

```

0 SparrowLink\caculate_metric.py
1 from monai.transforms import (
2     LoadImage,
3     SaveImage,
4 )
5 from tqdm import tqdm
6 from monai.data.meta_tensor import MetaTensor
7 import monai
8 import numpy as np
9 from openpyxl import Workbook
10 from metric_zoo import cLDice
11 import seg_metrics.seg_metrics as sg
12 import argparse
13 import pathlib
14
15 dice_metric = monai.metrics.DiceMetric(include_background=True, reduction="mean") # only label channel
16 hausdorff_metric = monai.metrics.HausdorffDistanceMetric(include_background=True, reduction="mean")
17 asd_metric = monai.metrics.SurfaceDistanceMetric(include_background=True, symmetric=True)
18 iou_metric = monai.metrics.MeanIoU(include_background=True, reduction="mean")
19
20
21 def update(pbar, record, result):
22     pbar.update()
23     record.append(result)
24
25
26 def error_back(err):
27     print(err)
28
29
30 def caculate_metric(label_root, seg_root):
31     label = LoadImage(image_only=False)(str(label_root))[0]
32     seg = LoadImage(image_only=False)(str(seg_root))[0]
33     if label.sum() == 0:
34         result = {"name": [label_root.name], "metric": [1000, 1000, 1000,
35                                                         1000, 1000]}
36         return result
37     if seg.sum() == 0:
38         result = {"name": [label_root.name], "metric": [0, 0, 0,
39                                                         1000, 1000]}
40         return result
41     dice = dice_metric(y_pred=seg[None, None], y=label[None, None])
42     cldice = cLDice(seg, label)
43     iou = iou_metric(y_pred=seg[None, None], y=label[None, None])
44     spacing = label.meta["pixdim"][1:4].tolist()
45     distance_metrics = sg.write_metrics(labels=[1], # exclude background if needed
46                                       gdth_img=np.array(label),
47                                       pred_img=np.array(seg),
48                                       csv_file=None,
49                                       spacing=spacing,
50                                       metrics=['msd', 'hd95'],
51                                       verbose=False)
52     result = {"name": [label_root.name], "metric": [dice.item(), cldice.item(), iou.item(),
53                                                     distance_metrics[0]['msd'][0], distance_metrics[0]['hd95'][0]]}
54     return result
55
56
57 if __name__ == '__main__':
58     parser = argparse.ArgumentParser()
59     parser.add_argument('--seg_path', type=str, default=None)
60     parser.add_argument('--label_path', type=str, default=None)
61     parser.add_argument('--metric_result_path', type=str, default=None)
62     parser.add_argument('--label_find', type=str, default="*.nii.gz")
63     parser.add_argument('--seg_find', type=str, default="*.nii.gz")
64     parser.add_argument("--multiprocess", action='store_true', default=False)
65     args = parser.parse_args()
66     assert args.seg_path is not None, "seg path is None"
67     assert args.label_path is not None, "label path is None"
68     if args.metric_result_path is None:
69         metric_result_path = str(pathlib.Path(args.seg_path) / "metric_result.xlsx")
70     else:
71         metric_result_path = args.metric_result_path
72         pathlib.Path(metric_result_path).parent.mkdir(parents=True, exist_ok=True)
73     # caculate the dice metric of the segmentation result
74
75     label_list = list(pathlib.Path(args.label_path).glob(args.label_find))
76     seg_list = list(pathlib.Path(args.seg_path).glob(args.seg_find))
77     # seg_list = [name for name in seg_list if 'auxiliary' not in name]
78     label_list.sort()
79     seg_list.sort()
80     # save the image name, dice metric in a excel file
81     # new a excel file
82     wb = Workbook()
83
84     ws = wb.active
85     ws.append(['image_name', 'dice_metric', 'cldice', 'mIoU', 'MSD', 'HD95'])
86     metric_list = []
87
88     print(f"\033[96m calculating metric for {args.seg_find} \033[00m")
89     assert len(seg_list) > 0, f"no file found in {args.seg_path} {args.seg_find}"
90     pbar = tqdm(total=len(label_list), colour='#87cefa')
91     if not args.multiprocess:
92         for label_root, seg_root in zip(label_list, seg_list):
93             # print(label_root.split("/")[-1], seg_root.split("/")[-1])
94             result_dict = caculate_metric(label_root, seg_root)
95             ws.append(result_dict["name"] + result_dict["metric"])
96             metric_list.append(result_dict["metric"])
97             pbar.set_description(f'metric:{str(seg_root.name), result_dict["metric"][0]}')
98             pbar.update()
99     else:
100         from multiprocessing import Pool
101         pool = Pool(14)
102         result_record = []
103         for label_root, seg_root in zip(label_list, seg_list):
104             pool.apply_async(func=caculate_metric,
105                             args=(label_root, seg_root),
106                             error_callback=error_back,
107                             callback=lambda x: update(pbar, result_record, x))
108         pool.close()
109         pool.join()
110         # sort the result by "name"

```

```

109     # sort the result by "name"
110     result_record.sort(key=lambda x: x.get("name"))
111     for result in result_record:
112         ws.append(result.get("name") + result.get("metric"))
113         metric_list.append(result.get("metric"))
114
115     metric_array = np.array(metric_list)
116     mean_metric = metric_array.mean(axis=0).tolist()
117     mean_metric.insert(0, 'mean')
118     ws.append(mean_metric)
119     wb.save(metric_result_path)

```

H:\gitRepo2PDF\dist\repo2pdfAPP\SparrowLink\data\loader.py

```

0 SparrowLink/data/loader.py
1 import json
2 import os
3 import glob
4 import random
5 import numpy as np
6 import pathlib
7
8
9 def five_fold_generator(L):
10     """
11     :param L: a list
12     :return: five_fold: shape 5*1
13     """
14     length = len(L)
15     index = list(range(length))
16     random.shuffle(index)
17     d = length // 5
18     five_fold = []
19     print(f"rand_index:{index}")
20     for i in range(5):
21         train_index = index[: (4 - i) * d] + index[(5 - i) * d:] if i > 0 else index[: (4 - i) * d]
22         val_index = index[(4 - i) * d: (5 - i) * d] if i > 0 else index[(4 - i) * d:]
23         five_fold.append({"train_files": [L[k] for k in train_index],
24                           "val_files": [L[k] for k in val_index]})
25     return five_fold
26     # print(f"{i}: train: {five_fold_index[i]['train']}, val: {five_fold_index[i]['val']}")
27
28
29 def shuffle_generator(L):
30     """
31     :param L: a list
32     :return: shuffle_fold: 20% randomly selected sample form origin dataset
33     """
34     length = len(L)
35     index = list(range(length))
36     random.shuffle(index)
37     train_index, val_index = index[:int(0.8*length)], index[int(0.8*length):]
38     shuffle_fold = [{"train_files": [L[k] for k in train_index],
39                      "val_files": [L[k] for k in val_index]}]
40
41     return shuffle_fold
42
43
44 def generator_multi_dataset(L1, L2, split_mode='shuffle'):
45     """
46     :split_mode:
47     :param L1, L2: a list, diast, syst
48     :return: shuffle_fold: 20% randomly selected sample form origin dataset
49     """
50     length = len(L1)
51     assert len(L1) == len(L2), 'length of two list are different '
52     train_index, val_index = index_generator(length, split_mode=split_mode)
53     fold = []
54     for i in range(len(train_index)):
55         fold_l1 = {"train_files": [L1[k] for k in train_index[i]],
56                   "val_files": [L1[k] for k in val_index[i]]}
57         fold_l2 = {"train_files": [L2[k] for k in train_index[i]],
58                   "val_files": [L2[k] for k in val_index[i]]}
59         fold.append({"train_files": fold_l1["train_files"] + fold_l2["train_files"],
60                     "val_files": fold_l1["val_files"] + fold_l2["val_files"], })
61     return fold
62
63
64 def prepare_datalist(image_file="images", label_file="label", split_mode='order'):
65     """
66     :param image_file: the name of image file
67     :param label_file: the name of label file
68     :return: train_files, val_files. Now we separate them directly. It needs to be modified for 5-fold validation.
69     """
70
71     image_path = pathlib.Path(image_file)
72     label_path = pathlib.Path(label_file)
73     assert image_path.is_dir(), f"img path not exist: {image_path}"
74     assert label_path.is_dir(), f"label path not exist: {label_path}"
75     # get the image name list in image_path using pathlib
76     train_images = [path.name for path in image_path.glob("*.nii.gz")]
77     data_dicts = [
78         {"image": str(pathlib.Path(image_path, image_name)), "label": str(pathlib.Path(label_path, image_name))}
79         for image_name in train_images
80     ]
81     if split_mode == 'five_fold':
82         return five_fold_generator(data_dicts)
83     elif split_mode == 'order':
84         length = len(data_dicts)
85         index = int(0.2 * length)
86         train_files, val_files = data_dicts[: -index], data_dicts[-index:]
87         return [{"train_files": train_files, "val_files": val_files}]
88     elif split_mode == 'shuffle':
89         length = len(data_dicts)
90         index = int(0.2 * length)
91         train_files, val_files = data_dicts[: -index], data_dicts[-index:]
92         return [{"train_files": train_files, "val_files": val_files}]
93     elif split_mode == 'all':
94         return data_dicts
95     else:
96         raise RuntimeError(f"{split_mode} is not supported")
97

```

```

98
99 def prepare_datalist_with_file(image_file="images", label_file="label",img_name="", ):
100     """
101     :param image_file: the name of image file
102     :param label_file: the name of label file
103     :return: train_files, val_files. Now we separate them directly. It needs to be modified for 5-fold validation.
104     """
105     # use json load image name list form img_name, which is a json file
106     with open(img_name, 'r') as f:
107         name_list = json.load(f)
108     image_path = pathlib.Path(image_file)
109     label_path = pathlib.Path(label_file)
110     assert os.path.isdir(image_path), f"img path not exist: {image_path}"
111     assert os.path.isdir(label_path), f"label path not exist: {label_path}"
112     data_dicts = [
113         {"image": str(pathlib.Path(image_path, name)), "label": str(pathlib.Path(label_path, name))}
114         for name in name_list
115     ]
116     return data_dicts
117
118
119 def prepare_multi_datalist_with_file(main_file, auxiliary_file, label_file, broken_file, broken_gt_file, img_name=None, select_file=None):
120     """
121     :param main_file: the root of data file.
122     :param auxiliary_file: the name of image file
123     :param label_file: the name of label file
124     :param broken_file: the name of broken image file
125     :param broken_gt_file: the name of broken gt file
126     :param select_file: the name of selected file, contains the name of selected image and the number broken part
127     :return: train_files, val_files. Now we separate them directly. It needs to be modified for 5-fold validation.
128     """
129     # use json load image name list form img_name, which is a json file
130
131     if img_name is not None:
132         with open(img_name, 'r') as f:
133             name_list = json.load(f)
134     else:
135         name_list = pathlib.Path(main_file).glob("*.nii.gz")
136         name_list = [path.name for path in name_list]
137         # sort the name list
138         name_list.sort()
139     if select_file is not None:
140         with open(select_file, 'r') as f:
141             select_list = json.load(f)
142             select_list = [path["name"] for path in select_list if path["num"] > 0]
143             name_list = [name for name in name_list if name.replace(".nii.gz", "") in select_list]
144
145     print(f"num:{len(name_list)}")
146     main_path = pathlib.Path(main_file)
147     auxiliary_path = pathlib.Path(auxiliary_file)
148     label_path = pathlib.Path(label_file)
149     assert os.path.isdir(main_path), f"img path not exist: {main_path}"
150     assert os.path.isdir(label_path), f"label path not exist: {label_path}"
151     assert os.path.isdir(auxiliary_path), f"img path not exist: {auxiliary_path}"
152     assert os.path.isdir(broken_file), f"img path not exist: {broken_file}"
153     assert os.path.isdir(broken_gt_file), f"img path not exist: {broken_gt_file}"
154     # because we do not have full data
155     data_dicts = [
156         {"main": str(pathlib.Path(main_path, name)),
157          "auxiliary": str(pathlib.Path(auxiliary_path, name)),
158          "label": str(pathlib.Path(label_path, name)),
159          "broken": str(pathlib.Path(broken_file, name)),
160          "broken_gt": str(pathlib.Path(broken_gt_file, name))},
161         for name in name_list if pathlib.Path(main_path, name).exists() and pathlib.Path(auxiliary_path, name).exists()
162     ]
163     return data_dicts
164
165
166 def prepare_main_auxiliary_with_img_datalist_with_file(main_file,
167                                                         auxiliary_file,
168                                                         main_img_file,
169                                                         auxiliary_img_file,
170                                                         broken_file,
171                                                         broken_gt_file,
172                                                         label_file=None,
173                                                         img_name="",
174                                                         select_file=None):
175     """
176     :param main_file: the path of coarse segmentation in main phase.
177     :param auxiliary_file: the path of refined segmentation in auxiliary phase.
178     :param main_img_file: the path of image in main phase.
179     :param auxiliary_img_file: the path of image in auxiliary phase.
180     :param broken_file: the path of broken spheres in main phase.
181     :param broken_gt_file: the path of broken spheres generated with gt in main phase.
182     :param label_file: the path of label file.
183     :param img_name: the name of image file.
184     :param select_file: record the number of discontinuity sphere.
185     """
186     # use json load image name list form img_name, which is a json file
187     if img_name is not None:
188         with open(img_name, 'r') as f:
189             name_list = json.load(f)
190     else:
191         name_list = pathlib.Path(main_file).glob("*.nii.gz")
192         name_list = [path.name for path in name_list]
193         # sort the name list
194         name_list.sort()
195
196     if select_file is not None:
197         with open(select_file, 'r') as f:
198             select_list = json.load(f)
199             select_list = [path["name"] for path in select_list if path["num"] > 0]
200             name_list = [name for name in name_list if name.replace(".nii.gz", "") in select_list]
201
202     main_path = pathlib.Path(main_file)
203     auxiliary_path = pathlib.Path(auxiliary_file)
204     main_image_path = pathlib.Path(main_img_file)
205     auxiliary_image_path = pathlib.Path(auxiliary_img_file)
206     assert os.path.isdir(main_path), f"main path not exist: {main_path}"
207     assert os.path.isdir(main_image_path), f"main image path not exist: {main_image_path}"
208     assert os.path.isdir(auxiliary_image_path), f"auxiliary image path not exist: {auxiliary_image_path}"

