg.select_file, )
            files = [{"train_files": train_files, "val_files": val_files}]

        else:
            raise ValueError("please provide a file path or val set and train set")
        # check_transform_in_dataloader(val_files=files[0]['val_files'], val_transforms=try_transforms)

        # --------------------------- create loss function ---------------------------- #
        # ------------- you can crate your own loss or metric here amd replace cfg ------------ #

        # dice_metric = DiceMetric(include_background=False, reduction="mean")
        # loss_function = DiceLoss(to_onehot_y=True, softmax=True)

        # -------------- training ---------------- #
        metric_record = []
        experiment_path = os.path.join(cfg.dic['experiments_path'], time.strftime("%d_%m_%Y_%H_%M_%S"))\
            if cfg.dic.get('time_name', None) else cfg.dic['experiments_path']

        if not os.path.exists(experiment_path):
            os.makedirs(experiment_path)
        for i in range(len(files)):
            if cfg.dic['train']['loader'].get('split_mode') == "five_fold":
                experiment_path_fold = os.path.join(experiment_path, f"{i}_fold")
                if not os.path.exists(experiment_path_fold):
                    os.makedirs(experiment_path_fold)
            else:
                experiment_path_fold = experiment_path
            write_data_reference(files[i], experiment_path_fold)
            save_json(files[i], os.path.join(experiment_path_fold, 'files.txt'))
            # -------------- Create Model, Loss, Optimizer in Config-------------- #
            # ----------------------- using config ------------------------------ #
            cfg.save_config(os.path.join(experiment_path_fold, 'configs.yaml'))
            if cfg.dic['train']['loader'].get('split_mode') == "five_fold":
                print(f"----------------fold{i} start!-----------------")
            else:
                print(f"----------------training start!-----------------")
            if cfg.dic['train']['loader'].get('persist'):
```

```
            if cfg.persist_path == 'default':
                persistent_cache = pathlib.Path(experiment_path_fold, "persistent_cache")
            else:
                persistent_cache = pathlib.Path(cfg.persist_path)

            persistent_cache.mkdir(parents=True, exist_ok=True)
            train_ds = PersistentDataset(data=files[i]['train_files'], transform=train_transforms,
                                         cache_dir=persistent_cache)
            val_ds = PersistentDataset(data=files[i]['val_files'], transform=val_transforms,
                                       cache_dir=persistent_cache)
        else:
            train_ds = CacheDataset(
                data=files[i]['train_files'], transform=train_transforms,
                cache_rate=cfg.dic['train']['loader']['cache'], num_workers=4)
            # train_ds = Dataset(data=train_files, transform=train_transforms)
            # use batch_size=2 to load images and use RandCropByPosNegLabeld
            val_ds = CacheDataset(
                data=files[i]['val_files'], transform=val_transforms,
                cache_rate=cfg.dic['train']['loader']['cache'], num_workers=4)

        # load pretrained model
        out_channels = cfg.dic['model']['out_channels'] if cfg.dic['model'].get('out_channels', None) else cfg.model.out_channels
        cardio_vessel_segmentation_multi_phase_with_image_train(cfg=cfg,
                                                                key=cfg.second_stage_key,
                                                                model=cfg.model,
                                                                num_class=out_channels,
                                                                loss_function=cfg.train_loss,
                                                                val_metric=cfg.val_metric,
                                                                optimizer=cfg.optimizer_init,
                                                                lr_scheduler=cfg.lr_scheduler_init,
                                                                train_dataset=train_ds,
                                                                val_dataset=val_ds,
                                                                experiment_path=experiment_path_fold,
                                                                device=cfg.device,
                                                                start_epoch=cfg.start_epoch,
                                                                sw_batch_size=cfg.train_sw_batch_size,
                                                                overlap=cfg.train_sw_overlap,
                                                                mirror_axes=cfg.train_mirror_axes,
                                                                )
        if cfg.dic['train']['loader'].get('persist'):
            import shutil
            shutil.rmtree(persistent_cache)

    elif cfg.dic['mode'] == 'test':
        train_transforms, val_transforms, save_transform = get_multi_phase_transform_with_image(cfg.dic['transform'])

        cfg.creat_test_require()
        cfg.save_config(os.path.join(cfg.test_output_path, 'config.yaml'))
        if cfg.dic["test"]["loader"].get("file_path"):
            files = load_json(cfg.dic["test"]["loader"]["file_path"])["val_files"]
        elif cfg.dic["test"]["loader"].get("val_set"):
            files = prepare_main_auxiliary_with_img_datalist_with_file(main_file=cfg.CS_M,
                                                                       auxiliary_file=cfg.CS_A,
                                                                       main_img_file=cfg.I_M,
                                                                       auxiliary_img_file=cfg.I_A,
                                                                       label_file=cfg.test_label_path,
                                                                       broken_file=cfg.CS_DL,
                                                                       broken_gt_file=cfg.CS_DLGT,
                                                                       img_name=cfg.val_set,
                                                                       select_file=cfg.select_file, )
        else:
            files = prepare_main_auxiliary_with_img_datalist_with_file(main_file=cfg.CS_M,
                                                                       auxiliary_file=cfg.CS_A,
                                                                       main_img_file=cfg.I_M,
                                                                       auxiliary_img_file=cfg.I_A,
                                                                       label_file=cfg.test_label_path,
                                                                       broken_file=cfg.CS_DL,
                                                                       broken_gt_file=cfg.CS_DLGT,
                                                                       img_name=None,
                                                                       select_file=None, )
        # ---------------------------- built transform sequence ---------------------------- #
        total_ds = CacheDataset(
            data=files, transform=val_transforms, cache_rate=cfg.dic['test']['loader']['cache'], num_workers=2)

        # ------------------------------------ test ------------------------------------ #
        cardio_vessel_segmentation_multi_phase_with_image_test(model=cfg.model,
                                                               key=['I_M', 'CS_M', 'CS_A', 'CS_DLGT'],
                                                               val_dataset=total_ds,
                                                               device=cfg.device,
                                                               output_path=cfg.test_output_path,
                                                               window_size=cfg.dic['transform']['patch_size'],
                                                               save_data=cfg.dic['test']['save_data'])

    elif cfg.dic['mode'] == 'infer':
        infer_transforms = get_multi_phase_transform_with_image(cfg.dic['transform'], mode='infer')

        cfg.creat_infer_require()
        cfg.save_config(os.path.join(cfg.infer_output_path, 'config.yaml'))

        # ---------------------------- built transform sequence ---------------------------- #
        # files = prepare_image_list(image_path=cfg.dic['infer']['loader']['path'])

        files = prepare_main_auxiliary_with_img_datalist_with_file(main_file=cfg.CS_M,
                                                                   auxiliary_file=cfg.CS_A,
                                                                   main_img_file=cfg.I_M,
                                                                   auxiliary_img_file=cfg.I_A,
                                                                   label_file=None,
                                                                   broken_file=cfg.CS_DL,
                                                                   broken_gt_file=cfg.CS_DLGT,
                                                                   img_name=None,
                                                                   select_file=None, )
        total_ds = CacheDataset(
            data=files, transform=infer_transforms, cache_rate=cfg.dic['infer']['loader']['cache'], num_workers=2)

        cardio_vessel_segmentation_multi_phase_with_image_infer(model=cfg.model,
                                                                key=cfg.second_stage_key,
                                                                val_dataset=total_ds,
                                                                device=cfg.device,
                                                                output_path=cfg.infer_output_path,
                                                                window_size=tuple(cfg.dic['transform']['patch_size']),
                                                                origin_transforms=infer_transforms,
                                                                overlap=cfg.infer_sw_overlap,
                                                                sw_batch_size=cfg.infer_sw_batch_size,
                                                                mirror_axes=cfg.infer_mirror_axes
                                                                )
    else:
        raise RuntimeError('Only train and infer mode are supported now')
```

```python
SparrowLink/second_stage_only_one_phase_main.py
from monai.utils import first, set_determinism
from monai.data import CacheDataset, DataLoader, Dataset, decollate_batch, ITKReader, PersistentDataset
from monai.config import print_config
from monai.metrics.meandice import DiceMetric
import torch
import matplotlib.pyplot as plt
import os
import argparse
import monai
import logging
from utils.Config import Config
import sys
import time
from data.loader import prepare_datalist, prepare_datalist_with_file, \
    prepare_image_list, save_json, write_data_reference, load_json, prepare_main_with_img_datalist_with_file
from transform.utils import get_transform, get_second_stage_only_one_phase
from utils.test import cardio_vessel_segmentation_test, cardio_vessel_segmentation_multi_phase_with_image_test
from utils.trainer import cardio_vessel_segmentation_multi_phase_with_image_train
from utils.inferer import cardio_vessel_segmentation_infer, cardio_vessel_segmentation_multi_phase_with_image_infer
import pathlib
import torch.nn as nn
from torch.distributions.normal import Normal
# print_config()


torch.multiprocessing.set_sharing_strategy('file_system')


def load_weight_from_coarse_segmentation(in_channels, model, weight_path, net_architecture):
    """
    it will change
    """
    in_channels = model.in_channels if hasattr(model, "in_channels") else in_channels
    if net_architecture == 'UNet':
        # d = torch.load(
        #     './experiments/Graduate_project/multi_phase/pretrain/ResUnet/checkpoint/best_metric_model.pth')
        d = torch.load(weight_path)
        shape = list(d['model.0.conv.unit0.conv.weight'].shape)
        shape[1] = in_channels - 1
        if d.get('model.0.residual.weight'):
            noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['model.0.residual.weight'].device)
            d['model.0.residual.weight'] = torch.cat((d['model.0.residual.weight'], noise), dim=1)
        noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['model.0.conv.unit0.conv.weight'].device)
        d['model.0.conv.unit0.conv.weight'] = torch.cat((d['model.0.conv.unit0.conv.weight'], noise), dim=1)
        model.load_state_dict(d)

    elif net_architecture == "CS2net":
        d = torch.load(weight_path)
        shape = list(d['encoder1.conv1.weight'].shape)
        shape[1] = in_channels - 1
        noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['enc_input.conv1.weight'].device)
        d['enc_input.conv1.weight'] = torch.cat((d['enc_input.conv1.weight'], noise), dim=1)
        model.load_state_dict(d)

    elif net_architecture == "SkipDenseUnet":
        d = torch.load(weight_path)
        shape = list(d['features.conv0.weight'].shape)
        shape[1] = in_channels - 1
        noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['features.conv0.weight'].device)
        d['features.conv0.weight'] = torch.cat((d['features.conv0.weight'], noise), dim=1)
        model.load_state_dict(d)

    elif net_architecture == "SwinTransformerSys3D":
        d = torch.load(weight path)
        shape = list(d['patch_embed.proj.weight'].shape)
        shape[1] = in_channels - 1
        noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['patch_embed.proj.weight'].device)
        d['patch_embed.proj.weight'] = torch.cat((d['patch_embed.proj.weight'], noise), dim=1)
        model.load_state_dict(d)

    elif net_architecture == "nnunetv2":
        all_information = torch.load(weight_path)
        d = all_information['network_weights']
        shape = list(d['encoder.stages.0.0.convs.0.conv.weight'].shape)
        shape[1] = in_channels - 1
        noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['encoder.stages.0.0.convs.0.conv.weight'].device)
        d['encoder.stages.0.0.convs.0.conv.weight'] = torch.cat((d['encoder.stages.0.0.convs.0.conv.weight'], noise), dim=1)
        noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['encoder.stages.0.0.convs.0.all_modules.0.weight'].device)
        d['encoder.stages.0.0.convs.0.all_modules.0.weight'] = torch.cat((d['encoder.stages.0.0.convs.0.all_modules.0.weight'], noise), dim=1)
        noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['decoder.encoder.stages.0.0.convs.0.conv.weight'].device)
        d['decoder.encoder.stages.0.0.convs.0.conv.weight'] = torch.cat((d['decoder.encoder.stages.0.0.convs.0.conv.weight'], noise), dim=1)
        noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['decoder.encoder.stages.0.0.convs.0.all_modules.0.weight'].device)

        d['decoder.encoder.stages.0.0.convs.0.all_modules.0.weight'] = torch.cat((d['decoder.encoder.stages.0.0.convs.0.all_modules.0.weight'], noise), dim=1)
        model.load_state_dict(d)
    return model


if __name__ == "__main__":

    parser = argparse.ArgumentParser(description="cardio vessel segmentation")
    parser.add_argument("--configs", dest="cfg",
                        help="The configs file.",
                        default='./configs/heart_server.yaml',
                        type=str)
    parser.add_argument('--iters', dest='iters', help='Iterations in training.', type=int, default=None)
    parser.add_argument('--batch_size', dest='batch_size', help='Mini batch size of one gpu or cpu.', type=int, default=None)
    parser.add_argument('--learning_rate', dest='learning_rate', help='Learning rate', type=float, default=None)
    parser.add_argument('--seed', dest='seed', help='seed', type=float, default=0)
    parser.add_argument('--mode', dest='mode', help='train, infer or both', type=str, default=None)

    parser.add_argument('--experiments_path', dest='experiments_path',
                        help='experiments_path, all output and checkpoint are saved here.', type=str, default=None)

    parser.add_argument('--pretrain_weight_path', help='pretrain weight for training, testing of inferring', type=str, default=None)
    parser.add_argument('--img_path', help='image path', type=str, default=None)

    parser.add_argument('--label_path', help='label path', type=str, default=None)
    parser.add_argument('--output_path', help='save path', type=str, default=None)
    parser.add_argument('--persist_path', help='persist path, the path used to save persist data in persist loader',
                        type=str, default=None)
    parser.add_argument('--val_set', help='val set, a txt path to select val data, only used if mode is train',
                        type=str, default=None)
    parser.add_argument('--train_set', help='train set, a txt path to select train data, only used if mode is train',
                        type=str, default=None)
    parser.add_argument('--dataset_information', help='dataset information',
                        type=str, default=None)

    # ---------------------------------------- multi information ---------------------------------------- #
    parser.add_argument('--CS_W',
                        help='pretrain weight from coarse segmentation, testing of inferring', type=str, default=None)
    parser.add_argument('--CS_M', help='coarse segmentation in main phase', type=str, default=None)
    parser.add_argument('--CS_A', help='coarse segmentation in auxiliary phase', type=str, default=None)
    parser.add_argument('--CS_DL', help='discontinuity label generated in main coarse segmentation', type=str,
                        default=None)
```

```
    parser.add_argument('--CS_DLGT', help='discontinuity label generated in auxiliary coarse segmentation', type=str,
                        default=None)
    parser.add_argument('--I_M', help='main image', type=str, default=None)
    parser.add_argument('--I_A', help='auxiliary image', type=str, default=None)
    parser.add_argument('--select_file', help='use discontinuity label to determine what data are used', type=str,
                        default=None)
    parser.add_argument('--delete_persist', help='pretrain weight from coarse segmentation, testing of inferring', type=str,)
    parser.add_argument('--view', action='store_true', default=False, help='delete persist cache, because it is too large')
    # ---------------------------------------- multi information ---------------------------------------- #

    args = parser.parse_args()
    monai.config.print_config()

    logging.basicConfig(stream=sys.stdout, level=logging.INFO)
    cfg = Config(
        args.cfg,
        learning_rate=args.learning_rate,
        iters=args.iters,
        batch_size=args.batch_size,
        seed=args.seed,
        mode=args.mode,
        img_path=args.img_path,
        I_M=args.I_M,
        I_A=args.I_A,
        CS_M=args.CS_M,
        CS_A=args.CS_A,
        CS_DL=args.CS_DL,
        CS_DLGT=args.CS_DLGT,
        CS_W=args.CS_W,
        select_file=args.select_file,
        label_path=args.label_path,
        output_path=args.output_path,
        persist_path=args.persist_path,
        val_set=args.val_set,
        train_set=args.train_set,
        experiments_path=args.experiments_path,
        pretrain_weight_path=args.pretrain_weight_path,
        dataset_information=args.dataset_information
        )
    set_determinism(seed=cfg.seed)

    if cfg.dic['mode'] == 'train':

        # ---------------------------- built transform sequence ---------------------------- #
        # ------------- take a look at val_transform on dataset and save a case ------------ #
        cfg.creat_training_require()
        load_weight_from_coarse_segmentation(in_channels=cfg.dic['model'].get("in_channels"), model=cfg.model,
                                             net_architecture=cfg.dic['model']['name'], weight_path=cfg.CS_W)
        train_transforms, val_transforms, save_transform = get_second_stage_only_one_phase(cfg.dic['transform'])
        if cfg.dic["train"]["loader"].get("file_path"):
            files = [load_json(cfg.dic["train"]["loader"]["file_path"])]
        elif cfg.dic["train"]["loader"].get("val_set"):
            val_files = prepare_main_with_img_datalist_with_file(main_file=cfg.CS_M,
                                                                 main_img_file=cfg.I_M,
                                                                 label_file=cfg.train_label_path,
                                                                 broken_file=cfg.CS_DL,
                                                                 broken_gt_file=cfg.CS_DLGT,
                                                                 img_name=cfg.val_set,
                                                                 select_file=cfg.select_file, )

            train_files = prepare_main_with_img_datalist_with_file(main_file=cfg.CS_M,
                                                                   main_img_file=cfg.I_M,
                                                                   label_file=cfg.train_label_path,
                                                                   broken_file=cfg.CS_DL,
                                                                   broken_gt_file=cfg.CS_DLGT,
                                                                   img_name=cfg.train_set,
                                                                   select_file=cfg.select_file, )
            files = [{"train_files": train_files, "val_files": val_files}]

        else:
            raise ValueError("please provide a file path or val set and train set")
        # check_transform_in_dataloader(val_files=files[0]['val_files'], val_transforms=try_transforms)

        # ---------------------------- create loss function ---------------------------- #
        # ------------- you can crate your own loss or metric here amd replace cfg ------------ #

        # dice_metric = DiceMetric(include_background=False, reduction="mean")
        # loss_function = DiceLoss(to_onehot_y=True, softmax=True)

        # -------------- training ---------------- #
        metric_record = []
        experiment_path = os.path.join(cfg.dic['experiments path'], time.strftime("%d_%m_%Y_%H_%M_%S"))\
            if cfg.dic.get('time_name', None) else cfg.dic['experiments_path']

        if not os.path.exists(experiment_path):
            os.makedirs(experiment_path)
        for i in range(len(files)):
            if cfg.dic['train']['loader'].get('split_mode') == "five_fold":
                experiment_path_fold = os.path.join(experiment_path, f"{i}_fold")
                if not os.path.exists(experiment_path_fold):
                    os.makedirs(experiment_path_fold)
            else:
                experiment_path_fold = experiment_path
            write_data_reference(files[i], experiment_path_fold)
            save_json(files[i], os.path.join(experiment_path_fold, 'files.txt'))
            # -------------- Create Model, Loss, Optimizer in Config-------------- #
            # ------------------------ using config ------------------------------ #
            cfg.save_config(os.path.join(experiment_path_fold, 'configs.yaml'))
            if cfg.dic['train']['loader'].get('split_mode') == "five_fold":
                print(f"----------------fold{i} start!-----------------")
            else:
                print(f"----------------training start!-----------------")
            if cfg.dic['train']['loader'].get('persist'):
                if cfg.persist_path == 'default':
                    persistent_cache = pathlib.Path(experiment_path_fold, "persistent_cache")
                else:
                    persistent_cache = pathlib.Path(cfg.persist_path)

                persistent_cache.mkdir(parents=True, exist_ok=True)
                train_ds = PersistentDataset(data=files[i]['train_files'], transform=train_transforms,
                                             cache_dir=persistent_cache)
                val_ds = PersistentDataset(data=files[i]['val_files'], transform=val_transforms,
                                           cache_dir=persistent_cache)
            else:
                train_ds = CacheDataset(
                    data=files[i]['train_files'], transform=train_transforms,
                    cache_rate=cfg.dic['train']['loader']['cache'], num_workers=4)
                # train_ds = Dataset(data=train_files, transform=train_transforms)
                # use batch_size=2 to load images and use RandCropByPosNegLabeld
                val_ds = CacheDataset(
                    data=files[i]['val_files'], transform=val_transforms,
                    cache_rate=cfg.dic['train']['loader']['cache'], num_workers=4)

            # load pretrained model
            out_channels = cfg.dic['model']['out_channels'] if cfg.dic['model'].get('out_channels', None) else cfg.model.out_channels
            cardio_vessel_segmentation_multi_phase_with_image_train(cfg=cfg,
                                                                    key=cfg.second_stage_key,
```

```
                                                                model=cfg.model,
                                                                num_class=out_channels,
                                                                loss_function=cfg.train_loss,
                                                                val_metric=cfg.val_metric,
                                                                optimizer=cfg.optimizer_init,
                                                                lr_scheduler=cfg.lr_scheduler_init,
                                                                train_dataset=train_ds,
                                                                val_dataset=val_ds,
                                                                experiment_path=experiment_path_fold,
                                                                device=cfg.device,
                                                                start_epoch=cfg.start_epoch,
                                                                sw_batch_size=cfg.train_sw_batch_size,
                                                                overlap=cfg.train_sw_overlap,
                                                                mirror_axes=cfg.train_mirror_axes,
                                                                )
            if cfg.dic['train']['loader'].get('persist'):
                import shutil
                shutil.rmtree(persistent_cache)

    elif cfg.dic['mode'] == 'test':
        train_transforms, val_transforms, save_transform = get_second_stage_only_one_phase(cfg.dic['transform'])

        cfg.creat_test_require()
        cfg.save_config(os.path.join(cfg.test_output_path, 'config.yaml'))
        if cfg.dic["test"]["loader"].get("file path"):
            files = load_json(cfg.dic["test"]["loader"]["file_path"])["val_files"]
        elif cfg.dic["test"]["loader"].get("val_set"):
            files = prepare_main_with_img_datalist_with_file(main_file=cfg.CS_M,
                                                             main_img_file=cfg.I_M,
                                                             label_file=cfg.test_label_path,
                                                             broken_file=cfg.CS_DL,
                                                             broken_gt_file=cfg.CS_DLGT,
                                                             img_name=cfg.val_set,
                                                             select_file=cfg.select_file, )
        else:
            files = prepare_main_with_img_datalist_with_file(main_file=cfg.CS_M,
                                                             main_img_file=cfg.I_M,
                                                             label_file=cfg.test_label_path,
                                                             broken_file=cfg.CS_DL,
                                                             broken_gt_file=cfg.CS_DLGT,
                                                             img_name=None,
                                                             select_file=None, )
        # --------------------------- built transform sequence --------------------------- #
        total_ds = CacheDataset(
            data=files, transform=val_transforms, cache_rate=cfg.dic['test']['loader']['cache'], num_workers=2)

        # ----------------------------------- test ----------------------------------- #
        cardio_vessel_segmentation_multi_phase_with_image_test(model=cfg.model,
                                                               key=['I_M', 'CS_M', 'CS_A', 'CS_DLGT'],
                                                               val_dataset=total_ds,
                                                               device=cfg.device,
                                                               output_path=cfg.test_output_path,
                                                               window_size=cfg.dic['transform']['patch_size'],
                                                               save_data=cfg.dic['test']['save_data'])

    elif cfg.dic['mode'] == 'infer':
        infer_transforms = get_second_stage_only_one_phase(cfg.dic['transform'], mode='infer')

        cfg.creat_infer_require()
        cfg.save_config(os.path.join(cfg.infer_output_path, 'config.yaml'))

        # --------------------------- built transform sequence --------------------------- #
        # files = prepare_image_list(image_path=cfg.dic['infer']['loader']['path'])

        files = prepare_main_with_img_datalist_with_file(main_file=cfg.CS_M,
                                                         main_img_file=cfg.I_M,
                                                         label_file=None,
                                                         broken_file=cfg.CS_DL,
                                                         broken_gt_file=cfg.CS_DLGT,
                                                         img_name=None,
                                                         select_file=None, )
        total_ds = CacheDataset(
            data=files, transform=infer_transforms, cache_rate=cfg.dic['infer']['loader']['cache'], num_workers=2)

        cardio_vessel_segmentation_multi_phase_with_image_infer(model=cfg.model,
                                                                key=cfg.second_stage_key,
                                                                val_dataset=total_ds,
                                                                device=cfg.device,
                                                                output_path=cfg.infer_output_path,
                                                                window_size=tuple(cfg.dic['transform']['patch_size']),
                                                                origin_transforms=infer_transforms,
                                                                overlap=cfg.infer_sw_overlap,
                                                                sw_batch_size=cfg.infer_sw_batch_size,
                                                                mirror_axes=cfg.infer_mirror_axes
                                                                )
    else:
        raise RuntimeError('Only train and infer mode are supported now')
```

```python
SparrowLink/slicer_visulization/slicer_mark_up.py
import json


def create_extended_plane_markup_json(center, normal, bounds, coordinate_system="LPS", plane_type="pointNormal",
                                      size_mode="absolute", auto_scaling_factor=1.0, markup_orientation=None,
                                      orientation=None, object_to_base=None, base_to_node=None, color=[0, 0, 1]):
    """
    Create an extended JSON string for a plane markup in 3D Slicer, including various properties.

    Parameters:
    - center: The center of the plane in world coordinates.
    - normal: The normal vector of the plane.
    - size: The size of the plane (length, width, height).
    - bounds: The bounds of the plane in world coordinates.
    - coordinate_system: The coordinate system used (default "LPS").
    - plane_type: The type of the plane (default "pointNormal").
    - size_mode: The size mode of the plane (default "absolute").
    - auto_scaling_factor: Auto scaling factor for the plane size (default 1.0).
    - orientation: The orientation matrix of the plane (3x3 matrix).
    - object_to_base: The object to base transform matrix (4x4 matrix).
    - base_to_node: The base to node transform matrix (4x4 matrix).

    Returns:
    - A JSON string representing the extended markup for 3D Slicer.
    """
    # Default values for orientation and transform matrices if not provided
    if orientation is None:
        orientation = [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
    if object_to_base is None:
        object_to_base = orientation[:3] + [0.0] + orientation[3:6] + [0.0] + orientation[6:] + [0.0, 0.0, 0.0, 0.0, 1.0]
    if base_to_node is None:
        base_to_node = markup_orientation[:3] + [center[0]] + markup_orientation[3:6] + [center[1]] + markup_orientation[6:] + [center[2], 0.0, 0.0, 0.0, 1.0]

    # calculate size
    length = bounds[1] - bounds[0]  # xMax - xMin
    width = bounds[3] - bounds[2]  # yMax - yMin
    size = [length, width, 0.0]

    markup_json = {
        "@schema": "https://raw.githubusercontent.com/slicer/slicer/master/Modules/Loadable/Markups/Resources/Schema/markups-schema-v1.0.3.json#",
        "markups": [
            {
                "type": "Plane",
                "coordinateSystem": coordinate_system,
                "coordinateUnits": "mm",
                "locked": False,
                "fixedNumberOfControlPoints": False,
                "labelFormat": "%N-%d",
                "lastUsedControlPointNumber": 1,
                "planeType": plane_type,
                "sizeMode": size_mode,
                "autoScalingFactor": auto_scaling_factor,
                "center": center,
                "normal": normal, # "objectToBase": object_to_base,
                "baseToNode": base_to_node,
                "orientation": markup_orientation,
                "size": size,
                "planeBounds": bounds,
                "display":
                    {
                        "color": color,
                    },
                "controlPoints": [
                    {
                        "id": "1",
                        "label": "P-1",
                        "position": center,
                        "orientation": orientation,
                        "selected": True,
                        "locked": False,
                        "visibility": True,
                        "positionStatus": "defined"
                    }
                ],
                "measurements": [
                    {
                        "name": "area",
                        "enabled": False,
                        "units": "cm2",
                        "printFormat": "%-#4.4g%s"
                    }
                ]

            }
        ]
    }

    return json.dumps(markup_json, indent=4)
```

```python
SparrowLink/slicer_visulization/slicer_mvf.py
import nrrd
import numpy as np


def get_space_full_name(space):
    assert len(space) == 3, "space should be a 3-letter string"
    full_name = []
    for s in space:
        if s == "L":
            full_name.append("left")
        elif s == "R":
            full_name.append("right")
        elif s == "A":
            full_name.append("anterior")
        elif s == "P":
            full_name.append("posterior")
        elif s == "S":
            full_name.append("superior")
        elif s == "I":
            full_name.append("inferior")
    # link with _ to form full name
    return "-".join(full_name)


def get_direction(space):
    """3D-slicer use default RAS space, so we need to convert the space to RAS"""
    direction = [1, 1, 1]
    if space[0] == "L":
        direction[0] = -1
    if space[1] == "P":
        direction[1] = -1
    if space[2] == "I":
        direction[2] = -1
    return direction


def save_mvf(mvf, save_path, affine, space, scale_factor=10):
    offset = np.array((affine[:3, 3]))
    direction = np.array((affine[:3, :3])).tolist()
    """Seems like mvf in 3D slicer do not consider the direction of the space,
    so we need to multiply the mvf with the direction matrix to make it consistent with the space"""
    mvf_direction = get_direction(space)
    mvf[:, :, :, 0] = mvf[:, :, :, 0] * mvf_direction[0]
    mvf[:, :, :, 1] = mvf[:, :, :, 1] * mvf_direction[1]
    mvf[:, :, :, 2] = mvf[:, :, :, 2] * mvf_direction[2]
    header = {
        'endian': 'little',
        'encoding': 'raw',
        'space': get_space_full_name(space),
        'space directions': direction + [None],
        'space origin': offset,
        'kinds': ['domain', 'domain', 'domain', 'vector'],
    }
    nrrd.write(save_path, mvf * scale_factor, header=header)


def save_image(image, save_path, affine, space):
    offset = np.array((affine[:3, 3]))
    direction = np.array((affine[:3, :3])).tolist()
    header = {
        'endian': 'little',
        'encoding': 'raw',
        'space': get_space_full_name(space),
        'space directions': direction,
        'space origin': offset,
        'kinds': ['domain', 'domain', 'domain'],
    }
    nrrd.write(save_path, image, header=header)
```

```python
SparrowLink/SparrowLink_metric.py
from post_processing.fracture_detection import sparrowlink_metric
import numpy as np
import SimpleITK as sitk
import json
from openpyxl import Workbook

class SparrowLinkMetric(object):
    def __init__(self,
                 max_broken_size_percentage=0.1,
                 min_sphere_radius_percentage=0.02,
                 sphere_dilation_percentage=0.01,
                 region_select=4,
                 cube_min_size=(10, 10, 10),
                 skeleton_refine_times=3,
                 region_threshold=5,
                 angle_threshold_1=0.0,
                 angle_threshold_2=0.0,
                 view=False
                 ):
        self.max_broken_size_percentage = max_broken_size_percentage
        self.min_sphere_radius_percentage = min_sphere_radius_percentage
        self.sphere_dilation_percentage = sphere_dilation_percentage
        self.region_select = region_select
        self.cube_min_size = cube_min_size
        self.skeleton_refine_times = skeleton_refine_times
        self.region_threshold = region_threshold
        self.angle_threshold_1 = angle_threshold_1
        self.angle_threshold_2 = angle_threshold_2
        self.view = view
    """
    :param max_broken_size_percentage: the max broken size percentage, used in constriction on distance
    :param min_sphere_radius_percentage: the min sphere radius percentage, used for some small fracture
    :param sphere_dilation_percentage: the sphere dilation percentage, to consider the structure around fracture area
    :param region_select: the region select, matching points consist of start point and paired point, start point is...
    ...constricted in the region with region_select.
    :param cube_min_size: the min size of the cube, used for small fracture
    :param skeleton_refine_times: the refine times of the skeleton, zhang algorithm might cause some small branch in the...
    ...skeleton, so we need to refine the skeleton to delete the small branch
    :param region_threshold: the region threshold, used to delete the small region in the segmentation
    :param angle_threshold_1: the angle threshold 1, used for constriction on orientation
    :param angle_threshold_2: the angle threshold 2, used for constriction on orientation
    """
    def __call__(self,
                 seg_path: str = None,
                 gt_path: str = None,
                 save_path: str = None
                 ):

        seg, spacing, origin, direction = self.read_image(seg_path)
        self.spacing, self.origin, self.direction = spacing, origin, direction
        gt, _, _, _ = self.read_image(gt_path)
        result_dict = sparrowlink_metric(
            label=seg,
            GT=gt,
            spacing=spacing,
            direction=direction,
            origin=origin,
            save_path=save_path,
            view=self.view,
            max_broken_size_percentage=self.max_broken_size_percentage,
            min_sphere_radius_percentage=self.min_sphere_radius_percentage,
            sphere_dilation_percentage=self.sphere_dilation_percentage,
            cube_min_size=self.cube_min_size,
            skeleton_refine_times=self.skeleton_refine_times,
            region_threshold=self.region_threshold,
            angle_threshold_1=self.angle_threshold_1,
            angle_threshold_2=self.angle_threshold_2,
        )
        d = {
            "name": pathlib.Path(seg_path).name[:9],
            "num_gt": result_dict["num_gt"],
        }
        ##### permenant #####
        if "_CS_M" in str(seg_path):
            print(1)
            sphere_gt = result_dict["mask_sphere_gt"]
            gt_segment = ((sphere_gt * gt) > 0).astype(np.uint16)
            self.save_image(gt_segment, save_path=seg_path.replace(".nii.gz", "_sphere_gt_segment.nii.gz"))
            self.save_image(sphere_gt, save_path=seg_path.replace(".nii.gz", "_sphere_gt.nii.gz"))
            path = seg_path.replace("_CS_M.nii.gz", "_RCS_NEW.nii.gz")
            rcs_new = self.read_image(path)[0]
            rcs_new_gt_sphere_segment = ((sphere_gt * rcs_new) > 0).astype(np.uint16)

            self.save_image(rcs_new_gt_sphere_segment,
                            save_path=seg_path.replace(".nii.gz", "_rcs_new_gt_sphere_segment.nii.gz"))
            cs_m_gt_sphere_segment = ((sphere_gt * seg) > 0).astype(np.uint16)
            self.save_image(cs_m_gt_sphere_segment,
                            save_path=seg_path.replace(".nii.gz", "_CS_M_gt_sphere_segment.nii.gz"))
        return d

    def save_image(self, image, save_path, save_type=np.uint16):
        sitk_image = sitk.GetImageFromArray(image.astype(save_type))
        sitk_image.SetSpacing(self.spacing)
        sitk_image.SetOrigin(self.origin)
        sitk_image.SetDirection(self.direction)
        sitk.WriteImage(sitk_image, save_path)

    @staticmethod
```

```python
    def read_image(path):
        sitk_image = sitk.ReadImage(path)
        image = sitk.GetArrayFromImage(sitk_image)
        spacing = sitk_image.GetSpacing()
        origin = sitk_image.GetOrigin()
        direction = sitk_image.GetDirection()
        return image, spacing, origin, direction


def update(pbar, record, result):
    pbar.update()
    record.append(result)


def error_back(err):
    print(err)


if __name__ == "__main__":
    import argparse
    import tqdm
    import pathlib
    from multiprocessing import Pool

    import warnings
    warnings.filterwarnings("ignore")  # ignore from np.int16 to np.uint8

    parser = argparse.ArgumentParser()
    parser.add_argument("--seg", type=str, default=None)
    parser.add_argument('--seg_find', type=str, default="*.nii.gz")
    parser.add_argument("--gt", type=str, default=None)
    parser.add_argument('--gt_find', type=str, default="*.nii.gz")
    parser.add_argument("--multiprocess", action='store_true', default=False)
    parser.add_argument('--metric_postfix', type=str, default=None)
    args = parser.parse_args()

    assert args.seg is not None, "seg_path is None"
    assert args.gt is not None, "label_path is None"
    if args.metric_postfix is None:
        metric_result_path = str(pathlib.Path(args.seg) / "sparrowlink_metric.xlsx")
    else:
        metric_result_path = str(pathlib.Path(args.seg) / f"sparrowlink_metric_{args.metric_postfix}.xlsx")

    wb = Workbook()
    ws = wb.active
    ws.append(['name', 'num_gt'])
    metric_list = []

    seg = pathlib.Path(args.seg)
    gt = pathlib.Path(args.gt)
    seg_list = list(seg.glob(f"{args.seg_find}"))
    gt_list = list(gt.glob(f"{args.gt_find}"))
    seg_list.sort()
    gt_list.sort()
    print(f"\033[96m SparrowLink Metric Calculating for {args.seg_find} \033[00m")
    pbar = tqdm.tqdm(total=len(seg_list), colour="#87cefa")
    pbar.set_description("SparrowLink Processing")
    metric_record = []
    poor = Pool(14)
    for seg_path, gt_path in zip(seg_list, gt_list):
        name = seg_path.name[:9]
        metric_calculator = SparrowLinkMetric(view=False)
        if not args.multiprocess:
            result_dict = metric_calculator(
                seg_path=str(seg_path),
                gt_path=str(gt_path),
                save_path=None
            )
            metric_record.append(result_dict)
            pbar.update()
        else:
            kwargs = {
                "seg_path": str(seg_path),
                "gt_path": str(gt_path),
                "save_path": None
            }
            poor.apply_async(
                func=metric_calculator,
                kwds=kwargs,
                callback=lambda x: update(pbar, metric_record, x),
                error_callback=error_back
            )
    metric_record.sort(key=lambda x: x.get("name"))
    if args.multiprocess:
        poor.close()
        poor.join()

    for result in metric_record:
        ws.append([result["name"], result["num_gt"]])

    mean_metric = np.array([metric_record[i].get('num_gt') for i in range(len(metric_record))]).mean(axis=0).tolist()
    ws.append(["mean", mean_metric])
    wb.save(metric_result_path)
```

```python
SparrowLink/SparrowLink_Post_Process.py
from post_processing.fracture_detection import run as discontinuity_detection
from post_processing.merge import merge
from post_processing.select_two_region import select_two_biggest_connected_region
import numpy as np
from typing import Any, Dict, Hashable, List, Mapping, Optional, Sequence, Tuple, Union
import SimpleITK as sitk
import pathlib
import cc3d
import shutil
import json


class SparrowLinkPostProcess:
    def __init__(self,
                 cs_m_path: str = None,
                 cs_a_path: str = None,
                 rs_path: str = None,
                 cs_dl_path: str = None,
                 save_path: str = None,
                 max_broken_size_percentage=0.1,
                 ):
        """
        cs_m_path: path to the cs_m (coarse segmentation in main phase)
        cs_a_path: path to the cs_a (coarse segmentation in auxiliary phase)
        rs_a_path: path to the rs_a (refined segmentation in auxiliary phase)
        cs_dl_path: path to the cs_dl (coarse segmentation discontinuity label)
        """
        assert cs_m_path is not None, "cs_m_path is None"
        self.cs_m_path = cs_m_path
        self.cs_a_path = cs_a_path
        self.rs_path = rs_path
        self.cs_dl_path = cs_dl_path
        self.save_path = save_path
        self.name = pathlib.Path(cs_m_path).name[:9] # need to be modify to fit the other data
        self.save_subdir = pathlib.Path(save_path) / self.name
        self.save_subdir.mkdir(parents=True, exist_ok=True)
        self.discontinuity_detection_default = SparrowLinkDiscontinuityDetection(view=False,
                                                                                  max_broken_size_percentage=max_broken_size_percentage)
        self.discontinuity_detection = SparrowLinkDiscontinuityDetection(angle_threshold_2=-1,
                                                                         region_select=None,
                                                                         view=False,
                                                                         max_broken_size_percentage=max_broken_size_percentage)
        self.spacing = None
        self.origin = None
        self.direction = None

    def run(self):
        """
        Run the post processing pipeline.
        1. load the data
        2. directly merge rs and cs_m with cs_dl -> rcs
        3. select_two_biggest_connected_region on rcs -> rcs_two
        4. run discontinuity detection on rcs with cs_a and cs_dl -> arcs
        5. select_two_biggest_connected_region on arcs -> arcs_two

        if selected merge:
        6. selectively merge rs and cs_m with cs_dl -> rcs_selected
        7. select_two_biggest_connected_region on rcs_select -> rcs_selected_two
        8. run final merge on rcs_select with cs_a and cs_dl -> arcs_selected
        9. select_two_biggest_connected_region on arcs_selected -> arcs_selected_two

        if new merge:
        10. selectively merge rs and cs_m without cs_dl -> rcs_new
        11. select_two_biggest_connected_region on rcs_new -> rcs_new_two
        12. run final merge on rcs_new with cs_a and cs_dl -> arcs_new
        13. select_two_biggest_connected_region on arcs_new -> arcs_new_two
        """
        cs_a, rs, cs_dl, cs_m = None, None, None, None
        # 1. load the data
        cs_m, spacing, origin, direction = self.read_image(self.cs_m_path)
        self.spacing, self.origin, self.direction = spacing, origin, direction

        cs_a, _, _, _ = self.read_image(self.cs_a_path)
        rs, _, _, _ = self.read_image(self.rs_path)
        cs_dl, _, _, _ = self.read_image(self.cs_dl_path)

        # 2. directly merge rs and cs_m with cs_dl -> rcs
        rcs = self.merge(m1=cs_m, m2=rs, dl=cs_dl)

        # 3. select_two_biggest_connected_region on rcs -> rcs_two
        rcs_two = select_two_biggest_connected_region(rcs)
        cs_m_two = select_two_biggest_connected_region(cs_m)


        # 4. run discontinuity detection on rcs with cs_a and cs_dl -> arcs
        result_dict = self.discontinuity_detection_default(seg_array=rcs,
                                                           gt_array=None,
                                                           auxiliary_array=cs_a,
                                                           pre_mask_array=cs_dl,
                                                           spacing=self.spacing,
                                                           origin=self.origin,
                                                           direction=self.direction,
                                                           save_path=None,
                                                           )
        rcs_sphere = result_dict.get("mask_sphere")
        rcs_refined_num = result_dict.get("num")
        assert rcs_sphere is not None, "rcs_sphere is None"
        assert rcs_refined_num is not None, "rcs_num is None"
        arcs = self.merge(m1=rcs, m2=cs_a, dl=rcs_sphere)

        # 5. select_two_biggest_connected_region on arcs -> arcs_two
        arcs_two = select_two_biggest_connected_region(arcs)

        # 6. selectively merge rs and cs_m with cs_dl -> rcs_selected
        result_dict = self.discontinuity_detection_default(seg_array=cs_m,
                                                           gt_array=None,
                                                           auxiliary_array=rs,
```

```
                                                    pre_mask_array=cs_dl,
                                                    spacing=self.spacing,
                                                    origin=self.origin,
                                                    direction=self.direction,
                                                    save_path=None,
                                                    )
        cs_m_selected_sphere = result_dict.get("mask_sphere")
        cs_m_selected_refined_num = result_dict.get("num")
        assert cs_m_selected_sphere is not None, "rcs_selected_sphere is None"
        assert cs_m_selected_refined_num is not None, "rcs_selected_refined_num is None"
        rcs_selected = self.merge(m1=cs_m, m2=rs, dl=cs_m_selected_sphere)