```

```

209 assert os.path.isdir(auxiliary_path), f"auxiliary_path not exist: {auxiliary_path}"
210 assert os.path.isdir(broken_file), f"broken_file not exist: {broken_file}"
211 assert os.path.isdir(broken_gt_file), f"broken_gt_file not exist: {broken_gt_file}"
212 if label_file is None:
213     data_dicts = [
214         {"CS_M": str(pathlib.Path(main_path, name)),
215          "CS_A": str(pathlib.Path(auxiliary_path, name)),
216          "I_M": str(pathlib.Path(main_image_path, name)),
217          "I_A": str(pathlib.Path(auxiliary_image_path, name)),
218          "CS_DL": str(pathlib.Path(broken_file, name)),
219          "CS_DLGT": str(pathlib.Path(broken_gt_file, name)), }
220         for name in name_list
221     ]
222     return data_dicts
223 else:
224     label_path = pathlib.Path(label_file)
225     assert os.path.isdir(label_path), f"label path not exist: {label_path}"
226     data_dicts = [
227         {"CS_M": str(pathlib.Path(main_path, name)),
228          "CS_A": str(pathlib.Path(auxiliary_path, name)),
229          "I_M": str(pathlib.Path(main_image_path, name)),
230          "I_A": str(pathlib.Path(auxiliary_image_path, name)),
231          "label": str(pathlib.Path(label_path, name)),
232          "CS_DL": str(pathlib.Path(broken_file, name)),
233          "CS_DLGT": str(pathlib.Path(broken_gt_file, name)), }
234         for name in name_list
235     ]
236     return data_dicts
237
238
239 def prepare_main_with_img_datalist_with_file(main_file,
240                                             main_img_file,
241                                             broken_file,
242                                             broken_gt_file,
243                                             label_file=None,
244                                             img_name="",
245                                             select_file=None):
246     """
247     :param main_file: the path of coarse segmentation in main phase.
248     :param auxiliary_file: the path of refined segmentation in auxiliary phase.
249     :param main_img_file: the path of image in main phase.
250     :param auxiliary_img_file: the path of image in auxiliary phase.
251     :param broken_file: the path of broken spheres in main phase.
252     :param broken_gt_file: the path of broken spheres generated with gt in main phase.
253     :param label_file: the path of label file.
254     :param img_name: the name of image file.
255     :param select_file: record the number of discontinuity sphere.
256     """
257     # use json load image name list form img_name, which is a json file
258     if img_name is not None:
259         with open(img_name, 'r') as f:
260             name_list = json.load(f)
261     else:
262         name_list = pathlib.Path(main_file).glob("*.nii.gz")
263         name_list = [path.name for path in name_list]
264         # sort the name list
265         name_list.sort()
266
267     if select_file is not None:
268         with open(select_file, 'r') as f:
269             select_list = json.load(f)
270             select_list = [path["name"] for path in select_list if path["num"] > 0]
271             name_list = [name for name in name_list if name.replace(".nii.gz", "") in select_list]
272
273     main_path = pathlib.Path(main_file)
274     main_image_path = pathlib.Path(main_img_file)
275     assert os.path.isdir(main_path), f"main_path not exist: {main_path}"
276     assert os.path.isdir(main_image_path), f"main image path not exist: {main_image_path}"
277     assert os.path.isdir(broken_file), f"broken_file not exist: {broken_file}"
278     assert os.path.isdir(broken_gt_file), f"broken_gt_file not exist: {broken_gt_file}"
279     if label_file is None:
280         data_dicts = [
281             {"CS_M": str(pathlib.Path(main_path, name)),
282              "I_M": str(pathlib.Path(main_image_path, name)),
283              "CS_DL": str(pathlib.Path(broken_file, name)),
284              "CS_DLGT": str(pathlib.Path(broken_gt_file, name)), }
285             for name in name_list
286         ]
287         return data_dicts
288     else:
289         label_path = pathlib.Path(label_file)
290         assert os.path.isdir(label_path), f"label path not exist: {label_path}"
291         data_dicts = [
292             {"CS_M": str(pathlib.Path(main_path, name)),
293              "I_M": str(pathlib.Path(main_image_path, name)),
294              "label": str(pathlib.Path(label_path, name)),
295              "CS_DL": str(pathlib.Path(broken_file, name)),
296              "CS_DLGT": str(pathlib.Path(broken_gt_file, name)), }
297             for name in name_list
298         ]
299         return data_dicts
300
301
302
303 def index_generator(length, split_mode='shuffle'):
304     """
305     :length: list length
306     """
307     train_index = []
308     val_index = []
309     index = list(range(length))
310     if split_mode == 'shuffle':
311         random.shuffle(index)
312         train_index.append(index[:int(0.8 * length)])
313
314         val_index.append(index[int(0.8 * length):])
315
316     elif split_mode == 'five_fold':
317         # random.shuffle(index)
318         d = length // 5
319         for i in range(5):

```

```

320         train_index.append(index[(4 - i) * d] + index[(5 - i) * d:] if i > 0 else index[: (4 - i) * d])
321         val_index.append(index[(4 - i) * d:(5 - i) * d] if i > 0 else index[(4 - i) * d:])
322
323     elif split_mode == 'order':
324         train_index.append(index[:int(0.8 * length)])
325         val_index.append(index[int(0.8 * length):])
326     else:
327         raise RuntimeError(f"{split_mode} is not supported")
328     return train_index, val_index
329
330
331 def prepare_datalist_with_heart_label(data_dir, image_file="images", label_file="label", heart_file="heart"):
332     """
333     :param data_dir: the root of data file.
334     :param image_file: the name of image file
335     :param label_file: the name of label file
336     :return: train_files, val_files. Now we separate them directly. It needs to be modified for 5-fold validation.
337     """
338
339     image_path = os.path.join(data_dir, image_file)
340     label_path = os.path.join(data_dir, label_file)
341     heart_path = os.path.join(data_dir, heart_file)
342     assert os.path.isdir(image_path), "img path not exist"
343     assert os.path.isdir(label_path), "label path not exist"
344     assert os.path.isdir(heart_path), "label path not exist"
345
346     train_images = sorted(glob.glob(os.path.join(image_path, "*.nii.gz")))
347     train_labels = sorted(glob.glob(os.path.join(label_path, "*.nii.gz")))
348     train_heart = sorted(glob.glob(os.path.join(heart_path, "*.nii.gz")))
349
350     data_dicts = [
351         {"image": image_name, "label": label_name, "heart": heart_seg}
352         for image_name, label_name, heart_seg in zip(train_images, train_labels, train_heart)
353     ]
354     return data_dicts
355
356
357 def write_data_reference(L, save_path):
358     """a list contain val files and train files"""
359     with open(os.path.join(save_path, 'train_set.txt'), 'w') as f:
360         for dic in L['train_files']:
361             for key, file in dic.items():
362                 f.write(os.path.split(file)[-1] + '\n')
363             break
364
365     with open(os.path.join(save_path, 'val_set.txt'), 'w') as f:
366         for dic in L['val_files']:
367             for key, file in dic.items():
368                 f.write(os.path.split(file)[-1] + '\n')
369             break
370
371
372 def save_json(L, save_path):
373     with open(save_path, 'w') as f:
374         json.dump(L, f)
375
376
377 def load_json(path):
378     with open(path, 'r') as f:
379         x = json.load(f)
380     return x
381
382
383 def prepare_image_list(image_path):
384     """
385     generate image list in a dir
386     """
387     assert os.path.isdir(image_path), "img path not exist"
388     train_images = sorted(glob.glob(os.path.join(image_path, "*.nii.gz")))
389     data_dicts = [
390         {"image": image_name}
391         for image_name in train_images
392     ]
393     return data_dicts
394
395
396 if __name__ == '__main__':
397     L = list(range(10))
398     print(shuffle_generator(L))

```

```

0 SparrowLink/loss_zoo/cldice.py
1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4 from .soft_skeleton import soft_skel
5 from monai.losses import DiceLoss
6 from monai.networks import one_hot
7
8
9 class soft_cldice(nn.Module):
10     def __init__(self, iter=3, smooth=1.):
11         super(soft_cldice, self).__init__()
12         self.iter = iter
13         self.smooth = smooth
14
15     def forward(self, y_true, y_pred):
16         skel_pred = soft_skel(y_pred, self.iter)
17         skel_true = soft_skel(y_true, self.iter)
18         tprec = (torch.sum(torch.multiply(skel_pred, y_true)[:1, ...]) + self.smooth) / (torch.sum(skel_pred[:1, ...]) + smooth)
19         tsens = (torch.sum(torch.multiply(skel_true, y_pred)[:1, ...]) + self.smooth) / (torch.sum(skel_true[:1, ...]) + smooth)
20         cl_dice = 1. - 2.0 * (tprec * tsens) / (tprec + tsens)
21         return cl_dice
22
23
24 def soft_dice(y_true, y_pred):
25     """(function to compute dice loss)
26     Args:
27         y_true ([float32]): [ground truth image]
28         y_pred ([float32]): [predicted image]
29     Returns:
30         [float32]: [loss value]
31     """
32     smooth = 1
33     intersection = torch.sum((y_true * y_pred)[:1, ...])
34     coeff = (2. * intersection + smooth) / (torch.sum(y_true[:1, ...]) + torch.sum(y_pred[:1, ...]) + smooth)
35     return (1. - coeff)
36
37
38 class soft_dice_cldice(nn.Module):
39     def __init__(self, iter=3, alpha=0.5, smooth=1.):
40         super(soft_dice_cldice, self).__init__()
41         self.iter = iter
42         self.smooth = smooth
43         self.alpha = alpha
44         self.dice = DiceLoss(to_onehot_y=False, softmax=False)
45
46     def forward(self, y_pred, y_true):
47         y_true = one_hot(y_true, num_classes=2)
48         y_pred = F.softmax(y_pred, dim=1)
49         dice = self.dice(y_pred, y_true)
50         skel_pred = soft_skel(y_pred, self.iter)
51         skel_true = soft_skel(y_true, self.iter)
52         tprec = (torch.sum(torch.multiply(skel_pred, y_true)[:1, ...]) + self.smooth) / (torch.sum(skel_pred[:1, ...]) + self.smooth)
53         tsens = (torch.sum(torch.multiply(skel_true, y_pred)[:1, ...]) + self.smooth) / (torch.sum(skel_true[:1, ...]) + self.smooth)
54         cl_dice = 1. - 2.0 * (tprec * tsens) / (tprec + tsens)
55         return (1.0 - self.alpha) * dice + self.alpha * cl_dice
56
57
58 class soft_dice_cldice_weighted(nn.Module):
59     def __init__(self, iter=3, alpha=0.5, smooth=1.):
60         super(soft_dice_cldice_weighted, self).__init__()
61         self.iter = iter
62         self.smooth = smooth
63         self.alpha = alpha
64         self.dice = DiceLoss(to_onehot_y=False, softmax=False, batch=True, reduction="none")
65
66     def forward(self, y_pred, y_true, weight):
67         # y_true = one_hot(y_true, num_classes=2)
68         # y_pred = F.softmax(y_pred, dim=1)
69         intersection = torch.sum((y_true * y_pred)[:1, ...], dim=(1, 2, 3, 4), keepdim=True)
70         coeff = (2. * intersection + self.smooth) / (torch.sum(y_true[:1, ...], dim=(1, 2, 3, 4), keepdim=True) +
71                                                    torch.sum(y_pred[:1, ...], dim=(1, 2, 3, 4), keepdim=True) + self.smooth)
72         dice = 1.0 - coeff
73         skel_pred = soft_skel(y_pred, self.iter)
74         skel_true = soft_skel(y_true, self.iter)
75         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), keepc
76         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), keepc
77         cl_dice = 1. - 2.0 * (tprec * tsens) / (tprec + tsens)
78         dice = torch.mean(dice * weight)
79         cl_dice = torch.mean(cl_dice * weight)
80         return (1.0 - self.alpha) * dice + self.alpha * cl_dice

```

H:\git\Repo2PDF\dist\repo2pdf\APP\SparrowLink\loss_zoo\soft_skeleton.py

```

0 SparrowLink/loss_zoo/soft_skeleton.py
1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4
5
6 def soft_erode(img):
7     if len(img.shape) == 4:
8         p1 = -F.max_pool2d(-img, (3,1), (1,1), (1,0))
9         p2 = -F.max_pool2d(-img, (1,3), (1,1), (0,1))
10        return torch.min(p1, p2)
11    elif len(img.shape) == 5:
12        p1 = -F.max_pool3d(-img, (3,1,1), (1,1,1), (1,0,0))
13        p2 = -F.max_pool3d(-img, (1,3,1), (1,1,1), (0,1,0))
14        p3 = -F.max_pool3d(-img, (1,1,3), (1,1,1), (0,0,1))
15        return torch.min(torch.min(p1, p2), p3)
16
17
18 def soft_dilate(img):
19     if len(img.shape) == 4:
20        return F.max_pool2d(img, (3,3), (1,1), (1,1))
21    elif len(img.shape) == 5:
22        return F.max_pool3d(img, (3,3,3), (1,1,1), (1,1,1))
23
24
25 def soft_open(img):
26     return soft_dilate(soft_erode(img))
27
28
29 def soft_skel(img, iter):
30     img1 = soft_open(img)
31     skel = F.relu(img - img1)
32     for j in range(iter):
33         img = soft_erode(img)
34         img1 = soft_open(img)
35         delta = F.relu(img - img1)
36         skel = skel + F.relu(delta - skel * delta)
37     return skel

```

```

0 SparrowLink/main.py
1 from monai.utils import first, set_determinism
2 from monai.data import CacheDataset, DataLoader, Dataset, decollate_batch, ITKReader, PersistentDataset
3 from monai.config import print_config
4 from monai.metrics.meandice import DiceMetric
5 import torch
6 import matplotlib.pyplot as plt
7 import os
8 import argparse
9 import monai
10 import logging
11 from utils.Config import Config
12 import sys
13 import time
14 from data_loader import prepare_datalist, prepare_datalist_with_file, prepare_image_list, save_json, write_data_reference, load_json
15 from transform_utils import get_transform
16 from utils.test import cardio_vessel_segmentation_test
17 from utils.trainer import cardio_vessel_segmentation_train
18 from utils.inferer import cardio_vessel_segmentation_infer
19 from pre_processing.checking import DatasetInformationExtractor
20 import pathlib
21 # print_config()
22
23
24 torch.multiprocessing.set_sharing_strategy('file_system')
25
26
27 if __name__ == "__main__":
28
29     parser = argparse.ArgumentParser(description="Run a basic UNet segmentation baseline.")
30     parser.add_argument("--configs",
31                         dest="cfg",
32                         help="The configs file.",
33                         default='./configs/heart_server.yaml',
34                         type=str)
35     parser.add_argument('--iters', dest='iters', help='Iterations in training.', type=int, default=None)
36     parser.add_argument('--batch_size', dest='batch_size', help='Mini batch size of one gpu or cpu.', type=int, default=None)
37     parser.add_argument('--learning_rate', dest='learning_rate', help='Learning rate', type=float, default=None)
38     parser.add_argument('--seed', dest='seed', help='seed', type=float, default=0)
39     parser.add_argument('--mode', dest='mode', help='train, infer or both', type=str, default=None)
40
41     parser.add_argument('--experiments_path', dest='experiments_path',
42                         help='experiments_path, all output and checkpoint are saved here.', type=str, default=None)
43
44     parser.add_argument('--pretrain_weight_path', help='pretrain weight for training, testing of inferring', type=str, default=None)
45     parser.add_argument('--img_path', help='image path', type=str, default=None)
46     parser.add_argument('--label_path', help='label path', type=str, default=None)
47     parser.add_argument('--output_path', help='save path', type=str, default=None)
48     parser.add_argument('--persist_path', help='persist path, the path used to save persist data in persist loader',
49                         type=str, default=None)
50     parser.add_argument('--val_set', help='val set, a txt path to select val data, only used if mode is train',
51                         type=str, default=None)
52     parser.add_argument('--train_set', help='train set, a txt path to select train data, only used if mode is train',
53                         type=str, default=None)
54     args = parser.parse_args()
55
56     monai.config.print_config()
57     logging.basicConfig(stream=sys.stdout, level=logging.INFO)
58     cfg = Config(
59         args.cfg,
60         learning_rate=args.learning_rate,
61         iters=args.iters,
62         batch_size=args.batch_size,
63         seed=args.seed,
64         mode=args.mode,
65         img_path=args.img_path,
66         label_path=args.label_path,
67         output_path=args.output_path,
68         persist_path=args.persist_path,
69         val_set=args.val_set,
70         train_set=args.train_set,
71         experiments_path=args.experiments_path,
72         pretrain_weight_path=args.pretrain_weight_path,
73     )
74     set_determinism(seed=cfg.seed)
75     if cfg.dic['mode'] == 'train':
76
77         # ----- built transform sequence ----- #
78         # ----- Create Model, Loss, Optimizer in Config ----- #
79         cfg.create_training_require()
80         train_transforms, val_transforms, save_transform = get_transform(cfg.dic['transform'])
81         if cfg.dic["train"]["loader"].get("file_path"):
82             files = [load_json(cfg.dic["train"]["loader"]["file_path"])]
83
84         elif cfg.dic["train"]["loader"].get("val_set"):
85             val_files = prepare_datalist_with_file(image_file=cfg.train_img_path,
86                                                  label_file=cfg.train_label_path,
87                                                  img_name=cfg.val_set, )
88             train_files = prepare_datalist_with_file(image_file=cfg.train_img_path,
89                                                    label_file=cfg.train_label_path,
90                                                    img_name=cfg.train_set, )
91             files = [{"train_files": train_files, "val_files": val_files}]
92         else:
93             files = prepare_datalist(image_file=cfg.train_img_path,
94                                    label_file=cfg.train_label_path,
95                                    split_mode=cfg.dic['train']['loader']['split_mode'], )
96         # ----- create loss function ----- #
97         # ----- you can crate your own loss or metric here amd replace cfg ----- #
98         # dice_metric = DiceMetric(include_background=False, reduction="mean")
99         # loss_function = DiceLoss(to_onehot_y=True, softmax=True)
100
101         # ----- training ----- #
102         metric_record = []
103         experiment_path = os.path.join(cfg.dic['experiments_path'], time.strftime("%d_%m_%Y_%H_%M_%S"))\
104             if cfg.dic.get('time_name', None) else cfg.dic['experiments_path']
105
106         if not os.path.exists(experiment_path):
107             os.makedirs(experiment_path)
108         for i in range(len(files)):
109             if cfg.dic['train']['loader'].get('split_mode') == "five_fold":
110                 experiment_path_fold = os.path.join(experiment_path, f"{i}_fold")
111                 if not os.path.exists(experiment_path_fold):
112                     os.makedirs(experiment_path_fold)
113             else:
114                 experiment_path_fold = experiment_path
115             write_data_reference(files[i], experiment_path_fold)
116             save_json(files[i], os.path.join(experiment_path_fold, 'files.txt'))
117             # ----- save config ----- #