        # 7. select_two_biggest_connected_region on rcs_select -> rcs_selected_two
        rcs_selected_two = select_two_biggest_connected_region(rcs_selected)

        # 8. merge

        arcs_selected = self.selected_final_merge(rcs_select=rcs_selected,
                                                  arcs=arcs,
                                                  dl1=cs_dl,
                                                  dl2=rcs_sphere)

        # 9. select_two_biggest_connected_region on arcs_selected -> arcs_selected_two
        arcs_selected_two = select_two_biggest_connected_region(arcs_selected)

        # 10. selectively merge rs and cs_m without cs_dl -> rcs_new
        result_dict = self.discontinuity_detection(seg_array=cs_m,
                                                   gt_array=None,
                                                   auxiliary_array=rs,
                                                   pre_mask_array=None,
                                                   spacing=self.spacing,
                                                   origin=self.origin,
                                                   direction=self.direction,
                                                   save_path=None,
                                                   )

        cs_m_new_sphere = result_dict.get("mask_sphere")
        cs_m_new_refined_num = result_dict.get("num")
        assert cs_m_new_sphere is not None, "rcs_new_sphere is None"
        assert cs_m_new_refined_num is not None, "rcs_new_refined_num is None"
        rcs_new = self.merge(m1=cs_m, m2=rs, dl=cs_m_new_sphere)

        # 11. select_two_biggest_connected_region on rcs_select -> rcs_selected_two
        rcs_new_two = select_two_biggest_connected_region(rcs_new)

        # 12. merge

        arcs_new = self.selected_final_merge(rcs_select=rcs_new,
                                             arcs=arcs,
                                             dl1=cs_dl,
                                             dl2=rcs_sphere)

        # 13. select_two_biggest_connected_region on arcs_selected -> arcs_selected_two
        arcs_new_two = select_two_biggest_connected_region(arcs_new)


        # 14. save
        self.save_image(rcs, self.save_subdir / f"{self.name}_RCS.nii.gz")
        self.save_image(rcs_two, self.save_subdir / f"{self.name}_RCS_TWO.nii.gz")
        self.save_image(arcs, self.save_subdir / f"{self.name}_ARCS.nii.gz")
        self.save_image(arcs_two, self.save_subdir / f"{self.name}_ARCS_TWO.nii.gz")
        self.save_image(rcs_selected, self.save_subdir / f"{self.name}_RCS_SELECTED.nii.gz")
        self.save_image(rcs_selected_two, self.save_subdir / f"{self.name}_RCS_SELECTED_TWO.nii.gz")
        self.save_image(cs_m_two, self.save_subdir / f"{self.name}_CS_M_TWO.nii.gz")
        self.save_image(arcs_selected, self.save_subdir / f"{self.name}_ARCS_SELECTED.nii.gz")
        self.save_image(arcs_selected_two, self.save_subdir / f"{self.name}_ARCS_SELECTED_TWO.nii.gz")
        self.save_image(rcs_new, self.save_subdir / f"{self.name}_RCS_NEW.nii.gz")
        self.save_image(rcs_new_two, self.save_subdir / f"{self.name}_RCS_NEW_TWO.nii.gz")
        self.save_image(arcs_new, self.save_subdir / f"{self.name}_ARCS_NEW.nii.gz")
        self.save_image(arcs_new_two, self.save_subdir / f"{self.name}_ARCS_NEW_TWO.nii.gz")

        # save little segment and sphere for visualization
        self.save_image(cs_m_selected_sphere, self.save_subdir / f"{self.name}_cs_m_selected_sphere.nii.gz")
        self.save_image(rcs_sphere, self.save_subdir / f"{self.name}_rcs_sphere.nii.gz")
        self.save_image(rcs * cs_dl, self.save_subdir / f"{self.name}_cs_m_merge_segment.nii.gz")
        self.save_image(cs_m_selected_sphere * rs, self.save_subdir / f"{self.name}_rcs_selected_merge_segment.nii.gz")
        self.save_image(rcs_sphere * cs_a, self.save_subdir / f"{self.name}_rcs_merge_segment.nii.gz")
        self.save_image(cs_m_new_sphere, self.save_subdir / f"{self.name}_cs_m_new_sphere.nii.gz")
        self.save_image(cs_m_new_sphere * rcs_new, self.save_subdir / f"{self.name}_rcs_new_merge_segment.nii.gz")

        # move cs_a, cs_m, cs_dl, rs, gt to save_subdir for visualization
        shutil.copy(self.cs_a_path, self.save_subdir / f"{self.name}_CS_A.nii.gz")
        shutil.copy(self.cs_m_path, self.save_subdir / f"{self.name}_CS_M.nii.gz")
        shutil.copy(self.cs_dl_path, self.save_subdir / f"{self.name}_CS_DL.nii.gz")
        shutil.copy(self.rs_path, self.save_subdir / f"{self.name}_RS.nii.gz")

        # save discontinuity metric
        d = {
            "name": self.name,
            "cs_m_selected_refined_num": cs_m_selected_refined_num,
            "arcs_improve": rcs_refined_num,
            "cs_m_new_refined_num": cs_m_new_refined_num,
        }
        return d

    def run_without_cs_a(self):
        """
        Run the post processing pipeline.
        1. load the data
        2. directly merge rs and cs_m with cs_dl -> rcs
        3. select_two_biggest_connected_region on rcs -> rcs_two
        4. run discontinuity detection on rcs with cs_a and cs_dl -> arcs
        5. select_two_biggest_connected_region on arcs -> arcs_two

        if selected merge:
        6. selectively merge rs and cs_m with cs_dl -> rcs_selected
        7. select_two_biggest_connected_region on rcs_select -> rcs_selected_two
        8. run final merge on rcs_select with cs_a and cs_dl -> arcs_selected
        9. select_two_biggest_connected_region on arcs_selected -> arcs_selected_two
```

```
        if new merge:
        10. selectively merge rs and cs_m without cs_dl -> rcs_new
        11. select_two_biggest_connected_region on rcs_new -> rcs_new_two
        12. run final merge on rcs_new with cs_a and cs_dl -> arcs_new
        13. select_two_biggest_connected_region on arcs_new -> arcs_new_two
        """
        cs_a, rs, cs_dl, cs_m = None, None, None, None
        # 1. load the data
        cs_m, spacing, origin, direction = self.read_image(self.cs_m_path)
        self.spacing, self.origin, self.direction = spacing, origin, direction

        rs, _, _, _ = self.read_image(self.rs_path)
        cs_dl, _, _, _ = self.read_image(self.cs_dl_path)

        rs_two = select_two_biggest_connected_region(rs)

        # 2. directly merge rs and cs_m with cs_dl -> rcs
        rcs = self.merge(m1=cs_m, m2=rs, dl=cs_dl)

        # 3. select_two_biggest_connected_region on rcs -> rcs_two
        rcs_two = select_two_biggest_connected_region(rcs)
        cs_m_two = select_two_biggest_connected_region(cs_m)

        # 6. selectively merge rs and cs_m with cs_dl -> rcs_selected
        result_dict = self.discontinuity_detection_default(seg_array=cs_m,
                                                           gt_array=None,
                                                           auxiliary_array=rs,
                                                           pre_mask_array=cs_dl,
                                                           spacing=self.spacing,
                                                           origin=self.origin,
                                                           direction=self.direction,
                                                           save_path=None,
                                                           )

        cs_m_selected_sphere = result_dict.get("mask_sphere")
        cs_m_selected_refined_num = result_dict.get("num")
        assert cs_m_selected_sphere is not None, "rcs_selected_sphere is None"
        assert cs_m_selected_refined_num is not None, "rcs_selected_refined_num is None"
        rcs_selected = self.merge(m1=cs_m, m2=rs, dl=cs_m_selected_sphere)

        # 7. select_two_biggest_connected_region on rcs_select -> rcs_selected_two
        rcs_selected_two = select_two_biggest_connected_region(rcs_selected)

        # 10. selectively merge rs and cs_m without cs_dl -> rcs_new
        result_dict = self.discontinuity_detection(seg_array=cs_m,
                                                   gt_array=None,
                                                   auxiliary_array=rs,
                                                   pre_mask_array=None,
                                                   spacing=self.spacing,
                                                   origin=self.origin,
                                                   direction=self.direction,
                                                   save_path=None,
                                                   )

        cs_m_new_sphere = result_dict.get("mask_sphere")
        cs_m_new_refined_num = result_dict.get("num")
        assert cs_m_new_sphere is not None, "rcs_new_sphere is None"
        assert cs_m_new_refined_num is not None, "rcs_new_refined_num is None"
        rcs_new = self.merge(m1=cs_m, m2=rs, dl=cs_m_new_sphere)

        # 11. select_two_biggest_connected_region on rcs_select -> rcs_selected_two
        rcs_new_two = select_two_biggest_connected_region(rcs_new)

        # 13. select_two_biggest_connected_region on arcs_selected -> arcs_selected_two


        # 14. save
        self.save_image(rcs, self.save_subdir / f"{self.name}_RCS.nii.gz")
        self.save_image(rcs_two, self.save_subdir / f"{self.name}_RCS_TWO.nii.gz")
        self.save_image(rcs_selected, self.save_subdir / f"{self.name}_RCS_SELECTED.nii.gz")
        self.save_image(rcs_selected_two, self.save_subdir / f"{self.name}_RCS_SELECTED_TWO.nii.gz")
        self.save_image(cs_m_two, self.save_subdir / f"{self.name}_CS_M_TWO.nii.gz")
        self.save_image(rcs_new, self.save_subdir / f"{self.name}_RCS_NEW.nii.gz")
        self.save_image(rcs_new_two, self.save_subdir / f"{self.name}_RCS_NEW_TWO.nii.gz")
        self.save_image(rs_two, self.save_subdir / f"{self.name}_RS_TWO.nii.gz")

        # save little segment and sphere for visualization
        self.save_image(cs_m_selected_sphere, self.save_subdir / f"{self.name}_cs_m_selected_sphere.nii.gz")
        self.save_image(cs_m_selected_sphere * rs, self.save_subdir / f"{self.name}_rcs_selected_merge_segment.nii.gz")
        self.save_image(cs_m_new_sphere, self.save_subdir / f"{self.name}_cs_m_new_sphere.nii.gz")

        # move cs_a, cs_m, cs_dl, rs, gt to save_subdir for visualization
        shutil.copy(self.cs_m_path, self.save_subdir / f"{self.name}_CS_M.nii.gz")
        shutil.copy(self.cs_dl_path, self.save_subdir / f"{self.name}_CS_DL.nii.gz")
        shutil.copy(self.rs_path, self.save_subdir / f"{self.name}_RS.nii.gz")

        # save discontinuity metric
        d = {
            "name": self.name,
            "cs_m_selected_refined_num": cs_m_selected_refined_num,
            "cs_m_new_refined_num": cs_m_new_refined_num,
        }
        return d


    def save_image(self, image, save_path, save_type=np.uint16):
        sitk_image = sitk.GetImageFromArray(image.astype(save_type))
        sitk_image.SetSpacing(self.spacing)
        sitk_image.SetOrigin(self.origin)
        sitk_image.SetDirection(self.direction)
        sitk.WriteImage(sitk_image, save_path)

    @staticmethod
    def merge(m1, m2, dl):
        m = m1 * (1 - dl) + m2 * dl
        return m

    @staticmethod
```

```
    def read_image(path):
        sitk_image = sitk.ReadImage(path)
        image = sitk.GetArrayFromImage(sitk_image)
        spacing = sitk_image.GetSpacing()
        origin = sitk_image.GetOrigin()
        direction = sitk_image.GetDirection()
        return image, spacing, origin, direction

    @staticmethod
    def selected_final_merge(rcs_select, arcs, dl1, dl2):
        """TODO: selected_final_merge."""
        """
        :param rcs_select: path
        :param arcs: merge cs rs and a
        :param dl1: first stage discontinuity label
        :param dl2: second stage discontinuity label
        :param save_path: save path
        :param save_postfix: save postfix
        """
        if np.sum(dl2) != 0:
            id_map = cc3d.connected_components(dl1, connectivity=26)
            remain_id = np.unique(id_map[dl2 > 0])
            remain_index = np.zeros_like(dl1)
            for id in remain_id:
                if id > 0:
                    remain_index[id_map == id] = 1
            arcs_selected = rcs_select * (1 - remain_index) + arcs * remain_index
        else:
            arcs_selected = rcs_select

        return arcs_selected


class SparrowLinkDiscontinuityDetection:
    def __init__(self,
                 max_broken_size_percentage=0.1,
                 min_sphere_radius_percentage=0.015,
                 sphere_dilation_percentage=0.2,  # 0.1
                 region_select=4,
                 cube_min_size=(10, 10, 10),
                 skeleton_refine_times=3,
                 angle_threshold_1=0.0,
                 angle_threshold_2=0.0,
                 view=False,
                 region_threshold: Union[int, None] = 5,
                 ):
        self.max_broken_size_percentage = max_broken_size_percentage
        self.min_sphere_radius_percentage = min_sphere_radius_percentage
        self.sphere_dilation_percentage = sphere_dilation_percentage
        self.region_select = region_select
        self.cube_min_size = cube_min_size
        self.skeleton_refine_times = skeleton_refine_times
        self.region_threshold = region_threshold
        self.angle_threshold_1 = angle_threshold_1
        self.angle_threshold_2 = angle_threshold_2
        self.view = view
    """
    :param max_broken_size_percentage: the max broken size percentage, used in constriction on distance
    :param min_sphere_radius_percentage: the min sphere radius percentage, used for some small fracture
    :param sphere_dilation_percentage: the sphere dilation percentage, to consider the structure around fracture area
    :param region_select: the region select, matching points consist of start point and paired point, start point is...
    ...constricted in the region with region_select.
    :param cube_min_size: the min size of the cube, used for small fracture
    :param skeleton_refine_times: the refine times of the skeleton, zhang algorithm might cause some small branch in the...
    ...skeleton, so we need to refine the skeleton to delete the small branch
    :param region_threshold: the region threshold, used to delete the small region in the segmentation
    :param angle_threshold_1: the angle threshold 1, used for constriction on orientation
    :param angle_threshold_2: the angle threshold 2, used for constriction on orientation
    """
    def __call__(self,
                 seg_array: np.array = None,
                 gt_array: np.array = None,
                 auxiliary_array: np.array = None,
                 pre_mask_array: np.array = None,
                 spacing: Sequence[float] = None,
                 origin: Sequence[float] = None,
                 direction: Sequence[float] = None,
                 save_path: str = None
                 ):
        result_dict = discontinuity_detection(
            label=seg_array,
            GT=gt_array,
            auxiliary=auxiliary_array,
            pre_mask=pre_mask_array,
            spacing=spacing,
            direction=direction,
            origin=origin,
            save_path=save_path,
            view=self.view,
            max_broken_size_percentage=self.max_broken_size_percentage,
            min_sphere_radius_percentage=self.min_sphere_radius_percentage,
            sphere_dilation_percentage=self.sphere_dilation_percentage,
            region_select=self.region_select,
            cube_min_size=self.cube_min_size,
            skeleton_refine_times=self.skeleton_refine_times,
            region_threshold=self.region_threshold,
            angle_threshold_1=self.angle_threshold_1,
            angle_threshold_2=self.angle_threshold_2,
        )
        return result_dict


def update(pbar, record, result):
    pbar.update()
    record.append(result)


def error_back(err):
    print(err)
```

```python

if __name__ == "__main__":
    import argparse
    import tqdm
    import pathlib
    from multiprocessing import Pool

    import warnings
    warnings.filterwarnings("ignore")  # ignore from np.int16 to np.uint8

    parser = argparse.ArgumentParser()
    parser.add_argument("--CS_M", type=str, default=None)
    parser.add_argument("--CS_DL", type=str, default=None)
    parser.add_argument("--RS", type=str, default=None)
    parser.add_argument("--CS_A", type=str, default=None)
    parser.add_argument("--save_dir", type=str, default=None)
    parser.add_argument("--max_broken_size_percentage", type=float, default=0.3)
    parser.add_argument("--multiprocess", action='store_true', default=False)
    args = parser.parse_args()
    CS_M_path = pathlib.Path(args.CS_M)
    CS_DL_path = pathlib.Path(args.CS_DL)
    RS_path = pathlib.Path(args.RS)
    CS_A_path = pathlib.Path(args.CS_A) if args.CS_A is not None else None
    data_list = list(CS_M_path.glob("*.nii.gz"))
    print(f"\033[96m SparrowLink Postprocessing \033[00m")
    pbar = tqdm.tqdm(total=len(list(data_list)), colour="#87cefa")
    pbar.set_description("SparrowLink Processing")
    process_record = []

    if args.CS_A is not None:
        if args.multiprocess:
            poor = Pool(14)
        for file in data_list:
            name = file.name
            processor = SparrowLinkPostProcess(
                cs_m_path=str(CS_M_path / name),
                cs_dl_path=str(CS_DL_path / name),
                rs_path=str(RS_path / name),
                cs_a_path=str(CS_A_path / name),
                save_path=args.save_dir,
            )
            if not args.multiprocess:
                process_record.append(processor.run())
                pbar.update()
            else:
                poor.apply_async(
                    func=processor.run,
                    callback=lambda x: update(pbar, process_record, x),
                    error_callback=error_back
                )
        if args.multiprocess:
            poor.close()
            poor.join()

        process_record.sort(key=lambda x: x.get("name"))
        with open(args.save_dir + "/process_record.json", "w") as f:
            json.dump(process_record, f, indent=4)

    else:
        if args.multiprocess:
            poor = Pool(14)
        for file in data_list:
            name = file.name
            processor = SparrowLinkPostProcess(
                cs_m_path=str(CS_M_path / name),
                cs_dl_path=str(CS_DL_path / name),
                rs_path=str(RS_path / name),
                cs_a_path=None,
                save_path=args.save_dir,
                max_broken_size_percentage=args.max_broken_size_percentage,
            )
            if not args.multiprocess:
                process_record.append(processor.run_without_cs_a())
                pbar.update()
            else:
                poor.apply_async(
                    func=processor.run_without_cs_a,
                    callback=lambda x: update(pbar, process_record, x),
                    error_callback=error_back
                )
        if args.multiprocess:
            poor.close()
            poor.join()

        process_record.sort(key=lambda x: x.get("name"))
        with open(args.save_dir + "/process_record.json", "w") as f:
            json.dump(process_record, f, indent=4)
```

SparrowLink/transform/Anatomical_augumentation_for_CCTA_images.md

Anatomy-Informed Data Augmentation for Coronary Artery in CCTA Image

* Video of the on-the-fly anatomy-based data augmentation (slice view)

(https://github.com/xxsxxsxxs666/SparrowLink/assets/61532031/5443c459-72b7-480e-8b30-06ff231b956d)

* Video of the on-the-fly anatomy-based data augmentation. (3D)

(https://github.com/xxsxxsxxs666/SparrowLink/assets/61532031/c203e8bf-a892-4227-98cb-060fc3c40671)

* real CCTA image video:

(https://github.com/xxsxxsxxs666/SparrowLink/assets/61532031/c6cdce80-186f-44d0-bcc9-4abb36e9f1ef)

* You can use our anatomy-based data augmentation tool by simply plugging it into MONAI transform architecture:

```python
save_transform = Compose(
        [
            LoadImaged(keys=["image", "label", "heart"]),
            EnsureChannelFirstd(keys=["image", "label", "heart"]),
            ArteryTransformD(keys=["image", "label"], image_key="image", artery_key="label", p_anatomy_per_sample=1,
                             p_contrast_per_sample=1,
                             contrast_reduction_factor_range=(0.6, 1), mask_blur_range=(3, 6),
                             mvf_scale_factor_range=(1, 2), mode=("bilinear", "nearest")),
            # HeartTransformD(keys=["image", "label", "heart"], artery_key="label", heart_key="heart",
            #                 p_anatomy_heart=0, p_anatomy_artery=1,
            #                 dil_ranges=((-10, 10), (-5, -3)), directions_of_trans=((1, 1, 1), (1, 1, 1)), blur=(32, 8),
            #                 mode=("bilinear", "nearest", "nearest"), visualize=True, batch_interpolate=True,
            #                 threshold=(-1, 0.5, 0.5)),
            # CASTransformD(keys=["image", "label", "heart"], label_key="label", heart_key="heart", p_anatomy_per_sample=1,
            #               dil_ranges=((-30, -40), (-300, -500)), directions_of_trans=((1, 1, 1), (1, 1, 1)), blur=[4, 32],
            #              mode=("bilinear", "nearest", "nearest"),),
            SaveImaged(keys=["image"], output_dir=save_dir, output_postfix='spatial_transform_image',
                       print_log=True, padding_mode="zeros"),
            SaveImaged(keys=["label"], output_dir=save_dir, output_postfix='spatial_transform_label',
                       print_log=True, padding_mode="zeros"),
        ]
    )
```

```python
save_transform = Compose(
        [
            LoadImaged(keys=["image", "label", "heart"]),
            EnsureChannelFirstd(keys=["image", "label", "heart"]),
            ArteryTransformD(keys=["image", "label"], image_key="image", artery_key="label", p_anatomy_per_sample=1,
                             p_contrast_per_sample=1,
                             contrast_reduction_factor_range=(0.6, 1), mask_blur_range=(3, 6),
                             mvf_scale_factor_range=(1, 2), mode=("bilinear", "nearest")),
            RandCropByPosNegLabeld(
                keys=["image", "label"],
                label_key="label",
                spatial_size=(128, 128, 128),
                pos=3,
                neg=1,
                num_samples=4,
                image_key="image",
                image_threshold=0,
            ),
            SaveImaged(keys=["image"], output_dir=save_dir, output_postfix='spatial_transform_image',
                       print_log=True, padding_mode="zeros"),
            SaveImaged(keys=["label"], output_dir=save_dir, output_postfix='spatial_transform_label',
                       print_log=True, padding_mode="zeros"),
        ]
    )
```

```python
SparrowLink/transform/AnatomyTransformD.py
import numpy as np
import torch
from monai.utils import TransformBackends, convert_to_tensor, ensure_tuple
from monai.data.meta_obj import get_track_meta
from monai.config import KeysCollection
from monai.transforms import MapTransform, RandomizableTransform, Randomizable, SpatialCrop, TraceableTransform

from typing import Dict, Hashable, List, Mapping, Optional, Sequence, Union, Tuple, Any
from batchgenerators.augmentations.utils import create_zero_centered_coordinate_mesh, elastic_deform_coordinates, \
    interpolate_img, \
    rotate_coords_2d, rotate_coords_3d, scale_coords, resize_segmentation, resize_multichannel_image, \
    elastic_deform_coordinates_2
import warnings
from monai.transforms.utils_pytorch_numpy_unification import unravel_index
from monai.transforms.utils import correct_crop_centers, map_binary_to_indices, convert_to_dst_type, create_translate, \
     ensure_tuple_rep, ensure_tuple
from monai.data.meta_tensor import MetaTensor
from copy import deepcopy
import time
from torch.nn.functional import grid_sample
from scipy.ndimage.filters import gaussian_filter
from skimage.morphology import skeletonize
from post_processing.fracture_detection import get_point_orientation
import cc3d


def get_mvf_by_gaussian_gradient(mask, spacing, blur, dil_magnitude, directions_of_trans, anisotropy_safety):
    shape = mask.shape
    coords = create_zero_centered_coordinate_mesh(shape)
    t, u, v = get_organ_gradient_field(mask,
                                       spacing=spacing,
                                       blur=blur)
    n_factor = np.sqrt(2 * np.pi)
    if directions_of_trans[0]:
        coords[0, :, :, :] = coords[0, :, :, :] + t * blur * dil_magnitude * n_factor
    if directions_of_trans[1]:
        coords[1, :, :, :] = coords[1, :, :, :] + u * blur * dil_magnitude * n_factor
    if directions_of_trans[2]:
        coords[2, :, :, :] = coords[2, :, :, :] + v * blur * dil_magnitude * n_factor
    deformation_record = (t * dil_magnitude * n_factor, u * dil_magnitude * n_factor, v * dil_magnitude * n_factor)
    for d in range(3):
        ctr = shape[d] / 2  # !!!
        coords[d] += ctr - 0.5  # !!!

    if anisotropy_safety:
        coords[0, 0, :, :][coords[0, 0, :, :] < 0] = 0.0
        coords[0, 1, :, :][coords[0, 1, :, :] < 0] = 0.0
        coords[0, -1, :, :][coords[0, -1, :, :] > (shape[-2] - 1)] = shape[-2] - 1
        coords[0, -2, :, :][coords[0, -2, :, :] > (shape[-2] - 1)] = shape[-2] - 1

    return coords, deformation_record


def find_random_one_numpy(image, patch_size=None):
    # 找到值为1的所有点的坐标
    # copy the image to avoid changing the original image
    if patch_size is not None:
        # set image to 0 in the corner
        # copy the image to avoid changing the original image
        image_copy = image.copy()
        image_copy[:patch_size[0] // 2, :, :] = 0
        image_copy[-patch_size[0] // 2:, :, :] = 0
        image_copy[:, :patch_size[1] // 2, :] = 0
        image_copy[:, -patch_size[1] // 2:, :] = 0
        image_copy[:, :, :patch_size[2] // 2] = 0
        image_copy[:, :, -patch_size[2] // 2:] = 0
        ones_indices = np.where(image_copy > 0)
    else:
        ones_indices = np.where(image == 1)
    # 转换坐标为列表形式 [(x1, y1), (x2, y2), ...]
    ones_list = list(zip(ones_indices[0], ones_indices[1], ones_indices[2]))

    if not ones_list:
        return None  # 如果没有找到值为1的点，则返回None

    return ones_list[np.random.randint(len(ones_list))]


def generate_slice_by_center_and_patch_size(center, patch_size):
    x_slice = slice(center[0] - patch_size[0] // 2, center[0] + patch_size[0] // 2)
    y_slice = slice(center[1] - patch_size[1] // 2, center[1] + patch_size[1] // 2)
    z_slice = slice(center[2] - patch_size[2] // 2, center[2] + patch_size[2] // 2)

    return (x_slice, y_slice, z_slice)


def get_organ_gradient_field(organ, spacing=(1, 1, 1), blur=32):
    """
    from batchgenerators.augmentations.utils, but data shape is (H, W, D) instead of (D, H, W)
    The returned MVF is in image coordinates.
    """
    u_ratio = spacing[0] / spacing[1]
    v_ratio = spacing[0] / spacing[2]

    organ_blurred = gaussian_filter(organ.astype(float),
                                    sigma=(blur, blur * u_ratio, blur * v_ratio),
                                    order=0,
                                    mode='nearest')
    t, u, v = np.gradient(organ_blurred)
    t = t
    u = u * u_ratio
    v = v * v_ratio

    return t, u, v


```

```python
def interpolator(data, coords, mode, border_mode=None, border_cval=None, keep_meta=True):
    data = convert_to_tensor(data, track_meta=get_track_meta())
    meta = data.meta.copy()
    data_result = np.zeros_like(data)
    if isinstance(mode, int):
        for channel_id in range(data.shape[0]):
            data_result[channel_id] = interpolate_img(np.array(data[channel_id]), coords, mode,
                                                      border_mode, cval=border_cval)
        data_result = MetaTensor(data_result, meta=meta) if keep_meta else data_result
    else:
        # import time
        # tic = time.time()
        data_batch = data.unsqueeze(0)
        data_result = \
            grid_sample(data_batch, coords, mode=mode, padding_mode="zeros", align_corners=False)[0]
        data_result = MetaTensor(data_result, meta=meta) if keep_meta else data_result
        # toc = time.time()
        # print(f"interpolator time: {toc - tic}")
    return data_result


def coords_numpy2torch(coords, shape, change_coords=True):
    """Prepare for torch's grid-sampling"""
    h, w, d = shape
    if change_coords:
        coords_torch = torch.from_numpy(coords)
    else:
        coords_torch = torch.from_numpy(coords).clone()
    coords_permute = torch.flip(coords_torch, dims=[0]).unsqueeze(0).permute(0, 2, 3, 4, 1).to(torch.float32)
    coords_permute[:, :, :, :, 0] /= (d - 1)
    coords_permute[:, :, :, :, 1] /= (w - 1)
    coords_permute[:, :, :, :, 2] /= (h - 1)
    coords_permute -= 0.5
    coords_permute *= 2
    return coords_permute


def mesh_generator_tensor(patch_size, rand_state, p_el_per_sample: float = 1, p_rot_per_sample: float = 1,
                          p_scale_per_sample: float = 1, do_elastic_deform=True,
                          alpha=(0., 1000.), sigma=(10., 13.), do_rotation=True,
                          angle_x=(0, 2 * np.pi), angle_y=(0, 2 * np.pi), angle_z=(0, 2 * np.pi),
                          p_rot_per_axis: float = 1, do_scale=True, scale=(0.75, 1.25),
                          independent_scale_for_each_axis=False, p_independent_scale_per_axis: int = 1,
                          num_samples: int = 1):
    """TODO: change numpy to torch"""
    dim = len(patch_size)
    coords_list = []
    modified_coords_list = []
    for _ in range(num_samples):
        coords = create_zero_centered_coordinate_mesh(patch_size)
        modified_coords = False
        if do_elastic_deform and rand_state.uniform() < p_el_per_sample:
            a = np.random.uniform(alpha[0], alpha[1])
            s = np.random.uniform(sigma[0], sigma[1])
            coords = elastic_deform_coordinates(coords, a, s)
            modified_coords = True

        if do_rotation and rand_state.uniform() < p_rot_per_sample:

            if np.random.uniform() <= p_rot_per_axis:
                a_x = rand_state.uniform(angle_x[0], angle_x[1])
            else:
                a_x = 0

            if dim == 3:
                if np.random.uniform() <= p_rot_per_axis:
                    a_y = rand_state.uniform(angle_y[0], angle_y[1])
                else:
                    a_y = 0

                if np.random.uniform() <= p_rot_per_axis:
                    a_z = rand_state.uniform(angle_z[0], angle_z[1])
                else:
                    a_z = 0
                coords = rotate_coords_3d(coords, a_x, a_y, a_z)
            else:
                coords = rotate_coords_2d(coords, a_x)
            modified_coords = True
        if do_scale and rand_state.uniform() < p_scale_per_sample:
            if independent_scale_for_each_axis and rand_state.uniform() < p_independent_scale_per_axis:
                sc = []
                for _ in range(dim):
                    if rand_state.random() < 0.5 and scale[0] < 1:
                        sc.append(rand_state.uniform(scale[0], 1))
                    else:
                        sc.append(rand_state.uniform(max(scale[0], 1), scale[1]))
            else:
                if rand_state.random() < 0.5 and scale[0] < 1:
                    sc = rand_state.uniform(scale[0], 1)
                else:
                    sc = rand_state.uniform(max(scale[0], 1), scale[1])

            coords = scale_coords(coords, sc)
            modified_coords = True
        coords_list.append(coords)
        modified_coords_list.append(modified_coords)

    return coords_list, modified_coords_list


def generate_pos_neg_label_crop_centers(
        label,
        spatial_size: Union[Sequence[int], int],
        num_samples: int,
        pos_ratio: float,
        label_spatial_shape: Sequence[int],
        fg_indices = None,
        bg_indices = None,
        rand_state: Optional[np.random.RandomState] = None
```

```
        rand_state: Optional[np.random.RandomState] = None,
        allow_smaller: bool = False,
) -> List[List[int]]:
    """
    Generate valid sample locations based on the label with option for specifying foreground ratio
    Valid: samples sitting entirely within image, expected input shape: [C, H, W, D] or [C, H, W]

    Args:
        label: used gof generating coordinates
        image: for checking correction of cropping
        spatial_size: spatial size of the ROIs to be sampled.
        num_samples: total sample centers to be generated.
        pos_ratio: ratio of total locations generated that have center being foreground.
        label_spatial_shape: spatial shape of the original label data to unravel selected centers.
        fg_indices: pre-computed foreground indices in 1 dimension.
        bg_indices: pre-computed background indices in 1 dimension.
        rand_state: numpy randomState object to align with other modules.
        allow_smaller: if `False`, an exception will be raised if the image is smaller than
            the requested ROI in any dimension. If `True`, any smaller dimensions will be set to
            match the cropped size (i.e., no cropping in that dimension).

    Raises:
        ValueError: When the proposed roi is larger than the image.
        ValueError: When the foreground and background indices lengths are 0.

    """
    if rand_state is None:
        rand_state = np.random.random.__self__  # type: ignore

    if fg_indices is None or bg_indices is None:
        fg_indices, bg_indices = map_binary_to_indices(label, image=None)

    centers = []
    fg_indices = np.asarray(fg_indices) if isinstance(fg_indices, Sequence) else fg_indices
    bg_indices = np.asarray(bg_indices) if isinstance(bg_indices, Sequence) else bg_indices
    if len(fg_indices) == 0 and len(bg_indices) == 0:
        raise ValueError("No sampling location available.")

    if len(fg_indices) == 0 or len(bg_indices) == 0:
        pos_ratio = 0 if len(fg_indices) == 0 else 1
        warnings.warn(
            f"Num foregrounds {len(fg_indices)}, Num backgrounds {len(bg_indices)}, "
            f"unable to generate class balanced samples, setting `pos_ratio` to {pos_ratio}."
        )

    for _ in range(num_samples):
        indices_to_use = fg_indices if rand_state.rand() < pos_ratio else bg_indices
        random_int = rand_state.randint(len(indices_to_use))
        idx = indices_to_use[random_int]
        center = unravel_index(idx, label_spatial_shape).tolist()
        # shift center to range of valid centers
        centers.append(correct_crop_centers(center, spatial_size, label_spatial_shape, allow_smaller))

    return centers


class CASTransformD(Randomizable, MapTransform):
    """
    Add three components in deformable transformation
    1. Muscle squeezing and muscle relaxation -> based on vessel segmentation, using normal vector of the surface.
    2. Heart motion -> based on chambers segmentation
    3. Cardiac Motion or curve deformation -> based on centerline of segmentation
    Args:
    `dil_ranges`: dilation range per organs
    `modalities`: on which input channels should the transformation be applied
    `directions_of_trans`: to which directions should the organs be dilated per organs
    `p_per_sample`: probability of the transformation per organs
    `spacing_ratio`: ratio of the transversal plane spacing and the slice thickness, in our case it was 0.3125/3
    `blur`: Gaussian kernel parameter, we used the value 32 for 0.3125mm transversal plane spacing
    `anisotropy_safety`: it provides a certain protection against transformation artifacts in 2 slices from the image border
    `max_annotation_value`: the value that should be still relevant for the main task
    `replace_value`: segmentation values larger than the `max_annotation_value` will be replaced with
    """
    backend = [TransformBackends.NUMPY, TransformBackends.TORCH]

    def __init__(self,
                 keys: KeysCollection,
                 label_key: str = 'label',
                 heart_key: str = 'heart',
                 p_anatomy_per_sample: float = 0.5,
                 dil_ranges: Tuple[Tuple[int, int], Tuple[int, int]] = ((0, 0), (0, 0)),
                 directions_of_trans: Tuple[Tuple[int, int, int], Tuple[int, int, int]] = ((1, 1, 1), (1, 1, 1)),
                 spacing_ratio: float = 0.334/0.5,
                 blur: List = [32, 32],
                 anisotropy_safety: bool = True,
                 max_annotation_value: int = 1,
                 allow_missing_keys: bool = False,
                 mode: Union[Sequence[int], int, Sequence[str], str] = 1,
                 border_mode: str = 'constant',
                 cval: float = 0.0,):
        MapTransform.__init__(self, keys, allow_missing_keys)
        self.label_key = label_key
        self.heart_key = heart_key
        self.p_anatomy = p_anatomy_per_sample
        self.dilation_ranges = dil_ranges
        self.directions_of_trans = directions_of_trans
        self.spacing_ratio = spacing_ratio
        self.blur = blur
        self.anisotropy_safety = anisotropy_safety
        self.max_annotation_value = max_annotation_value
        self.border_mode = ensure_tuple_rep(border_mode, len(self.keys))
        self.border_cval = ensure_tuple_rep(cval, len(self.keys))
        self.mode = ensure_tuple_rep(mode, len(self.keys))

    def __call__(self, data: Mapping[Hashable, torch.Tensor]) -> Dict[Hashable, torch.Tensor]:
        """TODO: if num_samples > 1, this function uses for loop to interpolate the data, which is not efficient."""
        d: Dict = dict(data)
        mask = [d[self.label_key],]
        if self.heart_key is not None:
```

```
            mask.append(d[self.heart_key])

        m, deformation_record = self.randomize(mask=mask)

        deformation_shape = list(d[self.label_key].shape)
        deformation_shape[0] = 3
        meta = d[self.label_key].meta.copy()
        label_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
        label_deformation_field[0, :, :, :] = torch.tensor(deformation_record[0][0])
        label_deformation_field[1, :, :, :] = torch.tensor(deformation_record[0][1])
        label_deformation_field[2, :, :, :] = torch.tensor(deformation_record[0][2])
        d['label_df'] = MetaTensor(label_deformation_field, meta=meta)
        if self.heart_key is not None:
            heart_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
            heart_deformation_field[0, :, :, :] = torch.tensor(deformation_record[1][0])
            heart_deformation_field[1, :, :, :] = torch.tensor(deformation_record[1][1])
            heart_deformation_field[2, :, :, :] = torch.tensor(deformation_record[1][2])
            d['heart_df'] = MetaTensor(heart_deformation_field, meta=meta)
            heart_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
            heart_deformation_field[0, :, :, :] = torch.tensor(deformation_record[2][0])
            heart_deformation_field[1, :, :, :] = torch.tensor(deformation_record[2][1])
            heart_deformation_field[2, :, :, :] = torch.tensor(deformation_record[2][2])
            d['heart_df1'] = MetaTensor(heart_deformation_field, meta=meta)
            heart_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
            heart_deformation_field[0, :, :, :] = torch.tensor(deformation_record[3][0])
            heart_deformation_field[1, :, :, :] = torch.tensor(deformation_record[3][1])
            heart deformation_field[2, :, :, :] = torch.tensor(deformation_record[3][2])
            d['heart_df2'] = MetaTensor(heart_deformation_field, meta=meta)
            heart_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
            heart_deformation_field[0, :, :, :] = torch.tensor(deformation_record[4][0])
            heart_deformation_field[1, :, :, :] = torch.tensor(deformation_record[4][1])
            heart_deformation_field[2, :, :, :] = torch.tensor(deformation_record[4][2])
            d['heart_df3'] = MetaTensor(heart_deformation_field, meta=meta)
            heart_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
            heart_deformation_field[0, :, :, :] = torch.tensor(deformation_record[5][0])
            heart_deformation_field[1, :, :, :] = torch.tensor(deformation_record[5][1])
            heart_deformation_field[2, :, :, :] = torch.tensor(deformation_record[5][2])
            d['heart_df4'] = MetaTensor(heart_deformation_field, meta=meta)
            heart_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
            heart_deformation_field[0, :, :, :] = torch.tensor(deformation_record[6][0])
            heart_deformation_field[1, :, :, :] = torch.tensor(deformation_record[6][1])
            heart_deformation_field[2, :, :, :] = torch.tensor(deformation_record[6][2])
            d['heart_df5'] = MetaTensor(heart_deformation_field, meta=meta)
            heart_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
            heart_deformation_field[0, :, :, :] = torch.tensor(deformation_record[7][0])
            heart_deformation_field[1, :, :, :] = torch.tensor(deformation_record[7][1])
            heart_deformation_field[2, :, :, :] = torch.tensor(deformation_record[7][2])
            d['heart_df6'] = MetaTensor(heart_deformation_field, meta=meta)

        for key, border_mode, border_cval, order in self.key_iterator(d, self.border_mode, self.border_cval, self.mode):
            d[key] = self.interpolator(data=d[key], coords=m, border_mode=border_mode, border_cval=border_cval,
                                       mode=order,)

        return d

    def randomize(self,
                  mask: List = None,
                  spacing=None,):
        """
        return coordinate mesh and dilation magnitude
        """
        deformation_record = []
        if self.R.uniform() < self.p_anatomy:
            shape = mask[0].shape
            coords = create_zero_centered_coordinate_mesh(shape[1:])
            if spacing is not None:
                # TODO: spacing in 1, 2 dimension may be different
                spacing_ratio = spacing[0] / spacing[2]
            else:
                spacing_ratio = self.spacing_ratio

            for i in range(len(mask)):
                dil_magnitude = np.random.uniform(low=self.dilation_ranges[i][0], high=self.dilation_ranges[i][1])
                if i == 0:
                    t, u, v = get_organ_gradient_field(mask[i][0] > 0,
                                                       spacing_ratio=spacing_ratio,
                                                       blur=self.blur[i])
                elif i == 1:
                    t, u, v = get_organ_gradient_field(mask[i][0] > 0,
                                                       spacing_ratio=spacing_ratio,
                                                       blur=self.blur[i])

                    t_1, u_1, v_1 = get_organ_gradient_field(mask[i][0] == 1,
                                                             spacing_ratio=spacing_ratio,
                                                             blur=self.blur[i])
                    t_2, u_2, v_2 = get_organ_gradient_field(mask[i][0] == 2,
                                                             spacing_ratio=spacing_ratio,
                                                             blur=self.blur[i])
                    t_3, u_3, v_3 = get_organ_gradient_field(mask[i][0] == 3,
                                                             spacing_ratio=spacing_ratio,
                                                             blur=self.blur[i])
                    t_4, u_4, v_4 = get_organ_gradient_field(mask[i][0] == 4,
                                                             spacing_ratio=spacing_ratio,
                                                             blur=self.blur[i])
                    t_5, u_5, v_5 = get_organ_gradient_field(mask[i][0] == 5,
                                                             spacing_ratio=spacing_ratio,
                                                             blur=self.blur[i])
                    t_6, u_6, v_6 = get_organ_gradient_field(mask[i][0] == 6,
                                                             spacing_ratio=spacing_ratio,
                                                             blur=self.blur[i])


                sigma = self.blur[i]
                if self.directions_of_trans[i][0]:
                    coords[0, :, :, :] = coords[0, :, :, :] + t * (sigma ** 2) * dil_magnitude
                if self.directions_of_trans[i][1]:
                    coords[1, :, :, :] = coords[1, :, :, :] + u * (sigma ** 2) * dil_magnitude
                if self.directions_of_trans[i][2]:
```