```

```

118     cfg.save_config(os.path.join(experiment_path_fold, 'configs.yaml'))
119     if cfg.dic['train']['loader'].get('split mode') == "five fold":
120         print(f"-----fold{i} start!-----")
121     else:
122         print(f"-----training start!-----")
123     if cfg.dic['train']['loader'].get('persist'):
124         print("----- using persist dataset -----")
125         if cfg.persist_path == 'default':
126             persistent_cache = pathlib.Path(experiment_path_fold, "persistent_cache")
127         else:
128             persistent_cache = pathlib.Path(cfg.persist_path)
129
130         persistent_cache.mkdir(parents=True, exist_ok=True)
131         train_ds = PersistentDataset(data=files[i]['train_files'], transform=train_transforms,
132                                     cache_dir=persistent_cache)
133         val_ds = PersistentDataset(data=files[i]['val_files'], transform=val_transforms,
134                                   cache_dir=persistent_cache)
135     else:
136         print("----- using cache dataset -----")
137         train_ds = CacheDataset(
138             data=files[i]['train_files'], transform=train_transforms,
139             cache_rate=cfg.dic['train']['loader']['cache'], num_workers=cfg.dic['train']['loader']['num_workers'])
140         # train_ds = Dataset(data=train_files, transform=train_transforms)
141         # use batch size=2 to load images and use RandCropByPosNegLabeld
142         val_ds = CacheDataset(
143             data=files[i]['val_files'], transform=val_transforms,
144             cache_rate=cfg.dic['train']['loader']['cache'], num_workers=cfg.dic['train']['loader']['num_workers'])
145         out_channels = cfg.dic['model']['out_channels'] if cfg.dic['model'].get('out_channels',
146                                     None) else cfg.model.out_channels
147         cardio_vessel_segmentation_train(cfg=cfg,
148                                         model=cfg.model,
149                                         num_class=out_channels,
150                                         loss_function=cfg.train_loss,
151                                         val_metric=cfg.val_metric,
152                                         optimizer=cfg.optimizer_init,
153                                         lr_scheduler=cfg.lr_scheduler_init,
154                                         train_dataset=train_ds,
155                                         val_dataset=val_ds,
156                                         experiment_path=experiment_path_fold,
157                                         device=cfg.device,
158                                         metric_record=metric_record,
159                                         start_epoch=cfg.start_epoch,
160                                         mirror_axes=cfg.train_mirror_axes,
161                                         sw_batch_size=cfg.train_sw_batch_size,
162                                         overlap=cfg.train_sw_overlap,
163                                         )
164
165     elif cfg.dic['mode'] == 'test':
166         # ----- built transform sequence ----- #
167         train_transforms, val_transforms, save_transform = get_transform(cfg.dic['transform'])
168
169         cfg.creat_test_require()
170         cfg.save_config(os.path.join(cfg.test_output_path, 'config.yaml'))
171         if cfg.dic["test"]["loader"].get("file path"):
172             files = load_json(cfg.dic["test"]["loader"]["file_path"])["val_files"]
173         elif cfg.dic["test"]["loader"].get("val set"):
174             val_files = prepare_datalist_with_file(image_file=cfg.test_img_path,
175                                                    label_file=cfg.test_label_path,
176                                                    img_name=cfg.dic["test"]["loader"]["val_set"], )
177         files = val_files
178     else:
179         files = prepare_datalist(image_file=cfg.train_img_path,
180                                 label_file=cfg.test_label_path,
181                                 split_mode=cfg.dic['test']['loader']['split_mode'], )
182
183     total_ds = CacheDataset(
184         data=files, transform=val_transforms, cache_rate=cfg.dic['test']['loader']['cache'], num_workers=2)
185
186     cardio_vessel_segmentation_test(model=cfg.model,
187                                     val_dataset=total_ds,
188                                     device=cfg.device,
189                                     output_path=cfg.test_output_path,
190                                     window_size=cfg.dic['test']['test_windows_size'],
191                                     save_data=cfg.dic['test']['save_data'])
192
193     elif cfg.dic['mode'] == 'infer':
194         # ----- built transform sequence ----- #
195         infer_transforms = get_transform(cfg.dic['transform'], mode='infer')
196
197         cfg.creat_infer_require()
198         cfg.save_config(os.path.join(cfg.infer_output_path, 'config.yaml'))
199
200         # files = prepare_image_list(image_path=cfg.dic['infer']['loader']['path'])
201
202         image_path = pathlib.Path(cfg.infer_img_path)
203         assert image_path.is_dir(), f"img path not exist: {image_path}"
204         # get the image name list in image_path using pathlib
205         train_images = [path.name for path in image_path.glob("*.nii.gz")]
206         data_dicts = [
207             {"image": str(pathlib.Path(image_path, image_name))}
208             for image_name in train_images
209         ]
210         total_ds = CacheDataset(
211             data=data_dicts, transform=infer_transforms, cache_rate=cfg.dic['infer']['loader']['cache'], num_workers=2)
212         cardio_vessel_segmentation_infer(model=cfg.model,
213                                         val_dataset=total_ds,
214                                         device=cfg.device,
215                                         output_path=cfg.infer_output_path,
216                                         window_size=tuple(cfg.dic['transform']['patch_size']),
217                                         overlap=cfg.infer_sw_overlap,
218                                         origin_transforms=infer_transforms,
219                                         mirror_axes=cfg.infer_mirror_axes,
220                                         sw_batch_size=cfg.infer_sw_batch_size,
221                                         mode="gaussian",
222                                         )
223     else:
224         raise RuntimeError('Only train and infer mode are supported now')

```

H:\git\Repo2PDF\dist\repo2pdf\APP\SparrowLink\model\CS2net.py

```

0 SparrowLink/model/CS2net.py
1 """
2 3D Channel and Spatial Attention Network (CSA-Net 3D).
3 """
4 from future import division
5 import torch
6 import torch.nn as nn
7 import torch.nn.functional as F
8
9

```



```

10 def downsample():
11     return nn.MaxPool3d(kernel_size=2, stride=2)
12
13
14 def deconv(in_channels, out_channels):
15     return nn.ConvTranspose3d(in_channels, out_channels, kernel_size=2, stride=2)
16
17
18 def initialize_weights(*models):
19     for model in models:
20         for m in model.modules():
21             if isinstance(m, nn.Conv3d) or isinstance(m, nn.Linear):
22                 nn.init.kaiming_normal(m.weight)
23                 if m.bias is not None:
24                     m.bias.data.zero_()
25             elif isinstance(m, nn.BatchNorm3d):
26                 m.weight.data.fill_(1)
27                 m.bias.data.zero_()
28
29
30 class ResEncoder3d(nn.Module):
31     def __init__(self, in_channels, out_channels):
32         super(ResEncoder3d, self).__init__()
33         self.conv1 = nn.Conv3d(in_channels, out_channels, kernel_size=3, padding=1)
34         self.bn1 = nn.BatchNorm3d(out_channels)
35         self.conv2 = nn.Conv3d(out_channels, out_channels, kernel_size=3, padding=1)
36         self.bn2 = nn.BatchNorm3d(out_channels)
37         self.relu = nn.ReLU(inplace=False)
38         self.conv1x1 = nn.Conv3d(in_channels, out_channels, kernel_size=1)
39
40     def forward(self, x):
41         residual = self.conv1x1(x)
42         out = self.relu(self.bn1(self.conv1(x)))
43         out = self.relu(self.bn2(self.conv2(out)))
44         out = residual + out
45         out = self.relu(out)
46         return out
47
48
49 class Decoder3d(nn.Module):
50     def __init__(self, in_channels, out_channels):
51         super(Decoder3d, self).__init__()
52         self.conv = nn.Sequential(
53             nn.Conv3d(in_channels, out_channels, kernel_size=3, padding=1),
54             nn.BatchNorm3d(out_channels),
55             nn.ReLU(inplace=False),
56             nn.Conv3d(out_channels, out_channels, kernel_size=3, padding=1),
57             nn.BatchNorm3d(out_channels),
58             nn.ReLU(inplace=False)
59         )
60
61     def forward(self, x):
62         out = self.conv(x)
63         return out
64
65
66 class SpatialAttentionBlock3d(nn.Module):
67     def __init__(self, in_channels):
68         super(SpatialAttentionBlock3d, self).__init__()
69         self.query = nn.Conv3d(in_channels, in_channels // 8, kernel_size=(1, 3, 1), padding=(0, 1, 0))
70         self.key = nn.Conv3d(in_channels, in_channels // 8, kernel_size=(3, 1, 1), padding=(1, 0, 0))
71         self.judge = nn.Conv3d(in_channels, in_channels // 8, kernel_size=(1, 1, 3), padding=(0, 0, 1))
72         self.value = nn.Conv3d(in_channels, in_channels, kernel_size=1)
73         self.gamma = nn.Parameter(torch.zeros(1))
74         self.softmax = nn.Softmax(dim=-1)
75
76     def forward(self, x):
77         """
78         :param x: input( BxCxHxWxD )
79         :return: affinity value + x
80         B: batch size
81         C: channels
82         H: height
83         W: width
84         D: slice number (depth)
85         """
86         B, C, H, W, D = x.size()
87         # compress x: [B,C,H,W,D]-->[B,H*W*D,C], make a matrix transpose
88         proj_query = self.query(x).view(B, -1, W * H * D).permute(0, 2, 1) # -> [B,W*H*D,C]
89         proj_key = self.key(x).view(B, -1, W * H * D) # -> [B,H*W*D,C]
90         proj_judge = self.judge(x).view(B, -1, W * H * D).permute(0, 2, 1) # -> [B,C,H*W*D]
91
92         affinity1 = torch.matmul(proj_query, proj_key)
93         affinity2 = torch.matmul(proj_judge, proj_key)
94         affinity = torch.matmul(affinity1, affinity2)
95         affinity = self.softmax(affinity)
96
97         proj_value = self.value(x).view(B, -1, H * W * D) # -> C*N
98         weights = torch.matmul(proj_value, affinity)
99         weights = weights.view(B, C, H, W, D)
100         out = self.gamma * weights + x
101         return out
102
103
104 class ChannelAttentionBlock3d(nn.Module):
105     def __init__(self, in_channels):
106         super(ChannelAttentionBlock3d, self).__init__()
107         self.gamma = nn.Parameter(torch.zeros(1))
108         self.softmax = nn.Softmax(dim=-1)
109
110     def forward(self, x):
111         """
112         :param x: input( BxCxHxWxD )
113         :return: affinity value + x
114         """
115         B, C, H, W, D = x.size()
116         proj_query = x.view(B, C, -1).permute(0, 2, 1)
117         proj_key = x.view(B, C, -1)
118         proj_judge = x.view(B, C, -1).permute(0, 2, 1)
119         affinity1 = torch.matmul(proj_key, proj_query)
120         affinity2 = torch.matmul(proj_key, proj_judge)
121         affinity = torch.matmul(affinity1, affinity2)
122         affinity_new = torch.max(affinity, -1, keepdim=True)[0].expand_as(affinity) - affinity
123         affinity_new = self.softmax(affinity_new)
124         proj_value = x.view(B, C, -1)
125         weights = torch.matmul(affinity_new, proj_value)
126         weights = weights.view(B, C, H, W, D)
127         out = self.gamma * weights + x
128         return out

```

```

129
130
131 class AffinityAttention3d(nn.Module):
132     """ Affinity attention module """
133
134     def __init__(self, in_channels):
135         super(AffinityAttention3d, self).__init__()
136         self.sab = SpatialAttentionBlock3d(in_channels)
137         self.cab = ChannelAttentionBlock3d(in_channels)
138         # self.conv1x1 = nn.Conv2d(in_channels * 2, in_channels, kernel_size=1)
139
140     def forward(self, x):
141         """
142         sab: spatial attention block
143         cab: channel attention block
144         :param x: input tensor
145         :return: sab + cab
146         """
147         sab = self.sab(x)
148         cab = self.cab(x)
149         out = sab + cab + x
150         return out
151
152
153 class CSNet3D(nn.Module):
154     def __init__(self, out_channels, in_channels):
155         """
156         :param classes: the object classes number.
157         :param channels: the channels of the input image.
158         """
159         super(CSNet3D, self).__init__()
160         self.out_channels = out_channels
161         self.enc_input = ResEncoder3d(in_channels, 16)
162         self.encoder1 = ResEncoder3d(16, 32)
163         self.encoder2 = ResEncoder3d(32, 64)
164         self.encoder3 = ResEncoder3d(64, 128)
165         self.encoder4 = ResEncoder3d(128, 256)
166         self.downsample = downsample()
167         self.affinity_attention = AffinityAttention3d(256)
168         self.attention_fuse = nn.Conv3d(256 * 2, 256, kernel_size=1)
169         self.decoder4 = Decoder3d(256, 128)
170         self.decoder3 = Decoder3d(128, 64)
171         self.decoder2 = Decoder3d(64, 32)
172         self.decoder1 = Decoder3d(32, 16)
173         self.deconv4 = deconv(256, 128)
174         self.deconv3 = deconv(128, 64)
175         self.deconv2 = deconv(64, 32)
176         self.deconv1 = deconv(32, 16)
177         self.final = nn.Conv3d(16, out_channels, kernel_size=1)
178         initialize_weights(self)
179
180     def forward(self, x):
181         enc_input = self.enc_input(x)
182         down1 = self.downsample(enc_input)
183
184         enc1 = self.encoder1(down1)
185         down2 = self.downsample(enc1)
186
187         enc2 = self.encoder2(down2)
188         down3 = self.downsample(enc2)
189
190         enc3 = self.encoder3(down3)
191         down4 = self.downsample(enc3)
192
193         input_feature = self.encoder4(down4)
194
195         # Do Attention operations here
196         attention = self.affinity_attention(input_feature)
197         attention_fuse = input_feature + attention
198
199         # Do decoder operations here
200         up4 = self.deconv4(attention_fuse)
201         up4 = torch.cat((enc3, up4), dim=1)
202         dec4 = self.decoder4(up4)
203
204         up3 = self.deconv3(dec4)
205         up3 = torch.cat((enc2, up3), dim=1)
206         dec3 = self.decoder3(up3)
207
208         up2 = self.deconv2(dec3)
209         up2 = torch.cat((enc1, up2), dim=1)
210         dec2 = self.decoder2(up2)
211
212         up1 = self.deconv1(dec2)
213         up1 = torch.cat((enc_input, up1), dim=1)
214         dec1 = self.decoder1(up1)
215
216         final = self.final(dec1)
217         final = F.sigmoid(final)
218         return final
219
220
221 if __name__ == "__main__":
222     from torch.distributions.normal import Normal
223     model1 = CSNet3D(2, 1)
224     model2 = CSNet3D(2, 4)
225
226
227     # from torch.distributions.normal import Normal
228     # d1 = model1.state_dict()
229     # d2 = model2.state_dict()
230     # for k2, v2 in d2.items():
231     #     if d1[k2].shape != v2.shape:
232     #         assert d1[k2].shape[1] != v2.shape[1] and len(d1[k2].shape) > 1 and len(v2.shape) > 1
233     #         noise_shape = torch.tensor(v2.shape)
234     #         noise_shape[1] = v2.shape[1] - d1[k2].shape[1]
235     #         noise = nn.Parameter(Normal(0, 1e-5).sample(noise_shape)).to(d1[k2].device)
236     #         d1[k2] = torch.cat((d1[k2], noise), dim=1)
237     #         print(f"key: {k2}, shape1: {d1[k2].shape}, mean1: {d1[k2].mean()}, shape2: {v2.shape}, mean2: {v2.mean()}")
238
239     d1 = torch.load('H:/Graduate_project/segment_server/experiments/Graduate_project/two_stage2/first_stage/CS2net/checkpoint/best_metric_model.pth')
240     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
241     model1.to(device)
242     model1.load_state_dict(d1)
243     d2 = model2.state_dict()
244     print("-----load checkpoint and modify-----")
245     for k2, v2 in d2.items():
246         if d1[k2].shape != v2.shape:
247             assert d1[k2].shape[1] != v2.shape[1] and len(d1[k2].shape) > 1 and len(v2.shape) > 1
248             noise_shape = torch.tensor(v2.shape)