```
                if self.directions_of_trans[1][2]:
                    coords[2, :, :, :] = coords[2, :, :, :] + v * (sigma ** 2) * dil_magnitude

                deformation_record.append((t * dil_magnitude, u * dil_magnitude, v * dil_magnitude)) #  * spacing_ratio))
                if i == 1:
                    deformation_record.append((t_1 * dil_magnitude, u_1 * dil_magnitude, v_1 * dil_magnitude))
                    deformation_record.append((t_2 * dil_magnitude, u_2 * dil_magnitude, v_2 * dil_magnitude))
                    deformation_record.append((t_3 * dil_magnitude, u_3 * dil_magnitude, v_3 * dil_magnitude))
                    deformation_record.append((t_4 * dil_magnitude, u_4 * dil_magnitude, v_4 * dil_magnitude))
                    deformation_record.append((t_5 * dil_magnitude, u_5 * dil_magnitude, v_5 * dil_magnitude))
                    deformation_record.append((t_6 * dil_magnitude, u_6 * dil_magnitude, v_6 * dil_magnitude))

            for d in range(3):
                ctr = shape[d + 1] / 2  # !!!
                coords[d] += ctr - 0.5  # !!!

            if self.anisotropy_safety:
                coords[0, 0, :, :][coords[0, 0, :, :] < 0] = 0.0
                coords[0, 1, :, :][coords[0, 1, :, :] < 0] = 0.0
                coords[0, -1, :, :][coords[0, -1, :, :] > (shape[-3] - 1)] = shape[-3] - 1
                coords[0, -2, :, :][coords[0, -2, :, :] > (shape[-3] - 1)] = shape[-3] - 1
        else:
            coords = None

        return coords, deformation_record

    def interpolator(self, data, coords, mode, border_mode=None, border_cval=None):
        data = convert_to_tensor(data, track_meta=get_track_meta())
        meta = data.meta.copy()
        data_result = np.zeros_like(data)
        if isinstance(mode, int):
            for channel_id in range(data.shape[0]):
                data_result[channel_id] = interpolate_img(np.array(data[channel_id]), coords, mode,
                                                          border_mode, cval=border_cval)
            data_result = MetaTensor(data_result, meta=meta)

        else:
            h, w, d = data.shape[1], data.shape[2], data.shape[3]
            coords = coords[::-1, :, :, :].copy()  # xyz -> zyx
            coords_permute = torch.from_numpy(coords).unsqueeze(0).permute(0, 2, 3, 4, 1).to(torch.float32)
            data_batch = data.unsqueeze(0)
            # coords_norm = (coords_permute + 1) / torch.tensor([h, w, d], dtype=torch.float32).reshape(1, 1, 1, 1, 3)
            coords_norm = torch.zeros_like(coords_permute)
            coords_norm[:, :, :, :, 0], coords_norm[:, :, :, :, 1], coords_norm[:, :, :, :, 2] = \
                (coords_permute[:, :, :, :, 0] + 1) / d, (coords_permute[:, :, :, :, 1] + 1) / w, \
                (coords_permute[:, :, :, :, 2] + 1) / h
            coords_norm = (coords_norm - 0.5) * 2
            data_result = \
                grid_sample(data_batch, coords_norm, mode=mode, padding_mode="zeros", align_corners=False)[0]
            data_result = MetaTensor(data_result, meta=meta)
        return data_result


class HeartTransformD(Randomizable, MapTransform):
    """
    Add three components in deformable transformation
    1. Cardiac muscle squeezing and muscle relaxation -> based on vessel segmentation, using normal vector of the surface.
    Args:
    `dil_ranges`: dilation range per organs
    `modalities`: on which input channels should the transformation be applied
    `directions_of_trans`: to which directions should the organs be dilated per organs
    `p_per_sample`: probability of the transformation per organs
    `spacing_ratio`: ratio of the transversal plane spacing and the slice thickness, in our case it was 0.3125/3
    `blur`: Gaussian kernel parameter, we used the value 32 for 0.3125mm transversal plane spacing
    `anisotropy_safety`: it provides a certain protection against transformation artifacts in 2 slices from the image border
    `max_annotation_value`: the value that should be still relevant for the main task
    `replace_value`: segmentation values larger than the `max_annotation_value` will be replaced with
    'heart_select' : select which chamber to do dilation and shrink, In this work [1, 2, 3, 4, 5, 6] represent
    [inner left-aorta, left-ventricular, right-aorta, right-ventricular, main-artery, left-aorta]
    'heart_key' : heart segmentation
    'label_key' : coronary artery segmentation
    """
    backend = [TransformBackends.NUMPY, TransformBackends.TORCH]

    def __init__(self,
                 keys: KeysCollection,
                 heart_key: str = 'heart',
                 artery_key: str = None,
                 heart_select: Tuple[Union[int, Tuple[int]]] = ((5, ), (2, 4), (3, 6)),
                 p_anatomy_heart: float = 0.5,
                 p_anatomy_artery: float = 0.5,
                 dil_ranges: Tuple[Tuple[float, float], Tuple[float, float]] = ((0, 0), (0, 0)),
                 directions_of_trans: Tuple[Tuple[int, int, int], Tuple[int, int, int]] = ((1, 1, 1), (1, 1, 1)),
                 blur: Tuple = (32, 32),
                 anisotropy_safety: bool = True,
                 max_annotation_value: int = 1,
                 allow_missing_keys: bool = False,
                 mode: Union[Sequence[int], int, Sequence[str], str] = 1,
                 batch_interpolate: bool = False,
                 threshold: Tuple = None,
                 border_mode: str = 'constant',
                 cval: float = 0.0,
                 visualize: bool = False,
                 del_heart: bool = True,
                 ):
        MapTransform.__init__(self, keys, allow_missing_keys)
        self.heart_key = heart_key
        self.heart_select = heart_select
        self.artery_key = artery_key
        self.p_anatomy_heart = p_anatomy_heart
        self.p_anatomy_artery = p_anatomy_artery
        self.dilation_ranges = dil_ranges
        self.directions_of_trans = directions_of_trans
        self.blur = blur
        self.anisotropy_safety = anisotropy_safety
        self.max_annotation_value = max_annotation_value
        self.border_mode = ensure_tuple_rep(border_mode, len(self.keys))
        self.border_cval = ensure_tuple_rep(cval, len(self.keys))
        self.mode = ensure_tuple_rep(mode, len(self.keys))
```

```
        self.visualize = visualize

        self.do_heart_transformation = False
        self.do_artery_transformation = False
        self.dil_magnitude_heart = 0
        self.dil_magnitude_artery = 0
        self.random_index = None  # random index to select chambers
        self.threshold = ensure_tuple_rep(threshold, len(self.keys))
        self.batch_interpolate = batch_interpolate

        self.del_heart = del_heart

    def __call__(self, data: Mapping[Hashable, torch.Tensor]) -> Dict[Hashable, torch.Tensor]:
        """TODO: if num_samples > 1, this function uses for loop to interpolate the data, which is not efficient."""
        d: Dict = dict(data)
        spacing = d[self.heart_key].meta['pixdim'][1:4]
        shape = d[self.heart_key].shape
        self.randomize()
        if self.do_heart_transformation:
            mask = [d[self.heart_key]]
            mask_dilation = torch.zeros_like(mask[0][0])
            for idx in self.random_index:
                mask_dilation += mask[0][0] == idx

            m, deformation_heart = get_mvf_by_gaussian_gradient(mask=mask_dilation > 0,
                                                                spacing=spacing,
                                                                blur=self.blur[0],
                                                                dil_magnitude=self.dil_magnitude_heart,
                                                                directions_of_trans=self.directions_of_trans[0],
                                                                anisotropy_safety=self.anisotropy_safety)
            if not isinstance(self.mode[0], int):  # assert all modes are the same, either int or str
                m = coords_numpy2torch(m, shape[1:], change_coords=True)
            if self.batch_interpolate:
                # only support torch
                self.batch_interpolator(d=d, m=m, shape=shape[1:])
            else:
                for key, border_mode, border_cval, order in \
                        self.key_iterator(d, self.border_mode, self.border_cval, self.mode):
                    # use torch's grid-sampling to interpolate the data
                    d[key] = interpolator(data=d[key], coords=m, border_mode=border_mode, border_cval=border_cval,
                                          mode=order, )

            if self.visualize:
                deformation_shape = list(d[self.heart_key].shape)
                deformation_shape[0] = 3
                meta = d[self.heart_key].meta.copy()
                heart_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
                #  change to world coordinates
                heart_deformation_field[0, :, :, :] = torch.tensor(deformation_heart[0]) * spacing[0]
                heart_deformation_field[1, :, :, :] = torch.tensor(deformation_heart[1]) * spacing[1]
                heart_deformation_field[2, :, :, :] = torch.tensor(deformation_heart[2]) * spacing[2]
                d['heart_df'] = MetaTensor(heart_deformation_field, meta=meta)

        if self.do_artery_transformation:
            m, deformation_artery = get_mvf_by_gaussian_gradient(mask=d[self.artery_key][0] > 0,
                                                                 spacing=spacing,
                                                                 blur=self.blur[1],
                                                                 dil_magnitude=self.dil_magnitude_artery,
                                                                 directions_of_trans=self.directions_of_trans[1],
                                                                 anisotropy_safety=self.anisotropy_safety)
            if not isinstance(self.mode[0], int):  # assert all modes are the same, either int or str
                m = coords_numpy2torch(m, shape[1:], change_coords=True)

            if self.batch_interpolate:
                self.batch_interpolator(d=d, m=m, shape=shape[1:])
            else:
                for key, border_mode, border_cval, order in \
                        self.key_iterator(d, self.border_mode, self.border_cval, self.mode):
                    d[key] = interpolator(data=d[key], coords=m, border_mode=border_mode, border_cval=border_cval,
                                          mode=order, )

            if self.visualize:
                deformation_shape = list(d[self.artery_key].shape)
                deformation_shape[0] = 3
                meta = d[self.artery_key].meta.copy()
                label_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
                label_deformation_field[0, :, :, :] = torch.tensor(deformation_artery[0]) * spacing[0]
                label_deformation_field[1, :, :, :] = torch.tensor(deformation_artery[1]) * spacing[1]
                label_deformation_field[2, :, :, :] = torch.tensor(deformation_artery[2]) * spacing[2]
                d['label_df'] = MetaTensor(label_deformation_field, meta=meta)

        if self.del_heart:
            d.pop(self.heart_key)  # delete heart segmentation to save memory
        return d

    def batch_interpolator(self, d, m, shape, device="cpu"):
        i_h, i_w, i_d = shape
        c = len(self.mode)
        batch_data = torch.zeros((1, c, i_h, i_w, i_d))
        meta_dict = {}

        for i, key in enumerate(self.key_iterator(d)):
            batch_data[0][i] = d[key]
            meta_dict[key] = d[key].meta

        with torch.no_grad():
            data_result = \
                grid_sample(batch_data.to(device), m.to(device), mode=self.mode[0], padding_mode="zeros",
                            align_corners=False)[0]

        for key, t, i in self.key_iterator(d, self.threshold, range(c)):
            if t > 0:
                d[key] = MetaTensor((data_result[i] > t).unsqueeze(0),
                                    meta=meta_dict[key])
            else:
                d[key] = MetaTensor(data_result[i].unsqueeze(0), meta=meta_dict[key])
```

```
    def randomize(self, data: Any = None):
        """ return coordinate mesh and dilation magnitude"""
        if self.R.uniform() < self.p_anatomy_heart:
            self.do_heart_transformation = True
            self.dil_magnitude_heart = self.R.uniform(low=9, high=self.dilation_ranges[0][1])
            self.random_index = self.heart_select[self.R.randint(0, len(self.heart_select))]
        else:
            self.do_heart_transformation = False

        if self.artery_key is not None:
            if self.R.uniform() < self.p_anatomy_artery:
                self.do_artery_transformation = True
                self.dil_magnitude_artery = self.R.uniform(low=self.dilation_ranges[1][0],
                                                           high=self.dilation_ranges[1][1])
                print(f"artery dilation magnitude: {self.dil_magnitude_artery}")
        else:
            self.do_artery_transformation = False

        #     shape = mask[0].shape
        #     coords = create_zero_centered_coordinate_mesh(shape[1:])
        #     random_index = self.heart_select[np.random.randint(0, len(self.heart_select))]
        #     mask_dilation = torch.zeros_like(mask[0][0])
        #     for idx in random_index:
        #         mask_dilation += mask[0][0] == idx
        #     t_h, u_h, v_h = get_organ_gradient_field(mask_dilation > 0,
        #                                              spacing=spacing,
        #                                              blur=self.blur[0])
        #     dil_magnitude_heart = np.random.uniform(low=self.dilation_ranges[0][0], high=self.dilation_ranges[0][1])
        #     dil_magnitude_artery = None
        #     t_a, u_a, v_a = None, None, None
        #
        #     if self.artery_key is not None:
        #         assert len(mask) > 0 and len(self.blur) > 0 and len(self.dilation_ranges) > 0, \
        #             "artery label is not available or parameters are not set"
        #         dil_magnitude_artery = np.random.uniform(low=self.dilation_ranges[0][0], high=self.dilation_ranges[0][1])
        #         t_a, u_a, v_a = get_organ_gradient_field(mask[1][0] > 0,
        #                                                  spacing=spacing,
        #                                                  blur=self.blur[1])
        #
        #     n_factor = np.sqrt(2 * np.pi)
        #     if self.directions_of_trans[0][0]:
        #         coords[0, :, :, :] = coords[0, :, :, :] + t_h * self.blur[0] * dil_magnitude_heart * n_factor
        #     if self.directions_of_trans[0][1]:
        #         coords[1, :, :, :] = coords[1, :, :, :] + u_h * self.blur[0] * dil_magnitude_heart * n_factor
        #     if self.directions_of_trans[0][2]:
        #         coords[2, :, :, :] = coords[2, :, :, :] + v_h * self.blur[0] * dil_magnitude_heart * n_factor
        #
        #     deformation_record.append((t_h * dil_magnitude_heart * n_factor, u_h * dil_magnitude_heart * n_factor,
        #                                v_h * dil_magnitude_heart * n_factor))
        #     if self.artery_key is not None:
        #         if self.directions_of_trans[0][0]:
        #             coords[0, :, :, :] = coords[0, :, :, :] + t_a * self.blur[1] * dil_magnitude_artery * n_factor
        #         if self.directions_of_trans[0][1]:
        #             coords[1, :, :, :] = coords[1, :, :, :] + u_a * self.blur[1] * dil_magnitude_artery * n_factor
        #         if self.directions_of_trans[0][2]:
        #             coords[2, :, :, :] = coords[2, :, :, :] + v_a * self.blur[1] * dil_magnitude_artery * n_factor
        #         deformation_record.append(
        #             (t_a * dil_magnitude_artery * n_factor, u_a * dil_magnitude_artery * n_factor,
        #              v_a * dil_magnitude_artery * n_factor))
        #
        #     for d in range(3):
        #         ctr = shape[d + 1] / 2  # !!!
        #         coords[d] += ctr - 0.5  # !!!
        #
        #     if self.anisotropy_safety:
        #         coords[0, 0, :, :][coords[0, 0, :, :] < 0] = 0.0
        #         coords[0, 1, :, :][coords[0, 1, :, :] < 0] = 0.0
        #         coords[0, -1, :, :][coords[0, -1, :, :] > (shape[-3] - 1)] = shape[-3] - 1
        #         coords[0, -2, :, :][coords[0, -2, :, :] > (shape[-3] - 1)] = shape[-3] - 1
        # else:
        #     coords = None
        #
        # return coords, deformation_record


def adjust_contrast(patch_image, contrast_reduction_factor, patch_patch_slice, mask_blur=4):
    # generate weighted mask
    patch_patch_mask = np.ones_like(patch_image)
    patch_patch_mask[patch_patch_slice] = contrast_reduction_factor
    patch_patch_mask_gaussian = gaussian_filter(patch_patch_mask, sigma=mask_blur)
    min_max = patch_patch_mask_gaussian.max() - patch_patch_mask_gaussian.min()
    assert min_max > 0.00001, f"min_max should be larger than 0, {patch_patch_mask_gaussian.max(), patch_patch_mask_gaussian.min()}" \
                              f"{patch_patch_mask.max(), patch_patch_mask.min()}" \
                              f"{contrast_reduction_factor}" \
                              f"{patch_patch_slice}"
    patch_min = patch_patch_mask_gaussian.min()
    patch_patch_mask_gaussian -= patch_min
    patch_patch_mask_gaussian *= ((1 - contrast_reduction_factor) / min_max)
    patch_patch_mask_gaussian += contrast_reduction_factor
    # patch_patch_mask_gaussian = (patch_patch_mask_gaussian - patch_min) * (1 - contrast_reduction_factor) / min_max \
    #                             + contrast_reduction_factor

    patch_mean = patch_image.mean()
    patch_image = (patch_image - patch_mean) * patch_patch_mask_gaussian + patch_mean
    return torch.from_numpy(patch_image)


def check_shape(a, b):
    flag = True
    for i, j in zip(a, b):
        if i > j:
            flag = False
    return flag


def scale_tuple(a, t):
    return tuple([a[i]*t for i in range(len(a))])
```

```python

def generate_random_vector_perpendicular_to(n):
    n = n / np.linalg.norm(n)

    # 生成一个随机向量
    r = np.random.rand(3)

    # 计算垂直于n的向量
    v = np.cross(n, r)

    # 如果v是零向量（非常罕见的情况，但理论上可能如果r和n平行），重新生成r
    while np.linalg.norm(v) == 0:
        r = np.random.rand(3)
        v = np.cross(n, r)
    # 标准化v
    v = v / np.linalg.norm(v)

    return v


def get_sphere_by_center_and_radius(center, radius, spacing, shape):
    """
    center: sphere center (h, w, d)
    radius: sphere radius, mm
    spacing: voxel spacing, mm
    shape: image shape or patch shape
    """
    h, w, d = shape
    tmp = tuple([np.arange(i) for i in (h, w, d)])
    coords = np.array(np.meshgrid(*tmp, indexing='ij')).astype(float)
    coords -= np.array(center)[:, None, None, None]
    coords = coords * np.array(spacing)[:, None, None, None]
    distance = np.linalg.norm(coords, axis=0)
    sphere = np.zeros((h, w, d))
    sphere[distance <= radius] = 1.0
    return sphere


def generate_mvf_vector(center, artery, centerline, direction_image, spacing, shape, scale_factor, radius, sigma=4):
    orientation_centerline_spacing = \
        get_point_orientation(point=center, centerline=centerline, spacing=spacing, size=3,
                              is_end_point=False)[0]
    if orientation_centerline_spacing is None:
        return None
    direction_image_com = np.array([direction_image[0], direction_image[4], direction_image[8]])
    orientation_centerline_world = orientation_centerline_spacing * direction_image_com
    random_direction_3d = generate_random_vector_perpendicular_to(orientation_centerline_world)
    h, w, d = shape
    mvf_world = np.zeros((h, w, d, 3))
    mvf_world += random_direction_3d
    center_point = center[0]

    mvf_mask = get_sphere_by_center_and_radius(center=center_point, radius=radius, spacing=spacing, shape=shape)
    mvf_mask = mvf_mask * (artery > 0)
    connected_region = cc3d.connected_components(mvf_mask)  # only select the region where center locates
    area_id = connected_region[center_point[0], center_point[1], center_point[2]]
    mvf_mask = mvf_mask * (connected_region == area_id)
    mvf_weight = gaussian_filter(mvf_mask, sigma=sigma) * scale_factor * sigma

    mvf_world = mvf_world * mvf_weight[..., None]
    mvf_image = (mvf_world / np.array(direction_image_com) / np.array(spacing))
    h, w, d = shape
    tmp = tuple([np.arange(i) for i in (h, w, d)])
    coords = np.array(np.meshgrid(*tmp, indexing='ij')).astype(float)
    return coords + mvf_image.transpose(3, 0, 1, 2)


class ArteryTransformD(Randomizable, MapTransform):
    """
    1. local contrast change
    2. artery shift
    3. local vessel shrink or dilation
    """
    backend = [TransformBackends.NUMPY, TransformBackends.TORCH]

    def __init__(self,
                 keys: KeysCollection,
                 image_key: str = None,
                 contrast_patch_patch_size_range: Union[Tuple, Tuple[Tuple]] = ((10, 30), (10, 30), (10, 30)),
                 deform_patch_patch_size_range: Tuple = (5.0, 10.0),
                 contrast_reduction_factor_range: Tuple = (0.6, 1),
                 mvf_scale_factor_range: Tuple = (-2, 2),
                 mask_blur_range: Tuple = (3, 6),
                 artery_key: str = None,
                 centerline_key: str = None,
                 allow_missing_keys: bool = False,
                 p_anatomy_per_sample: float = 0.0,
                 p_contrast_per_sample: float = 0.0,
                 mode: Union[Sequence[int], int, Sequence[str], str] = 1,
                 border_mode: str = 'constant',
                 cval: float = 0.0,
                 visualize: bool = False):
        MapTransform.__init__(self, keys, allow_missing_keys)
        self.artery_key = artery_key
        self.image_key = image_key
        self.centerline_key = centerline_key
        self.p_anatomy = p_anatomy_per_sample
        self.p_contrast = p_contrast_per_sample
        self.border_mode = ensure_tuple_rep(border_mode, len(self.keys))
        self.border_cval = ensure_tuple_rep(cval, len(self.keys))
        self.mode = ensure_tuple_rep(mode, len(self.keys))
        self.visualize = visualize
        self.do_artery_deformation = False
        self.do_local_contrast_change = False
        self.contrast_reduction_factor_range = contrast_reduction_factor_range
        self.contrast_reduction_factor = None
        self.mask_blur_range = mask_blur_range
        self.mask_blur = None
```

```
        self.contrast_patch_patch_size_range = ensure_tuple_rep(contrast_patch_patch_size_range, 3)
        self.deform_patch_patch_size_range = deform_patch_patch_size_range
        self.mvf_mask_radius = None
        self.patch_patch_size_x_1, self.patch_patch_size_y_1, self.patch_patch_size_z_1 = None, None, None
        self.mvf_scale_factor_range = mvf_scale_factor_range
        self.mvf_scale_factor = None

    def __call__(self, data: Mapping[Hashable, torch.Tensor]) -> Dict[Hashable, torch.Tensor]:
        """TODO: if num_samples > 1, this function uses for loop to interpolate the data, which is not efficient."""
        d: Dict = dict(data)
        # check whether this patch has label
        self.randomize()
        if self.do_artery_deformation or self.do_local_contrast_change:
            artery = d[self.artery_key]
            shape = artery.shape[1:]
            original_affine = artery.meta.get('original_affine')
            direction_image = np.sign(original_affine[:3, :3].reshape(-1, )).tolist()
            # spacing = artery.meta.get('pixdim')[1:4]
            r_matrix = artery.meta.get('affine')[:3, :3].reshape(-1, ).tolist()
            spacing = np.array([np.abs(r_matrix[0]), np.abs(r_matrix[4]), np.abs(r_matrix[8])], dtype=np.float32)
            origin = original_affine[:3, 3]
            if artery.sum() < 100:  # if no artery, return directly. We assume the artery should be larger than 100
                return d
            if self.centerline_key is not None:
                """get centerline"""
                centerline = data[self.centerline_key]
            else:
                centerline = skeletonize(d[self.artery_key][0] > 0)[None, ]
        else:
            return d
        if self.do_local_contrast_change:
            patch_patch_size = (self.patch_patch_size_x_1, self.patch_patch_size_y_1, self.patch_patch_size_z_1)
            center = np.array(find_random_one_numpy(centerline[0], patch_size=patch_patch_size))[None,]
            if center[0] is not None:
                patch_slice = generate_slice_by_center and patch size(center[0], patch size=patch_patch_size)
                assert patch_slice[-1].start >= 0, f"patch_slice is None, {center[0]}, {patch_patch_size}"
                d[self.image_key][0] = adjust_contrast(patch_image=d[self.image_key][0],
                                                       patch_patch_slice=patch_slice,
                                                       mask_blur=self.mask_blur,
                                                       contrast_reduction_factor=self.contrast_reduction_factor)
        if self.do_artery_deformation:
            patch_patch_size = (int(self.mvf_mask_radius / spacing[0]), int(self.mvf_mask_radius / spacing[1]),
                                int(self.mvf_mask_radius / spacing[2]))
            # check the patch_patch_size is suitable
            if check_shape(a=scale_tuple(patch_patch_size, 3), b=shape):
                center = np.array(find_random_one_numpy(centerline[0],
                                                        patch_size=scale_tuple(patch_patch_size, 3)))[None,]
                # times 3 here to reduce deformation effect in the edge
                if center[0] is not None:
                    mvf = generate_mvf_vector(center=center, artery=artery[0], centerline=centerline[0] > 0,
                                              scale_factor=self.mvf_scale_factor, radius=self.mvf_mask_radius,
                                              direction_image=direction_image, spacing=spacing, shape=shape)
                    if mvf is None:
                        RuntimeWarning("MVF is None, probably because the center is too close to the edge or isolated")
                        return d
                    if not isinstance(self.mode[0], int):  # assert all modes are the same, either int or str
                        mvf = coords_numpy2torch(mvf, shape, change_coords=True)
                    for key, border_mode, border_cval, order in \
                            self.key_iterator(d, self.border_mode, self.border_cval, self.mode):
                        # use torch's grid-sampling to interpolate the data
                        d[key] = interpolator(data=d[key], coords=mvf, border_mode=border_mode, border_cval=border_cval,
                                              mode=order, )
        return d

    def randomize(self, data: Any = None):
        if self.R.uniform() < self.p_anatomy:
            self.do_artery_deformation = True
            self.mvf_mask_radius = self.R.uniform(self.deform_patch_patch_size_range[0],
                                                  self.deform_patch_patch_size_range[1])
            self.mvf_scale_factor = self.R.uniform(self.mvf_scale_factor_range[0],
                                                   self.mvf_scale_factor_range[1])
        else:
            self.do_artery_deformation = False

        if self.R.uniform() < self.p_contrast:
            self.do_local_contrast_change = True
            self.contrast_reduction_factor = self.R.uniform(self.contrast_reduction_factor_range[0],
                                                            self.contrast_reduction_factor_range[1])
            self.patch_patch_size_x_1 = self.R.randint(self.contrast_patch_patch_size_range[0][0],
                                                       self.contrast_patch_patch_size_range[0][1])
            self.patch_patch_size_y_1 = self.R.randint(self.contrast_patch_patch_size_range[1][0],
                                                       self.contrast_patch_patch_size_range[1][1])
            self.patch_patch_size_z_1 = self.R.randint(self.contrast_patch_patch_size_range[2][0],
                                                       self.contrast_patch_patch_size_range[2][1])
            self.mask_blur = self.R.uniform(self.mask_blur_range[0], self.mask_blur_range[1])
        else:
            self.do_local_contrast_change = False
```

```python
SparrowLink/transform/cardiac_transformation.py
import numpy as np
import torch
from monai.utils import TransformBackends
from typing import Any, Callable, Dict, Hashable, List, Mapping, Optional, Sequence, Union
from monai.config import IndexSelection, KeysCollection, SequenceStr, NdarrayOrTensor, DtypeLike, KeysCollection
from monai.utils.type_conversion import convert_data_type, convert_to_dst_type, convert_to_tensor, get_equivalent_dtype
from monai.data.meta_tensor import MetaTensor
from monai.metrics.utils import get_surface_distance, get_mask_edges


from monai.transforms import (
    MapTransform,
)
from openpyxl import Workbook, load_workbook
import os
from scipy import ndimage
import numpy as np


class UseHeartsegDeleteInformationd(MapTransform):

    backend = [TransformBackends.TORCH]

    def __init__(
        self,
        keys: KeysCollection,
        heart_key: str = 'heart',
        vessel_key: str = 'vessel',
        heart_dilation_time: int = 1,
        vessel_dilation_time: int = 1,
        dilation_struct: int = 1,
        allow_missing_keys: bool = False,
    ) -> None:
        """
        test_key: show whether your mask is reserved totally.
        """
        self.heart_dilation_time = heart_dilation_time
        self.vessel_dilation_time = vessel_dilation_time
        self.vessel_key = vessel_key
        self.heart_key = heart_key
        self.dilation_struct = dilation_struct

        super().__init__(keys, allow_missing_keys=allow_missing_keys)

    def __call__(self, data: Mapping[Hashable, torch.Tensor]) -> Dict[Hashable, torch.Tensor]:
        d = dict(data)
        heart_region = convert_to_tensor(data=d[self.heart_key])
        vessel_region = convert_to_tensor(data=d[self.vessel_key])
        heart_region = heart_region.clip(min=0, max=1) > 0
        vessel_region = vessel_region.clip(min=0, max=1) > 0
        if self.heart_dilation_time > 1:
            heart_region = self.dilation(x=heart_region, dilation_time=self.heart_dilation_time)
        if self.vessel_dilation_time > 1:
            vessel_region = self.dilation(x=vessel_region, dilation_time=self.vessel_dilation_time)
        region = (heart_region + vessel_region) > 0
        d[self.heart_key] = MetaTensor(region, meta=d[self.heart_key].meta)
        for key in self.key_iterator(d):
            if key not in [self.heart_key, self.vessel_key]:
                d[key] = d[key] * region
        return d

    def dilation(self, x: torch.Tensor, dilation_time: int = 1):
        shape = x.shape
        if len(shape) > 3:
            x = x.squeeze()
        x = np.array(x.squeeze())
        struct1 = ndimage.generate_binary_structure(3, self.dilation_struct)
        x = ndimage.binary_dilation(x, structure=struct1, iterations=dilation_time).astype(x.dtype)
        if len(shape) > 3:
            return torch.tensor(x).unsqueeze(dim=0)
        else:
            return torch.tensor(x)
```

```python
SparrowLink/transform/fast_crop.py
import numpy as np

def fast_index(img, point):
    """point: n * 3"""
    H, W, D = img.shape
    point = point[:, 2] + D * point[:, 1] + W * D * point[:, 0]
    batch_crop = img.reshape(-1)[point]
    return batch_crop


def fast_crop(img, point):
    """
    point: n * h * w * d * 3
    return: n, h, w, d
    """
    n, h, w, d, _ = point.shape
    point = point.reshape(n * h * w * d, 3)
    batch_crop = fast_index(img, point)
    batch_crop = batch_crop.reshape(n, h, w, d)

    return batch_crop


def generate_cube(point, h, w, d):
    """
    point: n * 3
    return: n * h * w * d * 3
    """

    """"""""
    n, _ = point.shape
    x = np.arange(-h, h+1)
    y = np.arange(-w, w+1)
    z = np.arange(-d, d+1)

    [Y, X, Z] = np.meshgrid(y, x, z)
    crop_region_index = np.zeros((x.shape[0] * y.shape[0] * z.shape[0], 3))
    crop_region_index[:, 0],  crop_region_index[:, 1], crop_region_index[:, 2] = \
        X.reshape(-1), Y.reshape(-1), Z.reshape(-1)
    region_index = point[:, None, :] + crop_region_index[None, :, :]
    region_index = region_index.reshape(n, x.shape[0], y.shape[0], z.shape[0], 3)
    return region_index.astype(np.uint16)


def fancy_indexing(imgs, centers, pw, ph):
    n = imgs.shape[0]
    img_i, RGB, x, y = np.ogrid[:n, :3, :pw, :ph]
    corners = centers - [pw//2, ph//2]
    x_i = x + corners[:,0,None,None,None]
    y_i = y + corners[:,1,None,None,None]
    return imgs[img_i, RGB, x_i, y_i]


if __name__ == '__main__':
    H = 8
    W = 8
    D = 8
    img = np.arange(H * W * D).reshape(H, W, D)
    point = np.array([[3, 3, 3],
                      [2, 2, 2],
                      [1, 1, 1], ])

    cube_point = generate_cube(point, h=1, w=1, d=1)

    crop = fast_crop(img, point=cube_point)

    print(img)
    print(crop)
```

```python
SparrowLink/transform/IntensityTransformD.py
import numpy as np
import torch
from monai.utils import TransformBackends
from typing import Any, Callable, Dict, Hashable, List, Mapping, Optional, Sequence, Union
from monai.config import IndexSelection, KeysCollection, SequenceStr, NdarrayOrTensor, DtypeLike, KeysCollection
from monai.data.meta_obj import get_track_meta
from monai.utils.type_conversion import convert_data_type, convert_to_dst_type, convert_to_tensor, get_equivalent_dtype
from monai.transforms.utils_pytorch_numpy_unification import clip
from monai.data.meta_tensor import MetaTensor
from monai.metrics.utils import get_surface_distance, get_mask_edges
from monai.transforms import MapTransform, RandomizableTransform, Randomizable

from batchgenerators.augmentations.noise_augmentations import augment_gaussian_blur, augment_gaussian_noise, \
    augment_rician_noise
from batchgenerators.augmentations.color_augmentations import augment_contrast, augment_brightness_additive, \
    augment_brightness_multiplicative, augment_gamma, augment_illumination, augment_PCA_shift
from batchgenerators.augmentations.resample_augmentations import augment_linear_downsampling_scipy
from typing import Any, Callable, Dict, Hashable, List, Mapping, Optional, Sequence, Union, Tuple


class CTNormalizeD(MapTransform):
    """
    Dictionary-based wrapper of :py:class:`monai.transforms.NormalizeIntensity`.
    This transform can normalize only non-zero values or entire image, and can also calculate
    mean and std on each channel separately.
    """

    backend = [TransformBackends.TORCH, TransformBackends.NUMPY]

    def __init__(
        self,
        keys: KeysCollection,
        mean_intensity: float = None,
        std_intensity: float = None,
        lower_bound: float = None,
        upper_bound: float = None,
        allow_missing_keys: bool = False,
    ) -> None:
        super().__init__(keys, allow_missing_keys)
        self.mean_intensity = mean_intensity
        self.std_intensity = std_intensity
        self.lower_bound = lower_bound
        self.upper_bound = upper_bound

    @staticmethod
    def normalize_intensity(
            img: NdarrayOrTensor,
            mean_intensity: float = None,
            std_intensity: float = None,
            lower_bound: float = None,
            upper_bound: float = None,
            dtype: DtypeLike = np.float32,
    ):
        img = convert_to_tensor(img, track_meta=get_track_meta())
        dtype = dtype or img.dtype
        img = convert_to_tensor(img, track_meta=get_track_meta())
        img = clip(img, a_min=lower_bound, a_max=upper_bound)
        img = (img - mean_intensity) / max(std_intensity, 1e-8)
        # img = MetaTensor(img, meta=img.meta)
        ret: NdarrayOrTensor = convert_data_type(img, dtype=dtype)[0]
        return ret

    def __call__(self, data: Mapping[Hashable, NdarrayOrTensor]) -> Dict[Hashable, NdarrayOrTensor]:
        d = dict(data)
        for key in self.keys:
            d[key] = self.normalize_intensity(img=d[key],
                                              mean_intensity=self.mean_intensity,
                                              std_intensity=self.std_intensity,
                                              lower_bound=self.lower_bound,
                                              upper_bound=self.upper_bound,
                                              )
        return d


class BrightnessMultiplicativeD(RandomizableTransform, MapTransform):
    """
    Adds additive Gaussian Noise
    :param keys: selecting the keys to be transformed
    :param prob: probability of the noise being added, per sample
    :param prob_per_channel: probability of the noise being added, per channel
    CAREFUL: This transform will modify the value range of your data!
    """

    backend = [TransformBackends.TORCH, TransformBackends.NUMPY]

    def __init__(
            self,
            keys: KeysCollection,
            prob: float = 0.1,
            prob_per_channel: float = 1.0,
            per_channel: bool = True,
            multiplier_range: Tuple[float, float] = (0.5, 2),
            allow_missing_keys: bool = False,
    ) -> None:

```

```
        MapTransform.__init__(self, keys, allow_missing_keys)
        RandomizableTransform.__init__(self, prob=prob)
        self.keys = keys
        self.prob = prob
        self.multiplier_range = multiplier_range
        self.per_channel = per_channel
        self.prob_per_channel = prob_per_channel

    def set_random_state(
        self, seed: Optional[int] = None, state: Optional[np.random.RandomState] = None
    ) -> "BrightnessMultiplicativeD":
        super().set_random_state(seed=seed, state=state)
        return self

    def __call__(self, data: Mapping[Hashable, NdarrayOrTensor]) -> Dict[Hashable, NdarrayOrTensor]:
        d = dict(data)
        self.randomize(None)
        for key in self.keys:
            if self._do_transform:
                meta = None
                if get_track_meta():
                    meta = d[key].meta
                dtype = d[key].dtype
                shape = d[key].shape
                assert len(shape) == 4, "img should be 4D array, (c, w, h, d)"

                img = np.array(d[key])
                img = augment_brightness_multiplicative(
                    data_sample=img,
                    multiplier_range=self.multiplier_range,
                    per_channel=self.per_channel,
                )
                if meta:
                    img = MetaTensor(img, meta=meta)
                d[key] = convert_data_type(img, dtype=dtype)[0]

        return d


class ContrastAugmentationD(RandomizableTransform, MapTransform):
    """
    Adds additive Gaussian Noise
    :param keys: selecting the keys to be transformed
    :param prob: probability of the noise being added, per sample
    :param prob_per_channel: probability of the noise being added, per channel
    CAREFUL: This transform will modify the value range of your data!
    """
    backend = [TransformBackends.TORCH, TransformBackends.NUMPY]

    def __init__(
            self,
            keys: KeysCollection,
            prob: float = 0.1,
            prob_per_channel: float = 1.0,
            per_channel: bool = True,
            contrast_range: Tuple[float, float] = (0.75, 1.25),
            preserve_range: bool = True,
            allow_missing_keys: bool = False,
    ) -> None:

        MapTransform.__init__(self, keys, allow_missing_keys)
        RandomizableTransform.__init__(self, prob=prob)
        self.keys = keys
        self.prob = prob
        self.contrast_range = contrast_range
        self.preserve_range = preserve_range
        self.per_channel = per_channel
        self.prob_per_channel = prob_per_channel

    def set_random_state(
        self, seed: Optional[int] = None, state: Optional[np.random.RandomState] = None
    ) -> "ContrastAugmentationD":
        super().set_random_state(seed=seed, state=state)
        return self

    def __call__(self, data: Mapping[Hashable, NdarrayOrTensor]) -> Dict[Hashable, NdarrayOrTensor]:
        d = dict(data)
        self.randomize(None)
        for key in self.keys:
            if self._do_transform:
                meta = None
                if get_track_meta():
                    meta = d[key].meta
                dtype = d[key].dtype
                shape = d[key].shape
                assert len(shape) == 4, "img should be 4D array, (c, w, h, d)"

                img = np.array(d[key])
                img = augment_contrast(
                    data_sample=img,
                    contrast_range=self.contrast_range,
                    preserve_range=self.preserve_range,
                    per_channel=self.per_channel,
                )
                if meta:
                    img = MetaTensor(img, meta=meta)
```

```
                d[key] = convert_data_type(img, dtype=dtype)[0]

        return d


class SimulateLowResolutionD(RandomizableTransform, MapTransform):
    """
    Adds additive Gaussian Noise
    :param keys: selecting the keys to be transformed
    :param prob: probability of the noise being added, per sample
    :param prob_per_channel: probability of the noise being added, per channel
    CAREFUL: This transform will modify the value range of your data!
    """
    backend = [TransformBackends.TORCH, TransformBackends.NUMPY]

    def __init__(
            self,
            keys: KeysCollection,
            prob: float = 0.1,
            prob_per_channel: float = 1.0,
            per_channel: bool = True,
            zoom_range: Tuple[float, float] = (0.5, 1),
            order_downsample: int = 0,
            order_upsample: int = 3,
            ignore_axes: bool = None,
            allow_missing_keys: bool = False,
    ) -> None:

        MapTransform.__init__(self, keys, allow_missing_keys)
        RandomizableTransform.__init__(self, prob=prob)
        self.keys = keys
        self.prob = prob
        self.zoom_range = zoom_range
        self.order_downsample = order_downsample
        self.order_upsample = order_upsample
        self.ignore_axes = ignore_axes
        self.per_channel = per_channel
        self.prob_per_channel = prob_per_channel

    def set_random_state(
        self, seed: Optional[int] = None, state: Optional[np.random.RandomState] = None
    ) -> "SimulateLowResolutionD":
        super().set_random_state(seed=seed, state=state)
        return self

    def __call__(self, data: Mapping[Hashable, NdarrayOrTensor]) -> Dict[Hashable, NdarrayOrTensor]:
        d = dict(data)
        self.randomize(None)
        for key in self.keys:
            if self._do_transform:
                meta = None
                if get_track_meta():
                    meta = d[key].meta
                dtype = d[key].dtype
                shape = d[key].shape
                assert len(shape) == 4, "img should be 4D array, (c, w, h, d)"

                img = np.array(d[key])
                img = augment_linear_downsampling_scipy(
                    data_sample=img,
                    zoom_range=self.zoom_range,
                    order_downsample=self.order_downsample,
                    order_upsample=self.order_upsample,
                    ignore_axes=self.ignore_axes,
                    per_channel=self.per_channel,
                )

                if meta:
                    img = MetaTensor(img, meta=meta)
                d[key] = convert_data_type(img, dtype=dtype)[0]
        return d


class GammaD(RandomizableTransform, MapTransform):
    """
    Adds additive Gaussian Noise
    :param keys: selecting the keys to be transformed
    :param prob: probability of the noise being added, per sample
    :param prob_per_channel: probability of the noise being added, per channel
    CAREFUL: This transform will modify the value range of your data!
    """
    backend = [TransformBackends.TORCH, TransformBackends.NUMPY]

    def __init__(
            self,
            keys: KeysCollection,
            prob: float = 0.1,
            prob_per_channel: float = 1.0,
            per_channel: bool = True,
            gamma_range: Tuple[float, float] = (0.5, 2),
            invert_image: bool = False,
            retain_stats: bool = False,
            allow_missing_keys: bool = False,
    ) -> None:

        MapTransform.__init__(self, keys, allow_missing_keys)
        RandomizableTransform.  init  (self, prob=prob)
```