```

```

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

```

```

0 SparrowLink/model/denseunet_skip.py
1 import warnings
2 from typing import Optional, Sequence, Tuple, Union
3
4 import torch
5 import torch.nn as nn
6 import torch
7 import torch.nn as nn
8 import torch.nn.functional as F
9
10
11 from monai.networks.blocks.convolutions import Convolution, ResidualUnit
12 from monai.networks.layers.factories import Act, Norm
13 from monai.networks.layers.simplelayers import SkipConnection
14 from monai.utils import alias, deprecated_arg, export
15 from monai.networks.layers.convutils import same_padding
16
17 @export("monai.networks.nets")
18 @alias("UNet")
19 class SkipDenseUNet(nn.Module):
20
21     @deprecated_arg(
22         name="dimensions", new_name="spatial_dims", since="0.6", msg_suffix="Please use `spatial_dims` instead."
23     )
24     def __init__(
25         self,
26         spatial_dims: int,
27         in_channels: int,
28         out_channels: int,
29         channels: Sequence[int],
30         strides: Sequence[int],
31         kernel_size: Union[Sequence[int], int] = 3,
32         up_kernel_size: Union[Sequence[int], int] = 3,
33         num_res_units: int = 0,
34         act: Union[Tuple, str] = Act.PRELU,
35         norm: Union[Tuple, str] = Norm.INSTANCE,
36         dropout: float = 0.0,
37         bias: bool = True,
38         adn_ordering: str = "NDA",
39         dimensions: Optional[int] = None,
40     ) -> None:
41
42         super().__init__()
43
44         if len(channels) < 2:
45             raise ValueError("the length of `channels` should be no less than 2.")
46         delta = len(strides) - (len(channels) - 1)
47         if delta < 0:
48             raise ValueError("the length of `strides` should equal to `len(channels) - 1`.")
49         if delta > 0:
50             warnings.warn(f"`len(strides) > len(channels) - 1`, the last {delta} values of strides will not be used.")
51         if dimensions is not None:
52             spatial_dims = dimensions
53         if isinstance(kernel_size, Sequence):
54             if len(kernel_size) != spatial_dims:
55                 raise ValueError("the length of `kernel_size` should equal to `dimensions`.")
56         if isinstance(up_kernel_size, Sequence):
57             if len(up_kernel_size) != spatial_dims:
58                 raise ValueError("the length of `up_kernel_size` should equal to `dimensions`.")
59
60         self.dimensions = spatial_dims
61         self.in_channels = in_channels
62         self.out_channels = out_channels
63         self.channels = channels
64         self.strides = strides
65         self.kernel_size = kernel_size
66         self.up_kernel_size = up_kernel_size
67         self.num_res_units = num_res_units
68         self.act = act
69         self.norm = norm
70         self.dropout = dropout
71         self.bias = bias
72         self.adn_ordering = adn_ordering
73
74     def _create_block(
75         inc: int, outc: int, channels: Sequence[int], strides: Sequence[int], is_top: bool
76     ) -> nn.Module:
77         """
78         Builds the UNet structure from the bottom up by recursing down to the bottom block, then creating sequential
79         blocks containing the downsample path, a skip connection around the previous block, and the upsample path.
80
81         Args:
82             inc: number of input channels.
83
84             outc: number of output channels.
85             channels: sequence of channels. Top block first.
86             strides: convolution stride.
87             is_top: True if this is the top block.
88         """
89         c = channels[0]
90         s = strides[0]
91
92         subblock: nn.Module
93
94         if len(channels) > 2:
95             subblock = _create_block(c, c, channels[1:], strides[1:], False) # continue recursion down
96             upc = c * 2
97         else:
98             # the next layer is the bottom so stop recursion, create the bottom layer as the subblock for this layer
99             subblock = self._get_bottom_layer(c, channels[1])
100             upc = c + channels[1]
101
102         down = self._get_down_layer(inc, c, s, is_top) # create layer in downsampling path
103         up = self._get_up_layer(upc, outc, s, is_top) # create layer in upsampling path
104
105         return self._get_connection_block(down, up, subblock)
106
107     self.model = _create_block(in_channels, out_channels, self.channels, self.strides, True)
108
109 def _get_connection_block(self, down_path: nn.Module, up_path: nn.Module, subblock: nn.Module) -> nn.Module:
110     """
111     Returns the block object defining a layer of the UNet structure including the implementation of the skip
112     between encoding (down) and decoding (up) sides of the network.
113
114     Args:
115         down_path: encoding half of the layer
116         up_path: decoding half of the layer
117         subblock: block defining the next layer in the network.
118     Returns: block for this layer: `nn.Sequential(down_path, SkipConnection(subblock), up_path)`
119     """
120     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         )

```

```

242         (residual): Identity()
243     )
244
245     """
246
247     def __init__(
248         self,
249         in_channels: int,
250         out_channels: int,
251         strides: Union[Sequence[int], int] = 1,
252         kernel_size: Union[Sequence[int], int] = 3,
253
254         subunits: int = 2,
255         act_ordering: str = "NDA",
256         act: Optional[Union[Tuple, str]] = "PRELU",
257         norm: Optional[Union[Tuple, str]] = "INSTANCE",
258         dropout: Optional[Union[Tuple, str, float]] = None,
259         dropout_dim: Optional[int] = 1,
260         dilation: Union[Sequence[int], int] = 1,
261         bias: bool = True,
262         last_conv_only: bool = False,
263         padding: Optional[Union[Sequence[int], int]] = None,
264     ) -> None:
265         super().__init__()
266         self.in_channels = in_channels
267         self.out_channels = out_channels
268         self.strides = strides
269         self.dense_blocks = DenseBlock(in_channels=self.in_channels,
270                                       out_channels=self.out_channels,
271                                       nb_layers=subunits)
272         self.transition = TransitionLayer(in_channels=self.out_channels,
273                                         out_channels=self.out_channels,
274                                         stride=self.strides)
275
276     def forward(self, x: torch.Tensor) -> torch.Tensor:
277         x = self.dense_blocks(x)
278         x = self.transition(x)
279         return x
280
281 class BottleneckLayer(nn.Module):
282     def __init__(self, in_channels, out_channels):
283         super(BottleneckLayer, self).__init__()
284         self.conv1 = nn.Conv3d(in_channels, out_channels, kernel_size=1, stride=1, padding=0)
285         self.bn1 = nn.BatchNorm3d(out_channels)
286         self.relu1 = nn.ReLU(inplace=True)
287         self.conv2 = nn.Conv3d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)
288         self.bn2 = nn.BatchNorm3d(out_channels)
289         self.relu2 = nn.ReLU(inplace=True)
290
291     def forward(self, x):
292         out = self.conv1(x)
293         out = self.bn1(out)
294         out = self.relu1(out)
295         out = self.conv2(out)
296         out = self.bn2(out)
297         out = self.relu2(out)
298         return out
299
300 class TransitionLayer(nn.Module):
301     def __init__(self, in_channels, out_channels, stride=2):
302         super(TransitionLayer, self).__init__()
303         self.conv = nn.Conv3d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1)
304         self.bn = nn.BatchNorm3d(out_channels)
305         self.relu = nn.ReLU(inplace=True)
306
307     def forward(self, x):
308         out = self.conv(x)
309         out = self.bn(out)
310         out = self.relu(out)
311         return out
312
313 class DenseBlock(nn.Module):
314     def __init__(self, in_channels, out_channels, nb_layers=4):
315         super(DenseBlock, self).__init__()
316         self.layers = nn.ModuleList()
317         for i in range(nb_layers):
318             self.layers.append(BottleneckLayer(in_channels + i * out_channels, out_channels))
319         self.merge = nn.Sequential(
320             nn.Conv3d(in_channels + nb_layers * out_channels, out_channels, kernel_size=1, stride=1),
321             nn.BatchNorm3d(out_channels),
322             nn.ReLU(inplace=True),
323         )
324
325     def forward(self, x):
326         layers_concat = [x]
327         for layer in self.layers:
328             out = layer(Concatenation(layers_concat))
329             layers_concat.append(out)
330         return self.merge(Concatenation(layers_concat))
331
332 def Concatenation(layers):
333     return torch.cat(layers, dim=1)
334
335 if __name__ == "__main__":
336     import torch
337
338     model1 = SkipDenseUNet(
339         spatial_dims=3,
340         in_channels=1,
341         out_channels=2,
342         channels=(16, 32, 64, 128, 256),
343         strides=(2, 2, 2, 2),
344         num_res_units=2,
345         norm=Norm.BATCH,
346     )
347
348     model2 = SkipDenseUNet(
349         spatial_dims=3,
350         in_channels=4,
351         out_channels=2,
352         channels=(16, 32, 64, 128, 256),
353         strides=(2, 2, 2, 2),
354         num_res_units=2,
355         norm=Norm.BATCH,
356     )
357
358     from torch.distributions.normal import Normal
359     dl = torch.load('H:/Graduate_project/segment_server/experiments/Graduate_project/two_stage2/first_stage/DenseUnet/checkpoint/best_metric_model.pth')
360     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
361     model1.to(device)

```

```

364 model1.load_state_dict(d1)
365 d2 = model2.state_dict()
366 in_channels = 4
367 print("-----load checkpoint and modify-----")
368 for k2, v2 in d2.items():
369     if d1[k2].shape != v2.shape:
370         assert d1[k2].shape[1] != v2.shape[1] and len(d1[k2].shape) > 1 and len(v2.shape) > 1
371         noise_shape = torch.tensor(v2.shape)
372         noise_shape[1] = v2.shape[1] - d1[k2].shape[1]
373         noise = nn.Parameter(Normal(0, 1e-10).sample(noise_shape)).to(d1[k2].device)
374         print(f"key: {k2}, shape1: {d1[k2].shape}, mean1: {d1[k2].mean()}, shape2: {v2.shape}, mean2: {v2.mean()}")
375         d1[k2] = torch.cat((d1[k2][:, :1, ...], noise, d1[k2][:, 1:, ...]), dim=1)
376 model2.to(device)
377 model2.load_state_dict(d1)
378 x1 = torch.ones(1, 1, 64, 64, 64).to(device)
379 x2 = torch.ones(1, 4, 64, 64, 64).to(device)
380 y1 = model1(x1)
381 y2 = model2(x2)
382 print(((y1 - y2)**2).max(), ((y1 - y2)**2).mean())
383 pass

```

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

```

0 SparrowLink/model/swin_unet.py
1 from functools import Reduce, lru_cache
2 from operator import mul
3
4 import numpy as np
5 import torch
6 import torch.nn as nn
7 import torch.nn.functional as F
8 import torch.utils.checkpoint as checkpoint
9 from einops import rearrange
10 # from mmcv.runner import load_checkpoint
11 from timm.models.layers import DropPath, trunc_normal_
12 from IPython import embed
13
14 class Mlp(nn.Module):
15     """ Multilayer perceptron """
16
17     def __init__(self, in_features, hidden_features=None, out_features=None, act_layer=nn.GELU, drop=0.):
18         super().__init__()
19         out_features = out_features or in_features
20         hidden_features = hidden_features or in_features
21         self.fc1 = nn.Linear(in_features, hidden_features)
22         self.act = act_layer()
23         self.fc2 = nn.Linear(hidden_features, out_features)
24         self.drop = nn.Dropout(drop)
25
26     def forward(self, x):
27         x = self.fc1(x)
28         x = self.act(x)
29         x = self.drop(x)
30         x = self.fc2(x)
31         x = self.drop(x)
32         return x
33
34
35 def img2windows(img, H_sp, W_sp):
36     """
37     img: B C D H W
38     """
39     B, C, D, H, W = img.shape
40     img_reshape = img.view(B, C, D, H // H_sp, H_sp, W // W_sp, W_sp)
41     img_perm = img_reshape.permute(0, 2, 3, 5, 4, 6, 1).contiguous().reshape(-1, D * H_sp * W_sp, C)
42     return img_perm
43
44
45 def windows2img(img_splits_hw, H_sp, W_sp, D, H, W):
46     """
47     img_splits_hw: B' D H W C
48     """
49     B = int(img_splits_hw.shape[0] / (D * H * W / H_sp / W_sp))
50
51     img = img_splits_hw.view(B, D, H // H_sp, W // W_sp, H_sp, W_sp, -1)
52     img = img.permute(0, 1, 2, 4, 3, 5, 6).contiguous().view(B, D, H, W, -1)
53
54     return img
55
56
57 class Merge_Block(nn.Module):
58     def __init__(self, dim, dim_out, norm_layer=nn.LayerNorm):
59         super().__init__()
60         self.conv = nn.Conv3d(dim, dim_out, 3, 2, 1)
61         self.norm = norm_layer(dim_out)
62
63     def forward(self, x):
64         B, new_HW, C = x.shape
65         D = 32
66         H = W = int(np.sqrt(new_HW // D))
67         x = x.transpose(-2, -1).contiguous().view(B, C, D, H, W)
68         x = self.conv(x)
69         B, C = x.shape[:2]
70         x = x.view(B, C, -1).transpose(-2, -1).contiguous()
71         x = self.norm(x)
72
73         return x
74
75
76 def window_partition(x, window_size):
77     """
78     Args:
79         x: (B, D, H, W, C)
80         window_size (tuple[int]): window size
81
82     Returns:
83         windows: (B*num_windows, window_size*window_size, C)
84     """
85     B, D, H, W, C = x.shape
86     x = x.view(B, D // window_size[0], window_size[0], H // window_size[1], window_size[1], W // window_size[2],
87                window_size[2], C)
88     windows = x.permute(0, 1, 3, 5, 2, 4, 6, 7).contiguous().view(-1, reduce(mul, window_size), C)
89     return windows
90
91
92 def window_reverse(windows, window_size, B, D, H, W):
93     """
94     Args:
95         windows: (B*num_windows, window_size, window_size, C)
96         window_size (tuple[int]): Window size
97         H (int): Height of image
98         W (int): Width of image
99     """

```

```

99 Returns:
100 x: (B, D, H, W, C)
101 """
102 x = windows.view(B, D // window_size[0], H // window_size[1], W // window_size[2], window_size[0], window_size[1],
103                 window_size[2], -1)
104 x = x.permute(0, 1, 4, 2, 5, 3, 6, 7).contiguous().view(B, D, H, W, -1)
105 return x
106
107
108 def get_window_size(x_size, window_size, shift_size=None):
109     use_window_size = list(window_size)
110     if shift_size is not None:
111         use_shift_size = list(shift_size)
112         for i in range(len(x_size)):
113             if x_size[i] <= window_size[i]:
114                 use_window_size[i] = x_size[i]
115                 if shift_size is not None:
116                     use_shift_size[i] = 0
117
118     if shift_size is None:
119         return tuple(use_window_size)
120
121     else:
122         return tuple(use_window_size), tuple(use_shift_size)
123
124
125 class WindowAttention3D(nn.Module):
126     """ Window based multi-head self attention (W-MSA) module with relative position bias.
127     It supports both of shifted and non-shifted window.
128     Args:
129         dim (int): Number of input channels.
130         window_size (tuple[int]): The temporal length, height and width of the window.
131         num_heads (int): Number of attention heads.
132         qkv_bias (bool, optional): If True, add a learnable bias to query, key, value. Default: True
133         qk_scale (float | None, optional): Override default qk scale of head_dim ** -0.5 if set
134         attn_drop (float, optional): Dropout ratio of attention weight. Default: 0.0
135         proj_drop (float, optional): Dropout ratio of output. Default: 0.0
136     """
137
138     def __init__(self, dim, window_size, num_heads, qkv_bias=False, qk_scale=None, attn_drop=0., proj_drop=0.):
139         super().__init__()
140         self.dim = dim
141         self.window_size = window_size  # Wd, Wh, Ww
142         self.num_heads = num_heads
143         head_dim = dim // num_heads
144         self.scale = qk_scale or head_dim ** -0.5
145
146         # define a parameter table of relative position bias
147         self.relative_position_bias_table = nn.Parameter(
148             torch.zeros((2 * window_size[0] - 1) * (2 * window_size[1] - 1) * (2 * window_size[2] - 1),
149                         num_heads))  # 2*Wd-1 * 2*Wh-1 * 2*Ww-1, nH
150
151         # get pair-wise relative position index for each token inside the window
152         coords_d = torch.arange(self.window_size[0])
153         coords_h = torch.arange(self.window_size[1])
154         coords_w = torch.arange(self.window_size[2])
155         coords = torch.stack(torch.meshgrid(coords_d, coords_h, coords_w))  # 3, Wd, Wh, Ww
156         coords_flatten = torch.flatten(coords, 1)  # 3, Wd*Wh*Ww
157         relative_coords = coords_flatten[:, :, None] - coords_flatten[:, None, :]  # 3, Wd*Wh*Ww, Wd*Wh*Ww
158         relative_coords = relative_coords.permute(1, 2, 0).contiguous()  # Wd*Wh*Ww, Wd*Wh*Ww, 3
159         relative_coords[:, :, 0] += self.window_size[0] - 1  # shift to start from 0
160         relative_coords[:, :, 1] += self.window_size[1] - 1
161         relative_coords[:, :, 2] += self.window_size[2] - 1
162
163         relative_coords[:, :, 0] *= (2 * self.window_size[1] - 1) * (2 * self.window_size[2] - 1)
164         relative_coords[:, :, 1] *= (2 * self.window_size[2] - 1)
165         relative_position_index = relative_coords.sum(-1)  # Wd*Wh*Ww, Wd*Wh*Ww
166         self.register_buffer("relative_position_index", relative_position_index)
167
168         self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
169         self.attn_drop = nn.Dropout(attn_drop)
170         self.proj = nn.Linear(dim, dim)
171         self.proj_drop = nn.Dropout(proj_drop)
172
173         trunc_normal_(self.relative_position_bias_table, std=0.02)
174         self.softmax = nn.Softmax(dim=-1)
175
176     def forward(self, x, mask=None, prev_v=None, prev_k=None, prev_q=None, is_decoder=False):
177         """ Forward function.
178         Args:
179             x: input features with shape of (num_windows*B, N, C)
180             mask: (0/-inf) mask with shape of (num_windows, N, N) or None
181         """
182         B, N, C = x.shape
183         qkv = self.qkv(x).reshape(B, N, 3, self.num_heads, C // self.num_heads).permute(2, 0, 3, 1, 4)
184         q, k, v = qkv[0], qkv[1], qkv[2]  # B, nH, N, C
185
186         q = q * self.scale
187         attn = q @ k.transpose(-2, -1)
188
189         relative_position_bias = self.relative_position_bias_table[
190             self.relative_position_index[:N, :N].reshape(-1)].reshape(
191                 N, N, -1)  # Wd*Wh*Ww, Wd*Wh*Ww, nH
192         relative_position_bias = relative_position_bias.permute(2, 0, 1).contiguous()  # nH, Wd*Wh*Ww, Wd*Wh*Ww
193         attn = attn + relative_position_bias.unsqueeze(0)  # B, nH, N, N
194
195         if mask is not None:
196             nW = mask.shape[0]
197             attn = attn.view(B // nW, nW, self.num_heads, N, N) + mask.unsqueeze(1).unsqueeze(0)
198             attn = attn.view(-1, self.num_heads, N, N)
199             attn = self.softmax(attn)
200         else:
201             attn = self.softmax(attn)
202
203         attn = self.attn_drop(attn)
204
205         x = (attn @ v).transpose(1, 2).reshape(B, N, C)
206         x = self.proj(x)
207         x = self.proj_drop(x)
208         x2 = None
209
210         if is_decoder:
211             q = q * self.scale
212             attn2 = q @ prev_k.transpose(-2, -1)
213             attn2 = attn2 + relative_position_bias.unsqueeze(0)
214
215             if mask is not None:
216                 nW = mask.shape[0]
217                 attn2 = attn2.view(B // nW, nW, self.num_heads, N, N) + mask.unsqueeze(1).unsqueeze(0)
218                 attn2 = attn2.view(-1, self.num_heads, N, N)
219                 attn2 = self.softmax(attn2)

```



```

221         else:
222             attn2 = self.softmax(attn2)
223
224             attn2 = self.attn_drop(attn2)
225
226             x2 = (attn2 @ prev_v).transpose(1, 2).reshape(B_, N, C)
227             x2 = self.proj(x2)
228             x2 = self.proj_drop(x2)
229
230         return x, x2, v, k, q
231
232
233 class PositionalEncoding3D(nn.Module):
234     def __init__(self, channels):
235         """
236         :param channels: The last dimension of the tensor you want to apply pos emb to.
237         """
238         super(PositionalEncoding3D, self).__init__()
239         channels = int(np.ceil(channels / 6) * 2)
240         if channels % 2:
241             channels += 1
242         self.channels = channels
243         inv_freq = 1. / (10000 ** (torch.arange(0, channels, 2).float() / channels))
244         self.register_buffer('inv_freq', inv_freq)
245
246     def forward(self, tensor):
247         """
248         :param tensor: A 5d tensor of size (batch_size, x, y, z, ch)
249         :return: Positional Encoding Matrix of size (batch_size, x, y, z, ch)
250         """
251         if len(tensor.shape) != 5:
252             raise RuntimeError("The input tensor has to be 5d!")
253         batch_size, x, y, z, orig_ch = tensor.shape
254         pos_x = torch.arange(x, device=tensor.device).type(self.inv_freq.type())
255         pos_y = torch.arange(y, device=tensor.device).type(self.inv_freq.type())
256         pos_z = torch.arange(z, device=tensor.device).type(self.inv_freq.type())
257         sin_inp_x = torch.einsum("i,j->ij", pos_x, self.inv_freq)
258         sin_inp_y = torch.einsum("i,j->ij", pos_y, self.inv_freq)
259         sin_inp_z = torch.einsum("i,j->ij", pos_z, self.inv_freq)
260         emb_x = torch.cat((sin_inp_x.sin(), sin_inp_x.cos()), dim=-1).unsqueeze(1)
261         emb_y = torch.cat((sin_inp_y.sin(), sin_inp_y.cos()), dim=-1).unsqueeze(1)
262         emb_z = torch.cat((sin_inp_z.sin(), sin_inp_z.cos()), dim=-1)
263         emb = torch.zeros((x, y, z, self.channels * 3), device=tensor.device).type(tensor.type())
264         emb[:, :, :, :self.channels] = emb_x
265         emb[:, :, :, self.channels:2 * self.channels] = emb_y
266         emb[:, :, :, 2 * self.channels:] = emb_z
267
268         return emb[None, :, :, :, :orig_ch].repeat(batch_size, 1, 1, 1, 1)
269
270
271 class SwinTransformerBlock3D(nn.Module):
272     """ Swin Transformer Block.
273
274     Args:
275         dim (int): Number of input channels.
276         num_heads (int): Number of attention heads.
277         window_size (tuple[int]): Window size.
278         shift_size (tuple[int]): Shift size for SW-MSA.
279         mlp_ratio (float): Ratio of mlp hidden dim to embedding dim.
280         qkv_bias (bool, optional): If True, add a learnable bias to query, key, value. Default: True
281         qk_scale (float | None, optional): Override default qk scale of head_dim ** -0.5 if set.
282         drop (float, optional): Dropout rate. Default: 0.0
283         attn_drop (float, optional): Attention dropout rate. Default: 0.0
284         drop_path (float, optional): Stochastic depth rate. Default: 0.0
285         act_layer (nn.Module, optional): Activation layer. Default: nn.GELU
286         norm_layer (nn.Module, optional): Normalization layer. Default: nn.LayerNorm
287     """
288
289     def __init__(self, dim, num_heads, window_size=(7, 7, 7), shift_size=(0, 0, 0),
290                 mlp_ratio=4., qkv_bias=True, qk_scale=None, drop=0., attn_drop=0., drop_path=0.,
291                 act_layer=nn.GELU, norm_layer=nn.LayerNorm, use_checkpoint=False):
292         super().__init__()
293         self.dim = dim
294         self.num_heads = num_heads
295         self.window_size = window_size
296         self.shift_size = shift_size
297         self.mlp_ratio = mlp_ratio
298         self.use_checkpoint = use_checkpoint
299
300         assert 0 <= self.shift_size[0] < self.window_size[0], "shift_size must in 0-window_size"
301         assert 0 <= self.shift_size[1] < self.window_size[1], "shift_size must in 0-window_size"
302         assert 0 <= self.shift_size[2] < self.window_size[2], "shift_size must in 0-window_size"
303
304         self.norm1 = norm_layer(dim)
305         self.attn = WindowAttention3D(
306             dim, window_size=self.window_size, num_heads=num_heads,
307             qkv_bias=qkv_bias, qk_scale=qk_scale, attn_drop=attn_drop, proj_drop=drop)
308
309         self.drop_path = DropPath(drop_path) if drop_path > 0. else nn.Identity()
310         self.norm2 = norm_layer(dim)
311         mlp_hidden_dim = int(dim * mlp_ratio)
312         self.mlp = Mlp(in_features=dim, hidden_features=mlp_hidden_dim, act_layer=act_layer, drop=drop)
313
314     def forward_part1(self, x, mask_matrix, prev_v, prev_k, prev_q, is_decoder):
315         B, D, H, W, C = x.shape
316         window_size, shift_size = get_window_size((D, H, W), self.window_size, self.shift_size)
317
318         x = self.norm1(x)
319         # pad feature maps to multiples of window size
320         pad_l = pad_t = pad_d0 = 0
321         pad_d1 = (window_size[0] - D % window_size[0]) % window_size[0]
322         pad_b = (window_size[1] - H % window_size[1]) % window_size[1]
323         pad_r = (window_size[2] - W % window_size[2]) % window_size[2]
324         x = F.pad(x, (0, 0, pad_l, pad_r, pad_t, pad_b, pad_d0, pad_d1))
325         _, Dp, Hp, Wp, _ = x.shape
326         # cyclic shift
327         if any(i > 0 for i in shift_size):
328             shifted_x = torch.roll(x, shifts=(-shift_size[0], -shift_size[1], -shift_size[2]), dims=(1, 2, 3))
329             attn_mask = mask_matrix
330         else:
331             shifted_x = x
332             attn_mask = None
333         # partition windows
334         x_windows = window_partition(shifted_x, window_size) # B*nW, Wd*Wh*Ww, C
335         # W-MSA/SW-MSA
336         attn_windows, cross_attn_windows, v, k, q = self.attn(x_windows, mask=attn_mask, prev_v=prev_v, prev_k=prev_k,
337                                                                prev_q=prev_q, is_decoder=is_decoder) # B*nW, Wd*Wh*Ww, C
338
339         # merge windows
340         attn_windows = attn_windows.view(-1, *(window_size + (C,)))
341         shifted_x = window_reverse(attn_windows, window_size, B, Dp, Hp, Wp) # B D' H' W' C
342         # reverse cyclic shift

```

```

343     if any(i > 0 for i in shift_size):
344         x = torch.roll(shifted_x, shifts=(shift_size[0], shift_size[1], shift_size[2]), dims=(1, 2, 3))
345     else:
346         x = shifted_x
347
348     x2 = None
349     if pad_d1 > 0 or pad_r > 0 or pad_b > 0:
350         x = x[:, :D, :H, :W, :].contiguous()
351
352     if cross_attn_windows is not None:
353         # merge windows
354         cross_attn_windows = cross_attn_windows.view(-1, *(window_size + (C,)))
355         cross_shifted_x = window_reverse(cross_attn_windows, window_size, B, Dp, Hp, Wp) # B D' H' W' C
356         # reverse cyclic shift
357         if any(i > 0 for i in shift_size):
358             x2 = torch.roll(cross_shifted_x, shifts=(shift_size[0], shift_size[1], shift_size[2]), dims=(1, 2, 3))
359         else:
360             x2 = cross_shifted_x
361
362         if pad_d1 > 0 or pad_r > 0 or pad_b > 0:
363             x2 = x2[:, :D, :H, :W, :].contiguous()
364
365     return x, x2, v, k, q
366
367 def forward_part2(self, x):
368     return self.drop_path(self.mlp(self.norm2(x)))
369
370 def forward_part3(self, x):
371     return self.mlp(self.norm2(x))
372
373 def forward(self, x, mask_matrix, prev_v, prev_k, prev_q, is_decoder=False):
374     """ Forward function.
375
376     Args:
377         x: Input feature, tensor size (B, D, H, W, C).
378         mask_matrix: Attention mask for cyclic shift.
379     """
380
381     alpha = 0.5
382     shortcut = x
383     x2, v, k, q = None, None, None, None
384
385     if self.use_checkpoint:
386         x = checkpoint.checkpoint(self.forward_part1, x, mask_matrix)
387     else:
388         x, x2, v, k, q = self.forward_part1(x, mask_matrix, prev_v, prev_k, prev_q, is_decoder)
389
390     x = shortcut + self.drop_path(x)
391
392     if self.use_checkpoint:
393         x = x + checkpoint.checkpoint(self.forward_part2, x)
394     else:
395         x = x + self.forward_part2(x)
396
397     if x2 is not None:
398         x2 = shortcut + self.drop_path(x2)
399         if self.use_checkpoint:
400             x2 = x2 + checkpoint.checkpoint(self.forward_part2, x2)
401         else:
402             x2 = x2 + self.forward_part2(x2)
403
404     FPE = PositionalEncoding3D(x.shape[4])
405
406     x = torch.add((1 - alpha) * x, alpha * x2) + self.forward_part3(FPE(x))
407
408     return x, v, k, q
409
410
411 class PatchMerging(nn.Module):
412     """ Patch Merging Layer
413
414     Args:
415         dim (int): Number of input channels.
416         norm_layer (nn.Module, optional): Normalization layer. Default: nn.LayerNorm
417     """
418
419     def __init__(self, dim, norm_layer=nn.LayerNorm):
420         super().__init__()
421         self.dim = dim
422         self.reduction = nn.Linear(4 * dim, 2 * dim, bias=False)
423         self.norm = norm_layer(4 * dim)
424
425     def forward(self, x):
426         """ Forward function.
427
428         Args:
429             x: Input feature, tensor size (B, D, H, W, C).
430         """
431         B, D, H, W, C = x.shape
432
433         # padding
434         pad_input = (H % 2 == 1) or (W % 2 == 1)
435         if pad_input:
436             x = F.pad(x, (0, 0, 0, W % 2, 0, H % 2))
437
438         x0 = x[:, :, 0::2, 0::2, :] # B D H/2 W/2 C
439         x1 = x[:, :, 1::2, 0::2, :] # B D H/2 W/2 C
440         x2 = x[:, :, 0::2, 1::2, :] # B D H/2 W/2 C
441         x3 = x[:, :, 1::2, 1::2, :] # B D H/2 W/2 C
442         x = torch.cat([x0, x1, x2, x3], -1) # B D H/2 W/2 4*C
443
444         x = self.norm(x)
445         x = self.reduction(x)
446
447         return x
448
449
450 class PatchExpandUp(nn.Module):
451     def __init__(self, input_resolution, dim, dim_scale=2, norm_layer=nn.LayerNorm):
452         super().__init__()
453         self.input_resolution = input_resolution
454         self.dim_scale = dim_scale
455         self.dim = dim
456         self.expand = nn.Linear(dim, 2 * dim, bias=False) if dim_scale == 2 else nn.Identity()
457         self.norm = norm_layer(dim // dim_scale)
458
459     def forward(self, x):
460         """
461         x: B, H*W, C
462         """
463         D, H, W = self.input_resolution
464         x = x.flatten(2).transpose(1, 2)