```
        self.keys = keys
        self.prob = prob
        self.gamma_range = gamma_range
        self.invert_image = invert_image
        self.retain_stats = retain_stats
        self.per_channel = per_channel
        self.prob_per_channel = prob_per_channel

    def set_random_state(
        self, seed: Optional[int] = None, state: Optional[np.random.RandomState] = None
    ) -> "GammaD":
        super().set_random_state(seed=seed, state=state)
        return self

    def __call__(self, data: Mapping[Hashable, NdarrayOrTensor]) -> Dict[Hashable, NdarrayOrTensor]:
        d = dict(data)
        self.randomize(None)
        for key in self.keys:
            if self._do_transform:
                meta = None
                if get_track_meta():
                    meta = d[key].meta
                dtype = d[key].dtype
                shape = d[key].shape
                assert len(shape) == 4, "img should be 4D array, (c, w, h, d)"

                img = np.array(d[key])
                img = augment_gamma(
                    data_sample=img,
                    gamma_range=self.gamma_range,
                    invert_image=self.invert_image,
                    retain_stats=self.retain_stats,
                    per_channel=self.per_channel,
                )
                if meta:
                    img = MetaTensor(img, meta=meta)
                d[key] = convert_data_type(img, dtype=dtype)[0]

        return d
```

```python
SparrowLink/transform/NoiseTransformD.py
import numpy as np
import torch
from monai.utils import TransformBackends
from typing import Any, Callable, Dict, Hashable, List, Mapping, Optional, Sequence, Union
from monai.config import IndexSelection, KeysCollection, SequenceStr, NdarrayOrTensor, DtypeLike, KeysCollection
from monai.data.meta_obj import get_track_meta
from monai.utils.type_conversion import convert_data_type, convert_to_dst_type, convert_to_tensor, get_equivalent_dtype
from monai.transforms.utils_pytorch_numpy_unification import clip
from monai.data.meta_tensor import MetaTensor
from monai.metrics.utils import get_surface_distance, get_mask_edges
from monai.transforms import MapTransform, RandomizableTransform, Randomizable

from batchgenerators.augmentations.noise_augmentations import augment_gaussian_blur, augment_gaussian_noise, \
    augment_rician_noise
from typing import Any, Callable, Dict, Hashable, List, Mapping, Optional, Sequence, Union, Tuple


class GaussianNoiseD(RandomizableTransform, MapTransform):
    """
    Adds additive Gaussian Noise
    :param keys: selecting the keys to be transformed
    :param prob: probability of the noise being added, per sample
    :param prob_per_channel: probability of the noise being added, per channel
    CAREFUL: This transform will modify the value range of your data!
    """
    backend = [TransformBackends.TORCH, TransformBackends.NUMPY]

    def __init__(
            self,
            keys: KeysCollection,
            prob: float = 0.1,
            prob_per_channel: float = 1.0,
            per_channel: bool = True,
            noise_variance: Tuple[float, float] = (0.0, 0.1),
            allow_missing_keys: bool = False,
    ) -> None:

        MapTransform.__init__(self, keys, allow_missing_keys)
        RandomizableTransform.__init__(self, prob=prob)
        self.keys = keys
        self.prob = prob
        self.noise_variance = noise_variance
        self.per_channel = per_channel
        self.prob_per_channel = prob_per_channel

    def set_random_state(
        self, seed: Optional[int] = None, state: Optional[np.random.RandomState] = None
    ) -> "GaussianNoiseD":
        super().set_random_state(seed=seed, state=state)
        return self

    def __call__(self, data: Mapping[Hashable, NdarrayOrTensor]) -> Dict[Hashable, NdarrayOrTensor]:
        d = dict(data)
        self.randomize(None)
        for key in self.keys:
            if self._do_transform:
                meta = None
                if get_track_meta():
                    meta = d[key].meta
                dtype = d[key].dtype
                shape = d[key].shape
                assert len(shape) == 4, "img should be 4D array, (c, w, h, d)"

                img = np.array(d[key])
                img = augment_gaussian_noise(data_sample=img,
                                             noise_variance=self.noise_variance,
                                             p_per_channel=self.prob_per_channel,
                                             per_channel=self.per_channel,
                                             )
                if meta:
                    img = MetaTensor(img, meta=meta)
                d[key] = convert_data_type(img, dtype=dtype)[0]

        return d


class GaussianBlurD(RandomizableTransform, MapTransform):
    """
    Adds additive Gaussian Noise
    :param keys: selecting the keys to be transformed
    :param prob: probability of the noise being added
    CAREFUL: This transform will modify the value range of your data!

    """
    backend = [TransformBackends.TORCH, TransformBackends.NUMPY]

    def __init__(
            self,
            keys: KeysCollection,
            blur_sigma: Tuple[float, float] = (0.0, 0.1),
            prob: float = 0.2,
            prob_per_channel: float = 1.0,
            per_channel: bool = True,
            allow_missing_keys: bool = False,
    ) -> None:
```

```
        MapTransform.__init__(self, keys, allow_missing_keys)
        RandomizableTransform.__init__(self, prob=prob)
        self.prob = prob
        self.blur_sigma = blur_sigma
        self.per_channel = per_channel
        self.prob_per_channel = prob_per_channel

    def set_random_state(
        self, seed: Optional[int] = None, state: Optional[np.random.RandomState] = None
    ) -> "GaussianBlurD":
        super().set_random_state(seed=seed, state=state)
        return self

    def __call__(self, data: Mapping[Hashable, NdarrayOrTensor]) -> Dict[Hashable, NdarrayOrTensor]:
        d = dict(data)
        self.randomize(None)
        for key in self.keys:
            if self._do_transform:
                meta = None
                if get_track_meta():
                    meta = d[key].meta
                dtype = d[key].dtype
                shape = d[key].shape
                assert len(shape) == 4, "img should be 4D array, (c, w, h, d)"

                img = np.array(d[key])

                img = augment_gaussian_blur(data_sample=img,
                                            sigma_range=self.blur_sigma,
                                            per_channel=self.per_channel,
                                            p_per_channel=self.prob_per_channel,
                                            )
                if meta:
                    img = MetaTensor(img, meta=meta)
                d[key] = convert_data_type(img, dtype=dtype)[0]
        return d
```

```python
SparrowLink/transform/SpatialTransformationD.py
import numpy as np
import torch
from monai.utils import TransformBackends
from typing import Any, Callable, Dict, Hashable, List, Mapping, Optional, Sequence, Union
from monai.config import IndexSelection, KeysCollection, SequenceStr, NdarrayOrTensor, DtypeLike, KeysCollection
from monai.data.meta_obj import get_track_meta
from monai.utils.type_conversion import convert_data_type, convert_to_dst_type, convert_to_tensor, get_equivalent_dtype
from monai.transforms.utils_pytorch_numpy_unification import clip
from monai.data.meta_tensor import MetaTensor
from monai.metrics.utils import get_surface_distance, get_mask_edges
from monai.transforms import MapTransform, RandomizableTransform, Randomizable, InvertibleTransform

from batchgenerators.augmentations.noise_augmentations import augment_gaussian_blur, augment_gaussian_noise, \
    augment_rician_noise
from batchgenerators.augmentations.color_augmentations import augment_contrast, augment_brightness_additive, \
    augment_brightness_multiplicative, augment_gamma, augment_illumination, augment_PCA_shift
from batchgenerators.augmentations.resample_augmentations import augment_linear_downsampling_scipy
from typing import Any, Callable, Dict, Hashable, List, Mapping, Optional, Sequence, Union, Tuple


class PermuteD(MapTransform, InvertibleTransform):
    """
    Dictionary-based wrapper of :py:class:`monai.transforms.Orientation`.

    This transform assumes the channel-first input format.
    In the case of using this transform for normalizing the orientations of images,
    it should be used before any anisotropic spatial transforms.
    """

    backend = [TransformBackends.NUMPY, TransformBackends.TORCH]

    def __init__(
        self,
        keys: KeysCollection,
        axcodes: Optional[str] = None,
        as_closest_canonical: bool = False,
        labels: Optional[Sequence[Tuple[str, str]]] = (("L", "R"), ("P", "A"), ("I", "S")),
        meta_keys: Optional[KeysCollection] = None,
        meta_key_postfix: str = "meta_dict",
        allow_missing_keys: bool = False,
    ) -> None:
        """
        Args:
            axcodes: N elements sequence for spatial ND input's orientation.
                e.g. axcodes='RAS' represents 3D orientation:
                (Left, Right), (Posterior, Anterior), (Inferior, Superior).
                default orientation labels options are: 'L' and 'R' for the first dimension,
                'P' and 'A' for the second, 'I' and 'S' for the third.
            as_closest_canonical: if True, load the image as closest to canonical axis format.
            labels: optional, None or sequence of (2,) sequences
                (2,) sequences are labels for (beginning, end) of output axis.
                Defaults to ``(('L', 'R'), ('P', 'A'), ('I', 'S'))``.
            allow_missing_keys: don't raise exception if key is missing.

        See Also:
            `nibabel.orientations.ornt2axcodes`.

        """
        super().__init__(keys, allow_missing_keys)
        self.ornt_transform = Orientation(axcodes=axcodes, as_closest_canonical=as_closest_canonical, labels=labels)

    def __call__(self, data: Mapping[Hashable, torch.Tensor]) -> Dict[Hashable, torch.Tensor]:
        d: Dict = dict(data)
        for key in self.key_iterator(d):
            d[key] = self.ornt_transform(d[key])
        return d

    def inverse(self, data: Mapping[Hashable, torch.Tensor]) -> Dict[Hashable, torch.Tensor]:
        d = dict(data)
        for key in self.key_iterator(d):
            d[key] = self.ornt_transform.inverse(d[key])
        return d
```

```python
SparrowLink/transform/utils.py
import numpy as np
import torch
from monai.utils import TransformBackends
from typing import Any, Callable, Dict, Hashable, List, Mapping, Optional, Sequence, Union
from monai.config import IndexSelection, KeysCollection, SequenceStr, NdarrayOrTensor, DtypeLike, KeysCollection
from monai.data.meta_obj import get_track_meta
from monai.utils.type_conversion import convert_data_type, convert_to_dst_type, convert_to_tensor, get_equivalent_dtype
from monai.transforms.utils_pytorch_numpy_unification import clip
from monai.data.meta_tensor import MetaTensor
from monai.metrics.utils import get_surface_distance, get_mask_edges


from monai.transforms import (
    EnsureChannelFirstd,
    Compose,
    CropForegroundd,
    LoadImaged,
    Orientationd,
    RandCropByPosNegLabeld,
    SaveImaged,
    ScaleIntensityRanged,
    Spacingd,
    RandAffined,
    MapTransform,
    RandFlipd,
    RandRotated,
)
from openpyxl import Workbook, load_workbook
import os
from scipy import ndimage
import numpy as np

from transform.IntensityTransformD import CTNormalizeD, BrightnessMultiplicativeD, ContrastAugmentationD, \
    SimulateLowResolutionD, GammaD
from transform.NoiseTransformD import GaussianNoiseD, GaussianBlurD


class UseHeartsegDeleteInformationd(MapTransform):

    backend = [TransformBackends.TORCH]

    def __init__(
        self,
        keys: KeysCollection,
        dilation_key: str,
        dilation_time: int = 1,
        dilation_struct: int = 1,
        test_key: str = None,
        caculate_surface_disdance: bool = False,
        vessel_key: str = None,
        chamber_key: str = None,
        save_path: str = None,
        allow_missing_keys: bool = False,
        **pad_kwargs,
    ) -> None:
        """
        test_key: show whether your mask is reserved totally.
        """
        self.dilation_key = dilation_key
        self.dilation_time = dilation_time
        self.dilation_struct = dilation_struct
        self.test_key = test_key
        self.caculate_surface_disdance = caculate_surface_disdance
        self.vessel_key = vessel_key
        self.chamber_key = chamber_key
        self.save_path = save_path
        self.error_list = []
        self.wb = None

        if self.caculate_surface_disdance:
            row = ('image_name', 'surface disdance', 'error')
            self.wb = Workbook()
            self.excel_writer(row=row, file_name=self.save_path)

        super().__init__(keys, allow_missing_keys=allow_missing_keys)

    def __call__(self, data: Mapping[Hashable, torch.Tensor]) -> Dict[Hashable, torch.Tensor]:
        d = dict(data)
        region = convert_to_tensor(data=d[self.dilation_key])
        if self.dilation_time > 1:
            region = region.clip(min=0, max=1) > 0
            region = self.dilation(x=region)  # bool

            distance = -1
            error = 0
            for key in self.key_iterator(d):
                if self.caculate_surface_disdance:
                    vessel_edge, chamber_edge = get_mask_edges(d[self.vessel_key], d[self.chamber_key] > 0)
                    distance = get_surface_distance(vessel_edge, chamber_edge)
                    distance = distance.max()

                if key == self.dilation_key:
                    d[key] = MetaTensor(region, meta=d[key].meta)
                else:
                    # record the img information if label has some voxels out of dilation region
                    if key == self.test_key and (d[key] * (~region)).sum() > 0:
                        self.error_list.append(d[key].meta['filename_or_obj'])
                        error = 1
                        print("---------------error crop----------------")
                        print(d[key].meta['filename_or_obj'])

                    d[key] = d[key] * region

            # record image dilation information in Excel
            if self.wb:
                row = (os.path.split(d[self.test_key].meta['filename_or_obj'])[-1], distance, error)
                self.excel_writer(row=row, file_name=self.save_path)


        else:
```

```
            # if you choose to set dilation before training process
            region = region.clip(min=0, max=1) > 0
            for key in self.key_iterator(d):
                if key != self.dilation_key:
                    d[key] = d[key] * region
        return d

    def excel_writer(self, row, file_name):
        ws = self.wb.active
        ws.append(row)
        self.wb.save(file_name)

    def dilation(self, x: torch.Tensor):
        shape = x.shape
        if len(shape) > 3:
            x = x.squeeze()
        x = np.array(x.squeeze())
        struct1 = ndimage.generate_binary_structure(3, self.dilation_struct)
        x = ndimage.binary_dilation(x, structure=struct1, iterations=self.dilation_time).astype(x.dtype)
        if len(shape) > 3:
            return torch.tensor(x).unsqueeze(dim=0)
        else:
            return torch.tensor(x)


class Dilationd(MapTransform):

    backend = [TransformBackends.TORCH]

    def __init__(
        self,
        keys: KeysCollection,
        dilation_time: int = 1,
        dilation_struct: int = 1,
        allow_missing_keys: bool = False,
        **pad_kwargs,
    ) -> None:
        """
        test_key: show whether your mask is reserved totally.
        """
        self.dilation_time = dilation_time
        self.dilation_struct = dilation_struct
        super().__init__(keys, allow_missing_keys=allow_missing_keys)

    def __call__(self, data: Mapping[Hashable, torch.Tensor]) -> Dict[Hashable, torch.Tensor]:
        d = dict(data)
        for key in self.keys:
            region = convert_to_tensor(data=d[key])
            region = self.dilation(region)
            d[key] = MetaTensor(region, meta=d[key].meta)
        return d

    def dilation(self, x: torch.Tensor):
        shape = x.shape
        if len(shape) > 3:
            x = x.squeeze()
        x = np.array(x.squeeze())
        struct1 = ndimage.generate_binary_structure(3, self.dilation_struct)
        x = ndimage.binary_dilation(x, structure=struct1, iterations=self.dilation_time).astype(x.dtype)
        if len(shape) > 3:
            return torch.tensor(x).unsqueeze(dim=0)
        else:
            return torch.tensor(x)


def get_transform(transform_dic, mode='train'):
    """dic is transform block of config.yaml"""
    """
    only patch_size and normalization is controlled by config.yaml
    """
    if transform_dic.get('use_config') is True:
        return get_config_transform(transform_dic, mode=mode)
    else:
        return get_default_transform(transform_dic, mode=mode)


def get_default_transform(transform_dic, mode='train'):
    """dic is transform block of config.yaml"""
    """
    only patch_size and normalization is controlled by config.yaml
    """
    if mode == 'train':
        train_transforms = Compose(
            [
                LoadImaged(keys=["image", "label"], image_only=False),
                EnsureChannelFirstd(keys=["image", "label"]),
                ScaleIntensityRanged(
                    keys=["image"], a_min=-57, a_max=400,
                    b_min=0.0, b_max=1.0, clip=True,
                ),
                CTNormalizeD(keys=["image"],
                             mean_intensity=transform_dic["normalize"]["mean"],
                             std_intensity=transform_dic["normalize"]["std"],
                             lower_bound=transform_dic["normalize"]["min"],
                             upper_bound=transform_dic["normalize"]["max"], ),
                CropForegroundd(keys=["image", "label"], source_key="image"),
                Orientationd(keys=["image", "label"], axcodes="RAI"),
                # 记得改
                Spacingd(keys=["image", "label"], pixdim=transform_dic["spacing"]
                         , mode=("bilinear", "nearest")),
                RandRotated(
                    keys=["image", "label"],
                    range_x=np.pi / 6,
                    range_y=np.pi / 6,
                    range_z=np.pi / 6,
                    prob=1,
                    padding_mode="zeros",
                    mode=("bilinear", "nearest")
                ),
                RandCropByPosNegLabeld(
                    keys=["image", "label"],
                    label_key="label",
```

```
                    spatial_size=tuple(transform_dic["patch_size"]),
                    pos=3,
                    neg=1,
                    num_samples=4,
                    image_key="image",
                    image_threshold=0,
                ),
                GaussianNoiseD(keys=["image"], noise_variance=(0, 0.1), prob=0.1,),
                GaussianBlurD(keys=["image"], blur_sigma=(0.5, 1.15), prob=0.1,),
                BrightnessMultiplicativeD(keys=["image"], prob=0.15, multiplier_range=(0.7, 1.25)),
                ContrastAugmentationD(keys=["image"], prob=0.15),
                SimulateLowResolutionD(keys=["image"], zoom_range=(0.5, 1), prob=0.20,
                                       order_upsample=3, order_downsample=0, ignore_axes=None),
                GammaD(keys=["image"], gamma_range=(0.7, 1.5), invert_image=True,
                       per_channel=True, retain_stats=True, prob=0.1),
                GammaD(keys=["image"], gamma_range=(0.7, 1.5), invert_image=True,
                       per_channel=True, retain_stats=True, prob=0.3),
                RandFlipd(keys=["image", "label"], prob=0.5, spatial_axis=0),
                RandFlipd(keys=["image", "label"], prob=0.5, spatial_axis=1),
                RandFlipd(keys=["image", "label"], prob=0.5, spatial_axis=2),
            ]
        )
        val_transforms = Compose(
            [
                LoadImaged(keys=["image", "label"], image_only=False),
                EnsureChannelFirstd(keys=["image", "label"]),
                # ScaleIntensityRanged(
                #     keys=["image"], a_min=-57, a_max=400,
                #     b_min=0.0, b_max=1.0, clip=True,
                # ),
                CTNormalizeD(keys=["image"],
                             mean_intensity=transform_dic["normalize"]["mean"],
                             std_intensity=transform_dic["normalize"]["std"],
                             lower_bound=transform_dic["normalize"]["min"],
                             upper_bound=transform_dic["normalize"]["max"], ),
                CropForegroundd(keys=["image", "label"], source_key="image"),  # default: value>0
                Orientationd(keys=["image", "label"], axcodes="RAI"),
                Spacingd(keys=["image", "label"], pixdim=transform_dic["spacing"]
                         , mode=("bilinear", "nearest")),  # Now only Sequential_str, How to add spline
            ]
        )
        save_transform = Compose(
            [
                LoadImaged(keys=["image", "label"], image_only=False),
                EnsureChannelFirstd(keys=["image", "label"]),
                SaveImaged(keys=["image"], output_dir='./transform/test_transform', output_postfix='origin_image',
                           print_log=False),
                SaveImaged(keys=["label"], output_dir='./transform/test_transform', output_postfix='origin_label',
                           print_log=False),
                CTNormalizeD(keys=["image"],
                             mean_intensity=transform_dic["normalize"]["mean"],
                             std_intensity=transform_dic["normalize"]["std"],
                             lower_bound=transform_dic["normalize"]["min"],
                             upper_bound=transform_dic["normalize"]["max"], ),
                CropForegroundd(keys=["image", "label"], source_key="image"),
                Orientationd(keys=["image", "label"], axcodes="RAI"),
                Spacingd(keys=["image", "label"], pixdim=transform_dic["spacing"]
                         , mode=("bilinear", "nearest")),
                GaussianNoiseD(keys=["image"], noise_variance=(0, 0.1), prob=1.0,),
                GaussianBlurD(keys=["image"], blur_sigma=(0.5, 1.15), prob=1.0,),
                SaveImaged(keys=["image"], output_dir='./transform/test_transform', output_postfix='tran_image',
                           print_log=False),
                SaveImaged(keys=["label"], output_dir='./transform/test_transform', output_postfix='tran_label',
                           print_log=False),
            ]
        )
        return train transforms, val_transforms, save_transform
    elif mode == 'infer':
        infer_transforms = Compose(
            [
                LoadImaged(keys=["image"], image_only=False),
                EnsureChannelFirstd(keys=["image"]),
                # ScaleIntensityRanged(
                #     keys=["image"], a_min=-57, a_max=400,
                #     b_min=0.0, b_max=1.0, clip=True,
                # ),
                CTNormalizeD(keys=["image"],
                             mean_intensity=transform_dic["normalize"]["mean"],
                             std_intensity=transform_dic["normalize"]["std"],
                             lower_bound=transform_dic["normalize"]["min"],
                             upper_bound=transform_dic["normalize"]["max"], ),
                # CropForegroundd(keys=["image"], source_key="image"),  # default: value>0
                Orientationd(keys=["image"], axcodes="RAI"),
                Spacingd(keys=["image"], pixdim=transform_dic["spacing"], mode=("bilinear")),  # Now only Sequential_str, How to add spline
            ]
        )
        return infer_transforms
    else:
        raise RuntimeError(f"{mode} is not supported yet")


def get_config_transform(transform_dic, mode='train'):
    """dic is transform block of config.yaml"""
    """
    you can further control your random transformation
    """
    if transform_dic.get("image_resample") is not None:
        image_resample_mode = transform_dic["image_resample"]["mode"]
        image_resample_padding_mode = transform_dic["image_resample"]["padding_mode"]
    else:
        image_resample_mode = "bilinear"
        image_resample_padding_mode = "border"

    if mode == 'train':
        fixed_transforms = [
            LoadImaged(keys=["image", "label"]),
            EnsureChannelFirstd(keys=["image", "label"]),
            CTNormalizeD(keys=["image"],
                         mean_intensity=transform_dic["normalize"]["mean"],
                         std_intensity=transform_dic["normalize"]["std"],
                         lower_bound=transform_dic["normalize"]["min"],
                         upper_bound=transform_dic["normalize"]["max"], ),
```