```

```

465         x = self.expand(x)
466         B, L, C = x.shape
467         # assert L == D * H * W, "input feature has wrong size"
468
469         x = x.view(B, 32, H, W, C)
470         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)
471
472         x = self.norm(x)
473         x = x.permute(0, 4, 1, 2, 3)
474
475         return x
476
477
478 class PatchExpand(nn.Module):
479     def __init__(self, input_resolution, dim, dim_scale=2, norm_layer=nn.LayerNorm):
480         super().__init__()
481         self.input_resolution = input_resolution
482         self.dim_scale = dim_scale
483         self.dim = dim
484         # self.expand = nn.Linear(dim, 2 * dim, bias=False) if dim_scale == 2 else nn.Identity()
485         self.expand = nn.Linear(dim, 2 * dim, bias=False) if dim_scale == 2 else nn.Identity()
486         self.norm = norm_layer(dim // dim_scale)
487
488     def forward(self, x):
489         """
490         x: B, H*W, C
491         """
492         D, H, W = self.input_resolution
493         x = x.flatten(2).transpose(1, 2)
494         x = self.expand(x)
495         B, L, C = x.shape
496         # assert L == D * H * W, "input feature has wrong size"
497
498         x = x.view(B, D * 8, H, W, C)
499         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)
500
501         x = self.norm(x)
502         x = x.permute(0, 4, 1, 2, 3)
503
504         return x
505
506
507 class FinalPatchExpand_X4(nn.Module):
508     def __init__(self, input_resolution, dim, dim_scale=4, norm_layer=nn.LayerNorm):
509         super().__init__()
510         self.input_resolution = input_resolution
511         self.dim = dim
512         self.dim_scale = dim_scale
513         self.expand = nn.Linear(dim, 4 * 16 * dim, bias=False)
514         self.output_dim = dim
515         self.norm = norm_layer(self.output_dim)
516
517     def forward(self, x):
518         """
519         x: B, H*W, C
520         """
521         D, H, W = self.input_resolution
522         x = x.permute(0, 4, 1, 2, 3)
523         x = x.flatten(2).transpose(1, 2)
524         x = self.expand(x)
525         B, L, C = x.shape
526
527         x = x.view(B, D, H, W, C)
528         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,
529                        p3=self.dim_scale,
530                        c=C // (self.dim_scale ** 3))
531         # x = x.view(B, -1, self.output_dim)
532         x = self.norm(x)
533
534         return x
535
536
537 class BasicLayer_up(nn.Module):
538     """ A basic Swin Transformer layer for one stage.
539
540     Args:
541         dim (int): Number of input channels.
542         input_resolution (tuple[int]): Input resolution.
543         depth (int): Number of blocks.
544         num_heads (int): Number of attention heads.
545         window_size tuple(int): Local window size.
546
547         mlp_ratio (float): Ratio of mlp hidden dim to embedding dim.
548         qkv_bias (bool, optional): If True, add a learnable bias to query, key, value. Default: True
549         qk_scale (float | None, optional): Override default qk scale of head_dim ** -0.5 if set.
550         drop (float, optional): Dropout rate. Default: 0.0
551         attn_drop (float, optional): Attention dropout rate. Default: 0.0
552         drop_path (float | tuple[float], optional): Stochastic depth rate. Default: 0.0
553         norm_layer (nn.Module, optional): Normalization layer. Default: nn.LayerNorm
554         downsample (nn.Module | None, optional): Downsample layer at the end of the layer. Default: None
555         use_checkpoint (bool): Whether to use checkpointing to save memory. Default: False.
556
557     """
558
559     def __init__(self, dim, input_resolution, depth, num_heads, window_size=(7, 7, 7),
560                  mlp_ratio=4., qkv_bias=True, qk_scale=None, drop=0., attn_drop=0.,
561                  drop_path=0., norm_layer=nn.LayerNorm, upsample=None, use_checkpoint=False):
562
563         super().__init__()
564         self.dim = dim
565         self.input_resolution = input_resolution
566         self.window_size = window_size
567         self.shift_size = tuple(i // 2 for i in window_size)
568         self.depth = depth
569         self.use_checkpoint = use_checkpoint
570
571         # build blocks
572         self.blocks = nn.ModuleList([
573             SwinTransformerBlock3D(
574                 dim=dim,
575                 num_heads=num_heads,
576                 window_size=window_size,
577                 shift_size=(0, 0, 0) if (i % 2 == 0) else self.shift_size,
578                 mlp_ratio=mlp_ratio,
579                 qkv_bias=qkv_bias,
580                 qk_scale=qk_scale,
581                 drop=drop,
582                 attn_drop=attn_drop,
583                 drop_path=drop_path[i] if isinstance(drop_path, list) else drop_path,
584                 norm_layer=norm_layer,
585                 use_checkpoint=use_checkpoint,
586             )
587             for i in range(depth)])

```

```

587         # patch merging layer
588         if upsample is not None:
589             self.upsample = PatchExpand_Up(input_resolution, dim=dim, dim_scale=2, norm_layer=norm_layer)
590         else:
591             self.upsample = None
592
593     def forward(self, x, prev_v1, prev_k1, prev_q1, prev_v2, prev_k2, prev_q2):
594         """ Forward function.
595
596         Args:
597             x: Input feature, tensor size (B, C, D, H, W).
598
599         """
600         # calculate attention mask for SW-MSA
601         B, C, D, H, W = x.shape
602         window_size, shift_size = get_window_size((D, H, W), self.window_size, self.shift_size)
603         x = rearrange(x, 'b c d h w -> b d h w c')
604         Dp = int(np.ceil(D / window_size[0])) * window_size[0]
605         Hp = int(np.ceil(H / window_size[1])) * window_size[1]
606         Wp = int(np.ceil(W / window_size[2])) * window_size[2]
607         attn_mask = compute_mask(Dp, Hp, Wp, window_size, shift_size, x.device)
608
609         for idx, blk in enumerate(self.blocks):
610             if idx % 2 == 0:
611                 x, _, _ = blk(x, attn_mask, prev_v1, prev_k1, prev_q1, True)
612             else:
613                 x, _, _ = blk(x, attn_mask, prev_v2, prev_k2, prev_q2, True)
614
615         # x = x.view(B, D, H, W, -1)
616
617         if self.upsample is not None:
618             x = x.permute(0, 4, 1, 2, 3)
619             x = self.upsample(x)
620             # x = rearrange(x, 'b d h w c -> b c d h w')
621         return x
622
623     # cache each stage results
624     @lru_cache()
625     def compute_mask(D, H, W, window_size, shift_size, device):
626         img_mask = torch.zeros((1, D, H, W, 1), device=device) # 1 Dp Hp Wp 1
627         cnt = 0
628         for d in slice(-window_size[0]), slice(-window_size[0], -shift_size[0]), slice(-shift_size[0], None):
629             for h in slice(-window_size[1]), slice(-window_size[1], -shift_size[1]), slice(-shift_size[1], None):
630                 for w in slice(-window_size[2]), slice(-window_size[2], -shift_size[2]), slice(-shift_size[2], None):
631                     img_mask[:, d, h, w, :] = cnt
632                     cnt += 1
633         mask_windows = window_partition(img_mask, window_size) # nW, ws[0]*ws[1]*ws[2], 1
634         mask_windows = mask_windows.squeeze(-1) # nW, ws[0]*ws[1]*ws[2]
635         attn_mask = mask_windows.unsqueeze(1) - mask_windows.unsqueeze(2)
636         attn_mask = attn_mask.masked_fill(attn_mask != 0, float(-100.0)).masked_fill(attn_mask == 0, float(0.0))
637         return attn_mask
638
639     class BasicLayer(nn.Module):
640         """ A basic Swin Transformer layer for one stage.
641
642         Args:
643             dim (int): Number of feature channels
644             depth (int): Depths of this stage.
645             num_heads (int): Number of attention head.
646             window_size (tuple[int]): Local window size. Default: (1,7,7).
647             mlp_ratio (float): Ratio of mlp hidden dim to embedding dim. Default: 4.
648             qkv_bias (bool, optional): If True, add a learnable bias to query, key, value. Default: True
649             qk_scale (float | None, optional): Override default qk scale of head_dim ** -0.5 if set.
650             drop (float, optional): Dropout rate. Default: 0.0
651             attn_drop (float, optional): Attention dropout rate. Default: 0.0
652             drop_path (float | tuple[float], optional): Stochastic depth rate. Default: 0.0
653             norm_layer (nn.Module, optional): Normalization layer. Default: nn.LayerNorm
654             downsampler (nn.Module | None, optional): Downsampler layer at the end of the layer. Default: None
655
656         """
657
658         def __init__(self,
659                     dim,
660                     depth,
661                     num_heads,
662                     window_size=(1, 7, 7),
663                     mlp_ratio=4.,
664                     qkv_bias=False,
665                     qk_scale=None,
666                     drop=0.,
667                     attn_drop=0.,
668                     drop_path=0.,
669                     drop_path_rate=0.,
670                     norm_layer=nn.LayerNorm,
671                     downsampler=None,
672                     use_checkpoint=False):
673             super().__init__()
674             self.window_size = window_size
675             self.shift_size = tuple(i // 2 for i in window_size)
676             self.depth = depth
677             self.use_checkpoint = use_checkpoint
678
679             # build blocks
680             self.blocks = nn.ModuleList([
681                 SwinTransformerBlock3D(
682                     dim=dim,
683                     num_heads=num_heads,
684                     window_size=window_size,
685                     shift_size=(0, 0, 0) if (i % 2 == 0) else self.shift_size,
686                     mlp_ratio=mlp_ratio,
687                     qkv_bias=qkv_bias,
688                     qk_scale=qk_scale,
689                     drop=drop,
690                     attn_drop=attn_drop,
691                     drop_path=drop_path[i] if isinstance(drop_path, list) else drop_path,
692                     norm_layer=norm_layer,
693                     use_checkpoint=use_checkpoint,
694                 )
695                 for i in range(depth)])
696
697             self.downsample = downsampler
698             if self.downsample is not None:
699                 self.downsample = downsampler(dim=dim, norm_layer=norm_layer)
700
701     def forward(self, x, block_num):
702         """ Forward function.
703
704         Args:
705             x: Input feature, tensor size (B, C, D, H, W).
706
707         """
708         # calculate attention mask for SW-MSA

```

```

709     B, C, D, H, W = x.shape
710     window_size, shift_size = get_window_size((D, H, W), self.window_size, self.shift_size)
711     x = rearrange(x, 'b c d h w -> b d h w c')
712
713     Dp = int(np.ceil(D / window_size[0])) * window_size[0]
714     Hp = int(np.ceil(H / window_size[1])) * window_size[1]
715     Wp = int(np.ceil(W / window_size[2])) * window_size[2]
716
717     attn_mask = compute_mask(Dp, Hp, Wp, window_size, shift_size, x.device)
718
719     v1, k1, q1, v2, k2, q2 = None, None, None, None, None, None
720
721     for idx, blk in enumerate(self.blocks):
722         if idx % 2 == 0:
723             x, v1, k1, q1 = blk(x, attn_mask, None, None, None)
724         else:
725             x, v2, k2, q2 = blk(x, attn_mask, None, None, None)
726
727     x = x.reshape(B, D, H, W, -1)
728
729     if self.downsample is not None:
730         x = self.downsample(x)
731     x = rearrange(x, 'b d h w c -> b c d h w')
732
733     return x, v1, k1, q1, v2, k2, q2
734
735 class PatchEmbed3D(nn.Module):
736     """ Video to Patch Embedding.
737
738     Args:
739         patch_size (int): Patch token size. Default: (2,4,4).
740         in_chans (int): Number of input video channels. Default: 3.
741         embed_dim (int): Number of linear projection output channels. Default: 96.
742         norm_layer (nn.Module, optional): Normalization layer. Default: None
743     """
744
745     def __init__(self, img_size=(128, 128, 128), patch_size=(4, 4, 4), in_chans=3, embed_dim=96, norm_layer=None):
746         super().__init__()
747         self.patch_size = patch_size
748
749         self.in_chans = in_chans
750         self.embed_dim = embed_dim
751         patches_resolution = [img_size[0] // patch_size[0], img_size[1] // patch_size[1], img_size[2] // patch_size[2]]
752         self.patches_resolution = patches_resolution
753
754         self.proj = nn.Conv3d(in_chans, embed_dim, kernel_size=patch_size, stride=patch_size)
755         if norm_layer is not None:
756             self.norm = norm_layer(embed_dim)
757         else:
758             self.norm = None
759
760     def forward(self, x):
761         """Forward function."""
762         # padding
763         _, _, D, H, W = x.size()
764         if W % self.patch_size[2] != 0:
765             x = F.pad(x, (0, self.patch_size[2] - W % self.patch_size[2]))
766         if H % self.patch_size[1] != 0:
767             x = F.pad(x, (0, 0, 0, self.patch_size[1] - H % self.patch_size[1]))
768         if D % self.patch_size[0] != 0:
769             x = F.pad(x, (0, 0, 0, 0, self.patch_size[0] - D % self.patch_size[0]))
770
771         x = self.proj(x) # B C D Wh Ww
772         if self.norm is not None:
773             D, Wh, Ww = x.size(2), x.size(3), x.size(4)
774             x = x.flatten(2).transpose(1, 2)
775             x = self.norm(x)
776             x = x.transpose(1, 2).view(-1, self.embed_dim, D, Wh, Ww)
777
778         return x
779
780 class SwinTransformerSys3D(nn.Module):
781     """ Swin Transformer
782     A PyTorch impl of : `Swin Transformer: Hierarchical Vision Transformer using Shifted Windows` -
783         https://arxiv.org/pdf/2103.14030
784
785     Args:
786         img_size (int | tuple(int)): Input image size. Default 224
787         patch_size (int | tuple(int)): Patch size. Default: 4
788         in_chans (int): Number of input image channels. Default: 3
789         num_classes (int): Number of classes for classification head. Default: 1000
790         embed_dim (int): Patch embedding dimension. Default: 96
791         depths (tuple(int)): Depth of each Swin Transformer layer.
792         num_heads (tuple(int)): Number of attention heads in different layers.
793         window_size (tuple(int)): Window size. Default: (7,7,7)
794         mlp_ratio (float): Ratio of mlp hidden dim to embedding dim. Default: 4
795         qkv_bias (bool): If True, add a learnable bias to query, key, value. Default: True
796         qk_scale (float): Override default qk scale of head_dim ** -0.5 if set. Default: None
797         drop_rate (float): Dropout rate. Default: 0
798         attn_drop_rate (float): Attention dropout rate. Default: 0
799         drop_path_rate (float): Stochastic depth rate. Default: 0.1
800
801         norm_layer (nn.Module): Normalization layer. Default: nn.LayerNorm.
802         ape (bool): If True, add absolute position embedding to the patch embedding. Default: False
803         patch_norm (bool): If True, add normalization after patch embedding. Default: True
804         use_checkpoint (bool): Whether to use checkpointing to save memory. Default: False
805     """
806
807     def __init__(self, pretrained=None,
808                  pretrained2d=True,
809                  img_size=(128, 128, 128),
810                  patch_size=(4, 4, 4),
811                  in_chans=4,
812                  num_classes=3,
813                  embed_dim=96,
814                  depths=[2, 2, 2, 1],
815                  depths_decoder=[1, 2, 2, 2],
816                  num_heads=[3, 6, 12, 24],
817                  window_size=(7, 7, 7),
818                  mlp_ratio=4.,
819                  qkv_bias=True,
820                  qk_scale=None,
821                  drop_rate=0.,
822                  attn_drop_rate=0.,
823                  drop_path_rate=0.1,
824                  norm_layer=nn.LayerNorm,
825                  patch_norm=True,
826                  use_checkpoint=False,
827                  frozen_stages=-1,
828                  final_upsample="expand_first", **kwargs):
829         super().__init__()
830

```

```

831     print("SwinTransformerSys3D expand initial---depths:{};depths_decoder:{};drop_path_rate:{};num_classes:{};embed_dims:{};window:{}".format(
832         depths,
833         depths_decoder, drop_path_rate, num_classes, embed_dim, window_size))
834
835     self.pretrained = pretrained # None
836     self.pretrained2d = pretrained2d # True
837     self.num_classes = num_classes # 3
838     self.num_layers = len(depths) # number of swin transformer block
839     self.embed_dim = embed_dim # number of patch embedding size (default: 96)
840     self.patch_norm = patch_norm
841     self.num_features = int(embed_dim * 2 ** (self.num_layers - 1))
842     self.num_features_up = int(embed_dim * 2)
843     self.mlp_ratio = mlp_ratio
844     self.final_upsample = final_upsample
845     self.frozen_stages = frozen_stages
846
847     # split image into non-overlapping patches
848     self.patch_embed = PatchEmbed3D(img_size=img_size,
849                                     patch_size=patch_size, in_chans=in_chans, embed_dim=embed_dim,
850                                     norm_layer=norm_layer if self.patch_norm else None)
851
852     patches_resolution = self.patch_embed.patches_resolution
853     self.patches_resolution = patches_resolution
854
855     self.pos_drop = nn.Dropout(p=drop_rate)
856
857     # stochastic depth
858     dpr = [x.item() for x in torch.linspace(0, drop_path_rate, sum(depths))] # stochastic depth decay rule
859
860     # build encoder and bottleneck layers
861     self.layers = nn.ModuleList()
862     for i_layer in range(self.num_layers):
863         layer = BasicLayer(
864             dim=int(embed_dim * 2 ** i_layer),
865             depth=depths[i_layer],
866             depths=depths,
867             num_heads=num_heads[i_layer],
868             window_size=window_size,
869             mlp_ratio=mlp_ratio,
870             qkv_bias=qkv_bias,
871             qk_scale=qk_scale,
872             drop=drop_rate,
873             attn_drop=attn_drop_rate,
874             drop_path=dpr[sum(depths[:i_layer]):sum(depths[:i_layer + 1])],
875             drop_path_rate=drop_path_rate,
876             norm_layer=norm_layer,
877             downsample=PatchMerging if i_layer < self.num_layers - 1 else None,
878             use_checkpoint=use_checkpoint)
879         self.layers.append(layer)
880
881     # build decoder layers
882     self.layers_up = nn.ModuleList()
883     self.concat_back_dim = nn.ModuleList()
884     for i_layer in range(self.num_layers):
885         concat_linear = nn.Linear(2 * int(embed_dim * 2 ** (self.num_layers - 1 - i_layer)),
886                                   int(embed_dim * 2 ** (
887                                       self.num_layers - 1 - i_layer)),
888                                   bias=False) if i_layer > 0 else nn.Identity()
889         if i_layer == 0:
890             layer_up = PatchExpand(
891                 input_resolution=(patches_resolution[0] // (2 ** (self.num_layers - 1 - i_layer)),
892                               patches_resolution[1] // (2 ** (self.num_layers - 1 - i_layer)),
893                               patches_resolution[2] // (2 ** (self.num_layers - 1 - i_layer))),
894                 dim=int(embed_dim * 2 ** (self.num_layers - 1 - i_layer)), dim_scale=2, norm_layer=norm_layer)
895         else:
896             layer_up = BasicLayer_up(
897                 dim=int(embed_dim * 2 ** (self.num_layers - 1 - i_layer)),
898                 input_resolution=(patches_resolution[0] // (2 ** (self.num_layers - 1 - i_layer)),
899                               patches_resolution[1] // (2 ** (self.num_layers - 1 - i_layer)),
900                               patches_resolution[2] // (2 ** (self.num_layers - 1 - i_layer))),
901                 depth=depths[(self.num_layers - 1 - i_layer)],
902                 num_heads=num_heads[(self.num_layers - 1 - i_layer)],
903                 window_size=window_size,
904                 mlp_ratio=mlp_ratio,
905                 qkv_bias=qkv_bias,
906                 qk_scale=qk_scale,
907                 drop=drop_rate,
908                 attn_drop=attn_drop_rate,
909                 drop_path=dpr[sum(depths[(self.num_layers - 1 - i_layer)):sum(
910                     depths[(self.num_layers - 1 - i_layer) + 1])],
911                 norm_layer=norm_layer,
912                 upsample=PatchExpand if (i_layer < self.num_layers - 1) else None,
913                 use_checkpoint=use_checkpoint)
914             self.layers_up.append(layer_up)
915         self.concat_back_dim.append(concat_linear)
916
917     self.norm = norm_layer(self.num_features)
918     self.norm_up = norm_layer(self.embed_dim)
919
920     if self.final_upsample == "expand_first":
921         print("---final upsample expand first---")
922         self.up = FinalPatchExpand_X4(input_resolution=(
923             img_size[0] // patch_size[0], img_size[1] // patch_size[1], img_size[2] // patch_size[2]),
924             dim_scale=4, dim=embed_dim)
925         self.output = nn.Conv3d(in_channels=embed_dim, out_channels=self.num_classes, kernel_size=1, bias=False)
926         self.softmax = nn.Sigmoid()
927
928     self._freeze_stages()
929
930
931 @torch.jit.ignore
932 def no_weight_decay(self):
933     return {'absolute_pos_embed'}
934
935 @torch.jit.ignore
936 def no_weight_decay_keywords(self):
937     return {'Relative_position_bias_table'}
938
939 # Encoder and Bottleneck
940 def forward_features(self, x):
941     x = self.patch_embed(x)
942     x = self.pos_drop(x)
943     x_downsample = []
944     v_values_1 = []
945     k_values_1 = []
946     q_values_1 = []
947     v_values_2 = []
948     k_values_2 = []
949     q_values_2 = []
950
951     for i_layer in enumerate(self.layers):
952         x_downsample.append(x)

```

```

953         x, v1, k1, q1, v2, k2, q2 = layer(x, i)
954         v_values_1.append(v1)
955         k_values_1.append(k1)
956         q_values_1.append(q1)
957         v_values_2.append(v2)
958         k_values_2.append(k2)
959         q_values_2.append(q2)
960
961     x = rearrange(x, 'n c d h w -> n d h w c')
962     x = self.norm(x)
963     x = rearrange(x, 'n d h w c -> n c d h w')
964
965     return x, x_downsample, v_values_1, k_values_1, q_values_1, v_values_2, k_values_2, q_values_2
966
967 # Decoder and Skip connection
968 def forward_up_features(self, x, x_downsample, v_values_1, k_values_1, q_values_1, v_values_2, k_values_2,
969                        q_values_2):
970     for inx, layer_up in enumerate(self.layers_up):
971         if inx == 0:
972             x = layer_up(x)
973         else:
974             x = torch.cat([x, x_downsample[3 - inx]], 1)
975             B, C, D, H, W = x.shape
976             x = x.flatten(2).transpose(1, 2)
977             x = self.concat_back_dim[inx](x)
978             _, _, C = x.shape
979             x = x.view(B, D, H, W, C)
980
981             x = x.permute(0, 4, 1, 2, 3)
982             x = layer_up(x, v_values_1[3 - inx], k_values_1[3 - inx], q_values_1[3 - inx], v_values_2[3 - inx],
983                          k_values_2[3 - inx], q_values_2[3 - inx])
984
985     x = self.norm_up(x)
986
987     return x
988
989 def up_x4(self, x):
990     D, H, W = self.patches_resolution
991     B, _, _, C = x.shape
992
993     if self.final_upsample == "expand_first":
994         x = self.up(x)
995         x = x.view(B, 4 * D, 4 * H, 4 * W, -1)
996         x = x.permute(0, 4, 1, 2, 3) # B,C,D,H,W
997         x = self.output(x)
998         x = self.softmax(x)
999     return x
1000
1001 def freeze_stages(self):
1002     if self.frozen_stages >= 0:
1003         self.patch_embed.eval()
1004         for param in self.patch_embed.parameters():
1005             param.requires_grad = False
1006
1007     if self.frozen_stages >= 1:
1008         self.pos_drop.eval()
1009         for i in range(0, self.frozen_stages):
1010             m = self.layers[i]
1011             m.eval()
1012             for param in m.parameters():
1013                 param.requires_grad = False
1014
1015 def inflate_weights(self):
1016     """Inflate the swin2d parameters to swin3d.
1017
1018     The differences between swin3d and swin2d mainly lie in an extra
1019     axis. To utilize the pretrained parameters in 2d model,
1020     the weight of swin2d models should be inflated to fit in the shapes of
1021     the 3d counterpart.
1022
1023     Args:
1024         logger (logging.Logger): The logger used to print
1025             debugging information.
1026     """
1027     checkpoint = torch.load(self.pretrained, map_location='cpu')
1028     state_dict = checkpoint['model']
1029
1030     # delete relative position index since we always re-init it
1031     relative_position_index_keys = [k for k in state_dict.keys() if "relative_position_index" in k]
1032     for k in relative_position_index_keys:
1033         del state_dict[k]
1034
1035     # delete attn_mask since we always re-init it
1036     attn_mask_keys = [k for k in state_dict.keys() if "attn_mask" in k]
1037     for k in attn_mask_keys:
1038         del state_dict[k]
1039
1040     state_dict['patch_embed.proj.weight'] = state_dict['patch_embed.proj.weight'].unsqueeze(2).repeat(1, 1,
1041                                                                                                     self.patch_size[
1042                                                                                                     0], 1,
1043                                                                                                     1) / \
1044                                                                                                     self.patch_size[0]
1045
1046     # bicubic interpolate relative_position_bias_table if not match
1047     relative_position_bias_table_keys = [k for k in state_dict.keys() if "relative_position_bias_table" in k]
1048     for k in relative_position_bias_table_keys:
1049         relative_position_bias_table_pretrained = state_dict[k]
1050         relative_position_bias_table_current = self.state_dict()[k]
1051         L1, nH1 = relative_position_bias_table_pretrained.size()
1052         L2, nH2 = relative_position_bias_table_current.size()
1053         L2 = (2 * self.window_size[1] - 1) * (2 * self.window_size[2] - 1)
1054         wd = self.window_size[0]
1055         if nH1 != nH2:
1056             print(f"Error in loading {k}, passing")
1057         else:
1058             if L1 != L2:
1059                 S1 = int(L1 ** 0.5)
1060                 relative_position_bias_table_pretrained_resized = torch.nn.functional.interpolate(
1061                     relative_position_bias_table_pretrained.permute(1, 0).view(1, nH1, S1, S1),
1062                     size=(2 * self.window_size[1] - 1, 2 * self.window_size[2] - 1),
1063                     mode='bicubic')
1064                 relative_position_bias_table_pretrained = relative_position_bias_table_pretrained_resized.view(nH2,
1065                                                                                                     L2).permute(
1066                                                                                                     1, 0)
1067             state_dict[k] = relative_position_bias_table_pretrained.repeat(2 * wd - 1, 1)
1068
1069 msg = self.load_state_dict(state_dict, strict=False)
1070 print(msg)
1071 print(f"=> loaded successfully '{self.pretrained}'")
1072 del checkpoint
1073 torch.cuda.empty_cache()

```

```

1075
1076 def init_weights(self, pretrained=None):
1077     """Initialize the weights in backbone.
1078
1079     Args:
1080         pretrained (str, optional): Path to pre-trained weights.
1081             Defaults to None.
1082     """
1083
1084     def _init_weights(m):
1085         if isinstance(m, nn.Linear):
1086             trunc_normal_(m.weight, std=.02)
1087             if isinstance(m, nn.Linear) and m.bias is not None:
1088                 nn.init.constant_(m.bias, 0)
1089         elif isinstance(m, nn.LayerNorm):
1090             nn.init.constant_(m.bias, 0)
1091             nn.init.constant_(m.weight, 1.0)
1092
1093     if pretrained:
1094         self.pretrained = pretrained
1095     if isinstance(self.pretrained, str):
1096         self.apply(_init_weights)
1097
1098         print(f'load model from: {self.pretrained}')
1099
1100         if self.pretrained2d:
1101             # Inflate 2D model into 3D model.
1102             self.inflate_weights()
1103         else:
1104             # Directly load 3D model.
1105             pass
1106         # load_checkpoint(self, self.pretrained, strict=False)
1107     elif self.pretrained is None:
1108         self.apply(_init_weights)
1109     else:
1110         raise TypeError('pretrained must be a str or None')
1111
1112 def forward(self, x):
1113     x, x_downsample, v_values_1, k_values_1, q_values_1, v_values_2, k_values_2, q_values_2 = self.forward_features(
1114         x)
1115
1116     x = self.forward_up_features(x, x_downsample, v_values_1, k_values_1, q_values_1, v_values_2, k_values_2,
1117                                q_values_2)
1118     x = self.up_x4(x)
1119
1120     return x
1121
1122
1123 if __name__ == '__main__':
1124
1125     swin_unet = SwinTransformerSys3D(img_size=(128, 128, 128),
1126                                     patch_size=(4, 4, 4),
1127                                     in_chans=1,
1128                                     num_classes=2,
1129                                     )
1130     x1 = torch.randn(1, 4, 128, 128, 128)
1131
1132     x2 = torch.randn(1, 4, 128, 128, 128)
1133
1134     y = swin_unet(x1)
1135     embed()

```



```

0 SparrowLink/modify_key_in_config.py
1 import yaml
2 import argparse
3 import pathlib
4
5 if __name__ == "__main__":
6     parser = argparse.ArgumentParser()
7     parser.add_argument("--config_path", type=str, default=None)
8
9     args = parser.parse_args()
10    assert args.config_path is not None, "config_path is None"
11    name = pathlib.Path(args.config_path).name
12    parent = pathlib.Path(args.config_path).parent
13    with open(args.config_path, 'r') as f:
14        config = yaml.load(f, Loader=yaml.FullLoader)
15        # full key = ['I_M', 'I_A', 'CS_M', 'CS_A', 'CS_DL']
16        # config['infer']['key'] = ['I_M', 'I_A', 'CS_M', 'CS_A', 'CS_DL']
17        # config['train']['key'] = ['I_M', 'I_A', 'CS_M', 'CS_A', 'CS_DL']
18        # config['model']['in_channels'] = len(config['infer']['key'])
19        # config['model']['model_parameter']['num_input_channels'] = len(config['infer']['key'])
20        # with open(parent / name.replace(".yaml", f"_new_0.yaml"), 'w') as f:
21        #     yaml.dump(config, f, encoding='utf-8', allow_unicode=True)
22        #
23        # config['infer']['key'] = ['I_M', 'CS_M', 'CS_A', 'CS_DL']
24        # config['train']['key'] = ['I_M', 'CS_M', 'CS_A', 'CS_DL']
25        # config['model']['in_channels'] = len(config['infer']['key'])
26        # config['model']['model_parameter']['num_input_channels'] = len(config['infer']['key'])
27        # with open(parent / name.replace(".yaml", f"_new_1.yaml"), 'w') as f:
28        #     yaml.dump(config, f, encoding='utf-8', allow_unicode=True)
29        #
30        # config['infer']['key'] = ['I_M', 'I_A', 'CS_A', 'CS_DL']
31        # config['train']['key'] = ['I_M', 'I_A', 'CS_A', 'CS_DL']
32        # config['model']['in_channels'] = len(config['infer']['key'])
33        # config['model']['model_parameter']['num_input_channels'] = len(config['infer']['key'])
34        # with open(parent / name.replace(".yaml", f"_new_2.yaml"), 'w') as f:
35        #     yaml.dump(config, f, encoding='utf-8', allow_unicode=True)
36        #
37        # config['infer']['key'] = ['I_M', 'I_A', 'CS_M', 'CS_DL']
38        # config['train']['key'] = ['I_M', 'I_A', 'CS_M', 'CS_DL']
39        # config['model']['in_channels'] = len(config['infer']['key'])
40        # config['model']['model_parameter']['num_input_channels'] = len(config['infer']['key'])
41        # with open(parent / name.replace(".yaml", f"_new_3.yaml"), 'w') as f:
42        #     yaml.dump(config, f, encoding='utf-8', allow_unicode=True)
43        #
44        # config['infer']['key'] = ['I_M', 'I_A', 'CS_M', 'CS_A']
45        # config['train']['key'] = ['I_M', 'I_A', 'CS_M', 'CS_A']
46        # config['model']['in_channels'] = len(config['infer']['key'])
47        # config['model']['model_parameter']['num_input_channels'] = len(config['infer']['key'])
48        # with open(parent / name.replace(".yaml", f"_new_4.yaml"), 'w') as f:
49        #     yaml.dump(config, f, encoding='utf-8', allow_unicode=True)
50        #
51        # config['infer']['key'] = ['I_M']
52        # config['train']['key'] = ['I_M']
53        # config['model']['in_channels'] = len(config['infer']['key'])
54        # config['model']['model_parameter']['num_input_channels'] = len(config['infer']['key'])
55        # with open(parent / name.replace(".yaml", f"_new_I_M.yaml"), 'w') as f:
56        #     yaml.dump(config, f, encoding='utf-8', allow_unicode=True)
57        #
58        config['infer']['key'] = ['I_M', 'CS_M', 'CS_DL']
59        config['train']['key'] = ['I_M', 'CS_M', 'CS_DL']
60        config['model']['in_channels'] = len(config['infer']['key'])
61        config['model']['model_parameter']['num_input_channels'] = len(config['infer']['key'])
62        with open(parent / name.replace(".yaml", f"_new_no_a.yaml"), 'w') as f:
63            yaml.dump(config, f, encoding='utf-8', allow_unicode=True)
64        #
65        config['infer']['key'] = ['I_M', 'I_A', 'CS_A', 'CS_DL']
66        config['train']['key'] = ['I_M', 'CS_M', 'CS_A', 'CS_DL']
67        config['model']['in_channels'] = len(config['infer']['key'])
68        config['model']['model_parameter']['num_input_channels'] = len(config['infer']['key'])
69        with open(parent / name.replace(".yaml", f"_new_MMAD.yaml"), 'w') as f:
70            yaml.dump(config, f, encoding='utf-8', allow_unicode=True)