```
            CropForegroundd(keys=["image", "label"], source_key="image"),
            Orientationd(keys=["image", "label"], axcodes="RAI"),
            # 记得改
            Spacingd(keys=["image", "label"], pixdim=transform_dic["spacing"]
                     , mode=(image_resample_mode, "nearest"), padding_mode=(image_resample_padding_mode, "border")),
        ]
        print("----------------------training_fixed_transform-------------------------")
        if transform_dic.get("fixed") is not None and len(transform_dic.get("fixed")) > 0:
            for d in transform_dic["fixed"]:
                key = d.get('name')
                value = d.get('parameter')
                assert key is not None, "name of fixed transform is not defined in config.yaml"
                assert value is not None, "parameter of fixed transform is not defined in config.yaml"
                print(f"name: {key}, parameter: {value}")
                trans_class = eval(key)
                trans_module = trans_class(**value)
                fixed_transforms.append(trans_module)

        random_transforms = []
        # assert transform_dic.get("random") is not None, "random transform is not defined in config.yaml"
        if transform_dic.get("random") is None or len(transform_dic.get("random")) == 0:
            print("warning: random transform is not defined in config.yaml")
        else:
            print("----------------------training_random_transform-------------------------")
            for d in transform_dic["random"]:
                key = d.get('name')
                value = d.get('parameter')
                assert key is not None, "name of random transform is not defined in config.yaml"
                assert value is not None, "parameter of random transform is not defined in config.yaml"
                print(f"name: {key}, parameter: {value}")
                trans_class = eval(key)
                trans_module = trans_class(**value)
                random_transforms.append(trans_module)

        train_transforms = Compose(fixed_transforms + random_transforms)
        val_transforms = Compose(fixed_transforms)
        save_transform = None
        return train_transforms, val_transforms, save_transform
    elif mode == 'infer':
        infer_transforms = [
            LoadImaged(keys=["image"]),
            EnsureChannelFirstd(keys=["image"]),
            CTNormalizeD(keys=["image"],
                         mean_intensity=transform_dic["normalize"]["mean"],
                         std_intensity=transform_dic["normalize"]["std"],
                         lower_bound=transform_dic["normalize"]["min"],
                         upper_bound=transform_dic["normalize"]["max"], ),
            CropForegroundd(keys=["image"], source_key="image"),  # default: value>0
            Orientationd(keys=["image"], axcodes="RAI"),
            Spacingd(keys=["image"], pixdim=transform_dic["spacing"], mode=image_resample_mode,
                     padding_mode=image_resample_padding_mode),
            ]
        if transform_dic.get("infer") is not None and len(transform_dic.get("infer")) > 0:
            for d in transform_dic["infer"]:
                key = d.get('name')
                value = d.get('parameter')
                assert key is not None, "name of infer transform is not defined in config.yaml"
                assert value is not None, "parameter of infer transform is not defined in config.yaml"
                trans_class = eval(key)
                trans_module = trans_class(**value)
                infer_transforms.append(trans_module)

        infer_transforms = Compose(infer_transforms)
        return infer_transforms
    else:
        raise RuntimeError(f"{mode} is not supported yet")


def print_config_transforms(transforms_list):
    import inspect
    for transforms in transforms_list:
        print(f"name: {inspect.getmodule(transforms).__name__}, parameter: {inspect.getmembers(transforms)}")


def get_multi_phase_transform_with_image(transform_dic, mode='train'):
    """dic is transform block of config.yaml"""
    """
    only patch_size and normalization is controlled by config.yaml
    """
    if transform_dic.get('use_config') is True:
        return get_multi_phase_transform_with_image_config(transform_dic, mode=mode)
    else:
        return get_multi_phase_transform_with_image_default(transform_dic, mode=mode)


def get_multi_phase_transform_with_image_default(transform_dic, mode='train'):
    """dic is transform block of config.yaml"""
    if mode == 'train':
        train_transforms = Compose(
            [
                LoadImaged(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"]),
                EnsureChannelFirstd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"]),
                Orientationd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], axcodes="RAI"),

                CTNormalizeD(keys=["I_M", "I_A"],
                             mean_intensity=transform_dic["normalize"]["mean"],
                             std_intensity=transform_dic["normalize"]["std"],
                             lower_bound=transform_dic["normalize"]["min"],
                             upper_bound=transform_dic["normalize"]["max"], ),
                Spacingd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], pixdim=transform_dic['spacing']
                         , mode=("nearest", "nearest", "nearest", "bilinear", "bilinear", "nearest", "nearest")),
                RandCropByPosNegLabeld(
                    keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"],
                    label_key="CS_DLGT",
                    spatial_size=tuple(transform_dic['patch_size']),
                    pos=1,
                    neg=0,
                    num_samples=2,
                    image_key="I_M",
                    image_threshold=0,
                    allow_smaller=True,
                ),
```

```
                # RandFlipd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], prob=0.5, spatial_axis=0),
                # RandFlipd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], prob=0.5, spatial_axis=1),
                # RandFlipd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], prob=0.5, spatial_axis=2),
                # # user can also add other random transforms
                # RandAffined(
                #     keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"],
                #     mode=('nearest', 'nearest', 'nearest', 'bilinear', "bilinear", "nearest", "nearest"),
                #     prob=0.5,
                #     rotate_range=(0, 0, np.pi / 15),
                #     scale_range=(0.1, 0.1, 0.1),
                #     padding_mode="zeros"
                # ),
            ]
        )
        val_transforms = Compose(
            [
                LoadImaged(keys=["CS_M", "label", "CS A", "I M", "I A", "CS DL", "CS DLGT"]),
                EnsureChannelFirstd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"]),
                Orientationd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], axcodes="RAI"),
                CTNormalizeD(keys=["I_M", "I_A"],
                             mean_intensity=transform_dic["normalize"]["mean"],
                             std_intensity=transform_dic["normalize"]["std"],
                             lower_bound=transform_dic["normalize"]["min"],
                             upper_bound=transform_dic["normalize"]["max"], ),
                Spacingd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], pixdim=transform_dic["spacing"],
                         mode=("nearest", "nearest", "nearest", "bilinear", "bilinear", "nearest", "nearest")),
            ]
        )
        save_transform = Compose(
            [
            ]
        )
        return train transforms, val_transforms, save_transform
    elif mode == 'infer':
        infer_transforms = Compose(
            [
                LoadImaged(keys=["CS_M", "CS A", "I M", "I A", "CS DL", "CS DLGT"]),
                EnsureChannelFirstd(keys=["CS_M", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"]),
                Orientationd(keys=["CS_M", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], axcodes="RAI"),
                CTNormalizeD(keys=["I_M", "I_A"],
                             mean_intensity=transform_dic["normalize"]["mean"],
                             std_intensity=transform_dic["normalize"]["std"],
                             lower_bound=transform_dic["normalize"]["min"],
                             upper_bound=transform_dic["normalize"]["max"], ),
                Spacingd(keys=["CS_M", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], pixdim=transform_dic["spacing"],
                         mode=("nearest", "nearest", "bilinear", "bilinear", "nearest", "nearest")),
            ]
        )
        return infer_transforms
    else:
        raise RuntimeError(f"{mode} is not supported yet")


def get_multi_phase_transform_with_image_config(transform_dic, mode='train'):
    """dic is transform block of config.yaml"""
    if transform_dic.get("image_resample") is not None:
        image_resample_mode = transform_dic["image_resample"]["mode"]
        image_resample_padding_mode = transform_dic["image_resample"]["padding_mode"]
    else:
        image_resample_mode = "bilinear"
        image_resample_padding_mode = "border"
    if mode == 'train':
        fixed_transforms = [
                LoadImaged(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"]),
                EnsureChannelFirstd(keys=["CS M", "label", "CS A", "I M", "I A", "CS DL", "CS_DLGT"]),
                Orientationd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], axcodes="RAI"),
                CTNormalizeD(keys=["I_M", "I_A"],
                             mean_intensity=transform_dic["normalize"]["mean"],
                             std_intensity=transform_dic["normalize"]["std"],
                             lower_bound=transform_dic["normalize"]["min"],
                             upper_bound=transform_dic["normalize"]["max"], ),
                Spacingd(keys=["CS M", "label", "CS A", "I M", "I_A", "CS_DL", "CS_DLGT"], pixdim=transform_dic['spacing']
                         , mode=("nearest", "nearest", "nearest", image_resample_mode, image_resample_mode, "nearest", "nearest")),
            ]
        print("----------------------second-stage fixed_transform-------------------------")
        if transform_dic.get("fixed") is not None and len(transform_dic.get("fixed")) > 0:
            for d in transform dic["fixed"]:
                key = d.get('name')
                value = d.get('parameter')
                assert key is not None, "name of fixed transform is not defined in config.yaml"
                assert value is not None, "parameter of fixed transform is not defined in config.yaml"
                print(f"name: {key}, parameter: {value}")
                trans_class = eval(key)
                trans_module = trans_class(**value)
                fixed_transforms.append(trans_module)

        random_transforms = []
        # assert transform_dic.get("random") is not None, "random transform is not defined in config.yaml"
        if transform_dic.get("random") is None or len(transform_dic.get("random")) == 0:
            print("warning: random transform is not defined in config.yaml")
        else:
            print("----------------------second-stage random_transform-------------------------")
            for d in transform_dic["random"]:
                key = d.get('name')
                value = d.get('parameter')
                assert key is not None, "name of random transform is not defined in config.yaml"
                assert value is not None, "parameter of random transform is not defined in config.yaml"
                print(f"name: {key}, parameter: {value}")
                trans_class = eval(key)
                trans_module = trans_class(**value)
                random_transforms.append(trans_module)

        train_transforms = Compose(fixed_transforms + random_transforms)
        val_transforms = Compose(fixed_transforms)
        save_transform = None
        return train transforms, val_transforms, save_transform
    elif mode == 'infer':
        infer_transforms = [
                LoadImaged(keys=["CS_M", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"]),
                EnsureChannelFirstd(keys=["CS M", "CS A", "I M", "I A", "CS DL", "CS_DLGT"]),
                Orientationd(keys=["CS_M", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], axcodes="RAI"),
                CTNormalizeD(keys=["I_M", "I_A"],
```

```
                            mean_intensity=transform_dic["normalize"]["mean"],
                            std_intensity=transform_dic["normalize"]["std"],
                            lower_bound=transform_dic["normalize"]["min"],
                            upper_bound=transform_dic["normalize"]["max"], ),
                Spacingd(keys=["CS_M", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], pixdim=transform_dic["spacing"],
                         mode=("nearest", "nearest", image_resample_mode, image_resample_mode, "nearest", "nearest")),
            ]
        print("----------------------second-stage infer_transform-------------------------")
        if transform_dic.get("infer") is not None and len(transform_dic.get("infer")) > 0:
            for d in transform_dic["infer"]:
                key = d.get('name')
                value = d.get('parameter')
                assert key is not None, "name of infer transform is not defined in config.yaml"
                assert value is not None, "parameter of infer transform is not defined in config.yaml"
                print(f"name: {key}, parameter: {value}")
                trans_class = eval(key)
                trans_module = trans_class(**value)
                infer_transforms.append(trans_module)

        infer_transforms = Compose(infer_transforms)
        return infer_transforms
    else:
        raise RuntimeError(f"{mode} is not supported yet")


def get_second_stage_only_one_phase(transform_dic, mode='train'):
    """dic is transform block of config.yaml"""
    """
    only patch_size and normalization is controlled by config.yaml
    """
    if transform_dic.get('use_config') is True:
        return get_second_stage_only_one_phase_default(transform_dic, mode=mode)
    else:
        return get_second_stage_only_one_phase_config(transform_dic, mode=mode)


def get_second_stage_only_one_phase_default(transform_dic, mode='train'):
    """dic is transform block of config.yaml"""
    if mode == 'train':
        train_transforms = Compose(
            [
                LoadImaged(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"]),
                EnsureChannelFirstd(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"]),
                Orientationd(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"], axcodes="RAI"),

                CTNormalizeD(keys=["I_M"],
                            mean_intensity=transform_dic["normalize"]["mean"],
                            std_intensity=transform_dic["normalize"]["std"],
                            lower_bound=transform_dic["normalize"]["min"],
                            upper_bound=transform_dic["normalize"]["max"], ),
                Spacingd(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"], pixdim=transform_dic['spacing']
                         , mode=("nearest", "nearest", "bilinear", "nearest", "nearest")),
                RandCropByPosNegLabeld(
                    keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"],
                    label_key="CS_DLGT",
                    spatial_size=tuple(transform_dic['patch_size']),
                    pos=1,
                    neg=0,
                    num_samples=2,
                    image_key="I_M",
                    image_threshold=0,
                    allow_smaller=True,
                ),
                # RandFlipd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], prob=0.5, spatial_axis=0),
                # RandFlipd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], prob=0.5, spatial_axis=1),
                # RandFlipd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], prob=0.5, spatial_axis=2),
                # # user can also add other random transforms
                # RandAffined(
                #     keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"],
                #     mode=('nearest', 'nearest', 'nearest', 'bilinear', "bilinear", "nearest", "nearest"),
                #     prob=0.5,
                #     rotate_range=(0, 0, np.pi / 15),
                #     scale_range=(0.1, 0.1, 0.1),
                #     padding_mode="zeros"
                # ),
            ]
        )
        val_transforms = Compose(
            [
                LoadImaged(keys=["CS_M", "label", "I M", "CS DL", "CS DLGT"]),
                EnsureChannelFirstd(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"]),
                Orientationd(keys=["CS M", "label", "I_M", "CS_DL", "CS_DLGT"], axcodes="RAI"),
                CTNormalizeD(keys=["I_M"],
                            mean_intensity=transform_dic["normalize"]["mean"],
                            std_intensity=transform_dic["normalize"]["std"],
                            lower_bound=transform_dic["normalize"]["min"],
                            upper_bound=transform_dic["normalize"]["max"], ),
                Spacingd(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"], pixdim=transform_dic["spacing"],
                         mode=("nearest", "nearest", "bilinear", "nearest", "nearest")),
            ]
        )
        save_transform = Compose(
            [
            ]
        )
        return train_transforms, val_transforms, save_transform
    elif mode == 'infer':
        infer_transforms = Compose(
            [
                LoadImaged(keys=["CS_M", "I M", "CS DL", "CS DLGT"]),
                EnsureChannelFirstd(keys=["CS_M", "I_M", "CS_DL", "CS_DLGT"]),
                Orientationd(keys=["CS M", "I_M", "CS_DL", "CS_DLGT"], axcodes="RAI"),
                CTNormalizeD(keys=["I_M"],
                            mean_intensity=transform_dic["normalize"]["mean"],
                            std_intensity=transform_dic["normalize"]["std"],
                            lower_bound=transform_dic["normalize"]["min"],
                            upper_bound=transform_dic["normalize"]["max"], ),
                Spacingd(keys=["CS_M", "I_M", "CS_DL", "CS_DLGT"], pixdim=transform_dic["spacing"],
                         mode=("nearest", "bilinear", "nearest", "nearest")),
            ]
        )
        return infer_transforms
```

```
    else:
        raise RuntimeError(f"{mode} is not supported yet")


def get_second_stage_only_one_phase_config(transform_dic, mode='train'):
    """dic is transform block of config.yaml"""
    if transform_dic.get("image_resample") is not None:
        image_resample_mode = transform_dic["image_resample"]["mode"]
        image_resample_padding_mode = transform_dic["image_resample"]["padding_mode"]
    else:
        image_resample_mode = "bilinear"
        image_resample_padding_mode = "border"
    if mode == 'train':
        fixed_transforms = [
                LoadImaged(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"]),
                EnsureChannelFirstd(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"]),
                Orientationd(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"], axcodes="RAI"),
                CTNormalizeD(keys=["I_M"],
                             mean_intensity=transform_dic["normalize"]["mean"],
                             std_intensity=transform_dic["normalize"]["std"],
                             lower_bound=transform_dic["normalize"]["min"],
                             upper_bound=transform_dic["normalize"]["max"], ),
                Spacingd(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"], pixdim=transform_dic['spacing']
                         , mode=("nearest", "nearest", image_resample_mode, "nearest", "nearest")),
            ]
        print("----------------------second-stage fixed_transform-------------------------")
        if transform_dic.get("fixed") is not None and len(transform_dic.get("fixed")) > 0:
            for d in transform_dic["fixed"]:
                key = d.get('name')
                value = d.get('parameter')
                assert key is not None, "name of fixed transform is not defined in config.yaml"
                assert value is not None, "parameter of fixed transform is not defined in config.yaml"
                print(f"name: {key}, parameter: {value}")
                trans_class = eval(key)
                trans_module = trans_class(**value)
                fixed_transforms.append(trans_module)

        random_transforms = []
        # assert transform_dic.get("random") is not None, "random transform is not defined in config.yaml"
        if transform_dic.get("random") is None or len(transform_dic.get("random")) == 0:
            print("warning: random transform is not defined in config.yaml")
        else:
            print("----------------------second-stage random_transform-------------------------")
            for d in transform_dic["random"]:
                key = d.get('name')
                value = d.get('parameter')
                assert key is not None, "name of random transform is not defined in config.yaml"
                assert value is not None, "parameter of random transform is not defined in config.yaml"
                print(f"name: {key}, parameter: {value}")
                trans_class = eval(key)
                trans_module = trans_class(**value)
                random_transforms.append(trans_module)

        train_transforms = Compose(fixed_transforms + random_transforms)
        val_transforms = Compose(fixed_transforms)
        save_transform = None
        return train_transforms, val_transforms, save_transform
    elif mode == 'infer':
        infer_transforms = [
                LoadImaged(keys=["CS_M", "I_M", "CS_DL", "CS_DLGT"]),
                EnsureChannelFirstd(keys=["CS_M", "I_M", "CS_DL", "CS_DLGT"]),
                Orientationd(keys=["CS_M", "I_M", "CS_DL", "CS_DLGT"], axcodes="RAI"),
                CTNormalizeD(keys=["I_M"],
                             mean_intensity=transform_dic["normalize"]["mean"],
                             std_intensity=transform_dic["normalize"]["std"],
                             lower_bound=transform_dic["normalize"]["min"],
                             upper_bound=transform_dic["normalize"]["max"], ),
                Spacingd(keys=["CS_M", "I_M", "CS_DL", "CS_DLGT"], pixdim=transform_dic["spacing"],
                         mode=("nearest", image_resample_mode, "nearest")),
            ]
        print("----------------------second-stage infer_transform-------------------------")
        if transform_dic.get("infer") is not None and len(transform_dic.get("infer")) > 0:
            for d in transform_dic["infer"]:
                key = d.get('name')
                value = d.get('parameter')
                assert key is not None, "name of infer transform is not defined in config.yaml"
                assert value is not None, "parameter of infer transform is not defined in config.yaml"
                print(f"name: {key}, parameter: {value}")
                trans_class = eval(key)
                trans_module = trans_class(**value)
                infer_transforms.append(trans_module)

        infer_transforms = Compose(infer_transforms)
        return infer_transforms
    else:
        raise RuntimeError(f"{mode} is not supported yet")
```

tree.txt

```
SparrowLink/
├── ablation_nnunet.sh
├── caculate_metric.py
├── configs/
│   └── two_stage2/
│       ├── first_stage/
│       │   └── ResUnet_dice_with_random.yaml
│       └── second_stage/
│           ├── nnunetv2.yaml
│           ├── ResUnet_dice_with_random.yaml
│           └── ResUnet_dice_with_random_1.yaml
├── data/
│   └── loader.py
├── loss_zoo/
│   ├── cldice.py
│   └── soft_skeleton.py
├── main.py
├── model/
│   ├── CS2net.py
│   ├── denseunet_skip.py
│   └── swin_unet.py
├── modify_key_in_config.py
├── post_processing/
│   ├── fracture_detection.py
│   ├── move_file.py
│   └── select_two_region.py
├── README.md
├── registration/
│   ├── basic_registration.py
│   ├── label_registration.py
│   ├── SparrowLinkv3.sh
│   └── SparrowLinkv3_nnUnet.sh
├── second_stage_main.py
├── second_stage_only_one_phase_main.py
├── slicer_visulization/
│   ├── slicer_mark_up.py
│   └── slicer_mvf.py
├── SparrowLink_metric.py
├── SparrowLink_Post_Process.py
├── SparrowLinkv3.sh
├── SparrowLinkv3_nnUnet.sh
├── SparrowLinkv3_nnUnet_ASOCA.sh
├── transform/
│   ├── Anatomical_augumentation_for_CCTA_images.md
│   ├── AnatomyTransformD.py
│   ├── cardiac_transformation.py
│   ├── fast_crop.py
│   ├── IntensityTransformD.py
│   ├── NoiseTransformD.py
│   ├── SpatialTransformationD.py
│   ├── test_contrast_reduce.ipynb
│   ├── test_motion_artifact.ipynb
│   ├── try_HeartTransformD.ipynb
│   └── utils.py
└── utils/
    ├── Config.py
    ├── get_module.py
    ├── get_network_from_plans_nnunet.py
    ├── inferer.py
    ├── test.py
    └── trainer.py
```