```

```

0 SparrowLink/post_processing/move_file.py
1 import pathlib
2 import argparse
3 import shutil
4 from tqdm import tqdm
5
6 if __name__ == "__main__":
7     parser = argparse.ArgumentParser()
8     parser.add_argument('--data_path',
9                         type=str,
10                        help='move discontinuity detection file to sphere or cube file for easiler dataloading',
11                        default='test')
12     parser.add_argument('--save_path',
13                         type=str,
14                        help='save path',
15                        default='test')
16     parser.add_argument('--data_postfix',
17                         type=str,
18                        help='sphere or cube',
19                        default='')
20     parser.add_argument('--hierarchical',
21                         type=str,
22                        help='sphere or cube',
23                        default='')
24     parser.add_argument('--save_postfix',
25                         type=str,
26                        help='discontinuity detection file',
27                        default='')
28     parser.add_argument('--separate_folder',
29                        action="store_true",
30                        default=False,
31                        )
32
33     args = parser.parse_args()
34     data_path = pathlib.Path(args.data_path)
35     save_path = pathlib.Path(args.save_path)
36     save_path.mkdir(exist_ok=True, parents=True)
37     data_list = list(data_path.glob(f'*{args.hierarchical}{args.data_postfix}.nii.gz'))
38     assert len(data_list) > 0, f"no file found in {data_path} {args.data_postfix}"
39     print(f"move file for better viewing from \n"
40           f"{args.data_path} to \n"
41           f"{args.save_path}, \n"
42           f"finding={args.hierarchical}{args.data_postfix}, save_postfix={args.save_postfix}",)
43     pbar = tqdm(total=len(data_list))
44     pbar.set_description('copy file')
45     for file in data_list:
46         file_name = file.name
47         new_file_name = file_name.replace(f'{args.data_postfix}.nii.gz', f'{args.save_postfix}.nii.gz')
48         if args.separate_folder:
49             patient_id = file_name[:9]
50             (save_path / patient_id).mkdir(exist_ok=True, parents=True)
51             new_file = save_path / patient_id / new_file_name
52         else:
53             new_file = save_path / new_file_name
54         shutil.copy(file, new_file)
55         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.

```

0 SparrowLink/registration/basic_registration.py
1 import pathlib
2 import ants
3 import time
4 import argparse
5 from multiprocessing import Pool
6
7
8 def Reg(fixed_path, moving_path, apply_path, save_path, save_path_apply, mode='SyNRA', reg_interpolator='linear',
9         apply_interpolator='nearestNeighbor'):
10     tic = time.time()
11     fixed = ants.image_read(str(fixed_path))
12     moving = ants.image_read(str(moving_path))
13     apply = ants.image_read(str(apply_path))
14     reg = ants.registration(fixed=fixed, moving=moving, type_of_transform=mode)
15     moving_reg = ants.apply_transforms(fixed=fixed, moving=moving, transformlist=reg['fwdtransforms'],
16                                       interpolator=reg_interpolator)
17     apply_reg = ants.apply_transforms(fixed=fixed, moving=apply, transformlist=reg['fwdtransforms'],
18                                     interpolator=apply_interpolator)
19
20     ants.image_write(moving_reg, str(save_path / moving_path.name))
21     ants.image_write(apply_reg, str(save_path_apply / moving_path.name))
22
23     return {"name": moving_path.name, "time": time.time()-tic}
24
25
26 def update(pbar, result):
27     pbar.update()
28     # print(result)
29
30
31 def call_fun(result):
32     print(result)
33
34
35 def errorback(err):
36     print(err)
37
38
39 if __name__ == "__main__":
40     arg = argparse.ArgumentParser()
41     arg.add_argument('--reg_algorithm', type=str, default='SyNRA') # SyNRA, Rigid
42     arg.add_argument('--fixed_dir', type=str,
43                     default="/public/home/v-xiongxx/Graduate_project/Cardio_vessel_segmentaion_based_on_monai/data/ZhangX/CT_Img/")
44     arg.add_argument('--moving_dir', type=str,
45                     default="/public/home/v-xiongxx/Graduate_project/Cardio_vessel_segmentaion_based_on_monai/data/ZhangX/CTA_Img/")
46     arg.add_argument('--apply_dir', type=str,
47                     default="/public/home/v-xiongxx/Graduate_project/Cardio_vessel_segmentaion_based_on_monai/data/ZhangX/CTA_infer/")
48     arg.add_argument('--save_dir', type=str,
49                     default="/public/home/v-xiongxx/Graduate_project/Cardio_vessel_segmentaion_based_on_monai/data/ZhangX/CTA_Img_reg/")
50     arg.add_argument('--save_dir_apply', type=str,
51                     default="/public/home/v-xiongxx/Graduate_project/Cardio_vessel_segmentaion_based_on_monai/data/ZhangX/CTA_infer_reg/")
52
53
54     args = arg.parse_args()
55     fixed_dir = pathlib.Path(args.fixed_dir)
56     moving_dir = pathlib.Path(args.moving_dir)
57     apply_dir = pathlib.Path(args.apply_dir)
58     save_dir = pathlib.Path(args.save_dir)
59     save_dir_apply = pathlib.Path(args.save_dir_apply)
60     save_dir.mkdir(exist_ok=True)
61     save_dir_apply.mkdir(exist_ok=True)
62     print(f"-----{args.reg_algorithm}_registration-----")
63     pool = Pool(4)
64     for file in fixed_dir.glob('*.nii.gz'):
65         name = file.name
66         moving_path = moving_dir / name.replace('CT', 'CTA')
67         apply_path = apply_dir / name.replace('CT', 'CTA')
68         pool.apply_async(Reg,
69                         args=(file, moving_path, apply_path, save_dir, save_dir_apply, args.reg_algorithm),
70                         callback=call_fun,
71                         error_callback=errorback)
72
73     pool.close()
74     pool.join()

```

```

0 SparrowLink/registration/label_registration.py
1 import pathlib
2 import ants
3 import time
4 import argparse
5 from multiprocessing import Pool
6
7
8 def Reg(target_path, moving_path, target_path_1, moving_path_1, save_path, save_path_1, mode='SyNRA', t=1):
9     tic = time.time()
10    target = ants.image_read(str(target_path))
11    moving = ants.image_read(str(moving_path))
12    target_1 = ants.image_read(str(target_path_1))
13    moving_1 = ants.image_read(str(moving_path_1))
14    moving_loop = moving
15    moving_1_loop = moving_1
16    target_loop = target
17    target_1_loop = target_1
18    for i in range(t):
19        reg = ants.registration(fixed=target, moving=moving_loop, type_of_transform=mode)
20        moving_loop = ants.apply_transforms(fixed=target, moving=moving_loop, transformlist=reg['fwdtransforms'], interpolator='nearestNeighbor')
21        moving_1_loop = ants.apply_transforms(fixed=target_1, moving=moving_1_loop, transformlist=reg['fwdtransforms'])
22        target_loop = ants.apply_transforms(fixed=moving, moving=target_loop, transformlist=reg['invtransforms'], interpolator='nearestNeighbor')
23        target_1_loop = ants.apply_transforms(fixed=moving_1_loop, moving=target_1_loop, transformlist=reg['invtransforms'])
24
25
26    ants.image_write(moving_loop, str(save_path))
27    ants.image_write(moving_1_loop, str(save_path_1))
28
29    save_path_2 = save_path.parent.parent / save_path.parent.name.replace('auxiliary', 'main')
30    save_path_2.mkdir(exist_ok=True, parents=True) if not save_path_2.exists() else None
31    save_path_2 = save_path_2 / save_path.name
32    save_path_3 = save_path_1.parent.parent / save_path_1.parent.name.replace('auxiliary', 'main')
33    save_path_3.mkdir(exist_ok=True, parents=True)
34    save_path_3 = save_path_3 / save_path_1.name
35
36    ants.image_write(target_loop, str(save_path_2))
37    ants.image_write(target_1_loop, str(save_path_3))
38
39    return {"name": moving_path.name, "time": time.time()-tic}
40
41
42 def update(pbar, result):
43     pbar.update()
44     # print(result)
45
46
47 def call_fun(result):
48     print(result)
49
50
51 def errorback(err):
52     print(err)
53
54
55 if __name__ == "__main__":
56     arg = argparse.ArgumentParser()
57     arg.add_argument('--reg_algorithm', type=str, default='SyNRA') # SyNRA, Rigid
58     arg.add_argument('--mode', type=str, help='used in save name', default='test')
59     arg.add_argument('--time', type=int, help='time for registration loop', default=1)
60     arg.add_argument('--net', type=str, default='used in save name')
61     arg.add_argument('--save_root', type=str, help='save path for img and mask after registration',
62                     default="/public/home/v-xiongxx/Graduate project/"
63                             "Cardio_vessel_segmentaion based on monai/"
64                             "experiments/Graduate_project/multi phase/pretrain")
65     arg.add_argument('--main_mask_path', type=str, default=None)
66     arg.add_argument('--auxiliary_mask_path', type=str, default=None)
67     arg.add_argument('--main_img_path', type=str, default=None)
68     arg.add_argument('--auxiliary_img_path', type=str, default=None)
69
70
71     args = arg.parse_args()
72     main_label_path = pathlib.Path(args.main_mask_path)
73     auxiliary_label_path = pathlib.Path(args.auxiliary_mask_path)
74     main_img_path = pathlib.Path(args.main_img_path)
75     auxiliary_img_path = pathlib.Path(args.auxiliary_img_path)
76     save_auxiliary_label_registration = pathlib.Path(args.save_root) / f"auxiliary_{args.mode}_infer_{args.reg_algorithm}_reg_{args.time}"
77     save_auxiliary_img_registration = pathlib.Path(args.save_root) / f"auxiliary_{args.mode}_img_{args.reg_algorithm}_reg_{args.time}"
78     save_auxiliary_img_registration.mkdir(exist_ok=True, parents=True)
79     save_auxiliary_label_registration.mkdir(exist_ok=True, parents=True)
80     # find the corresponding file in segment syst
81     main_list = list(main_img_path.glob('*.nii.gz'))
82     # sort the file name
83
84     main_list.sort(key=lambda x: str(x.stem))
85     # q: what is x.stem?
86
87     pool = Pool(4)
88     print(f"-----{args.mode}_registration-----")
89     for i, img in enumerate(main_list):
90         img_name = img.name
91
92         main_img = img
93         main_label = main_label_path / img_name
94         auxiliary_img = auxiliary_img_path / img_name
95         auxiliary_label = auxiliary_label_path / img_name
96         save_auxiliary_label = save_auxiliary_label_registration / img_name
97         save_auxiliary_img = save_auxiliary_img_registration / img_name
98
99         assert main_label.exists(), f"{str(main_label)} not exist"
100        assert main_img.exists(), f"{str(main_img)} not exist"
101        assert auxiliary_img.exists(), f"{str(auxiliary_img)} not exist"
102        assert auxiliary_label.exists(), f"{str(auxiliary_label)} not exist"
103        # Reg(target_path=main_label,
104        #     moving_path=auxiliary_label,
105        #     target_path_1=main_img,
106        #     moving_path_1=auxiliary_img,
107        #     save_path=save_auxiliary_img,
108        #     save_path_1=save_auxiliary_label,
109        #     mode=args.mode)
110        pool.apply_async(Reg,
111                        args=(main_label, auxiliary_label, main_img, auxiliary_img, save_auxiliary_label, save_auxiliary_img, args.reg_algorithm, args.time),
112                        callback=call_fun,
113                        error_callback=errorback)
114
115    pool.close()
116    pool.join()

```

```

1 SparrowLink/second_stage_main.py
2 from monai.utils import first, set_determinism
3 from monai.data import CacheDataset, DataLoader, Dataset, decollate_batch, ITKReader, PersistentDataset
4 from monai.config import print_config
5 from monai.metrics.meandice import DiceMetric
6 import torch
7 import matplotlib.pyplot as plt
8 import os
9 import argparse
10 import monai
11 import logging
12 from utils.Config import Config
13 import sys
14 import time
15 from data_loader import prepare_datalist, prepare_datalist_with_file, \
16     prepare_image_list, save_json, write_data_reference, load_json, prepare_main_auxiliary_with_img_datalist_with_file
17 from transform.utils import get_transform, get_multi_phase_transform_with_image
18 from utils.test import cardio_vessel_segmentation_test, cardio_vessel_segmentation_multi_phase_with_image_test
19 from utils.trainer import cardio_vessel_segmentation_multi_phase_with_image_train
20 from utils.inferer import cardio_vessel_segmentation_infer, cardio_vessel_segmentation_multi_phase_with_image_infer
21 import pathlib
22 import torch.nn as nn
23 from torch.distributions.normal import Normal
24 # print_config()
25
26 torch.multiprocessing.set_sharing_strategy('file_system')
27
28
29 def load_weight_from_coarse_segmentation(in_channels, model, weight_path, net_architecture):
30     """
31     it will change
32     """
33     in_channels = model.in_channels if hasattr(model, "in_channels") else in_channels
34     if net_architecture == 'UNet':
35         # d = torch.load(
36         #     './experiments/Graduate_project/multi_phase/pretrain/ResUnet/checkpoint/best_metric_model.pth')
37         d = torch.load(weight_path)
38         shape = list(d['model.0.conv.unit0.conv.weight'].shape)
39         shape[1] = in_channels - 1
40         if d.get('model.0.residual.weight') is not None:
41             noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['model.0.residual.weight'].device)
42             d['model.0.residual.weight'] = torch.cat((d['model.0.residual.weight'], noise), dim=1)
43             noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['model.0.conv.unit0.conv.weight'].device)
44             d['model.0.conv.unit0.conv.weight'] = torch.cat((d['model.0.conv.unit0.conv.weight'], noise), dim=1)
45         model.load_state_dict(d)
46         return model
47
48     elif net_architecture == "CSNet3D":
49         d1 = torch.load(weight_path)
50         d2 = model.state_dict()
51         print("-----load checkpoint and modify-----")
52         for k2, v2 in d2.items():
53             if d1[k2].shape != v2.shape:
54                 assert d1[k2].shape[1] != v2.shape[1] and len(d1[k2].shape) > 1 and len(v2.shape) > 1
55                 noise_shape = torch.tensor(v2.shape)
56                 noise_shape[1] = v2.shape[1] - d1[k2].shape[1]
57                 noise = nn.Parameter(Normal(0, 1e-7).sample(noise_shape)).to(d1[k2].device)
58                 print(
59                     f"key: {k2}, shape1: {d1[k2].shape}, mean1: {d1[k2].mean()}, shape2: {v2.shape}, mean2: {v2.mean()}")
60                 d1[k2] = torch.cat((d1[k2], noise), dim=1)
61         model.load_state_dict(d1)
62         return model
63
64     elif net_architecture == "SkipDenseUNet":
65         d1 = torch.load(weight_path)
66         d2 = model.state_dict()
67         print("-----load checkpoint and modify-----")
68         for k2, v2 in d2.items():
69             if d1[k2].shape != v2.shape:
70                 assert d1[k2].shape[1] != v2.shape[1] and len(d1[k2].shape) > 1 and len(v2.shape) > 1
71                 noise_shape = torch.tensor(v2.shape)
72                 noise_shape[1] = v2.shape[1] - d1[k2].shape[1]
73                 noise = nn.Parameter(Normal(0, 1e-10).sample(noise_shape)).to(d1[k2].device)
74                 print(
75                     f"key: {k2}, shape1: {d1[k2].shape}, mean1: {d1[k2].mean()}, shape2: {v2.shape}, mean2: {v2.mean()}")
76                 d1[k2] = torch.cat((d1[k2][:, :1, ...], noise, d1[k2][:, 1, ...]), dim=1)
77         model.load_state_dict(d1)
78         return model
79
80     elif net_architecture == "SwinTransformerSys3D":
81         d = torch.load(weight_path)
82         shape = list(d['patch_embed.proj.weight'].shape)
83         shape[1] = in_channels - 1
84
85         noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['patch_embed.proj.weight'].device)
86         d['patch_embed.proj.weight'] = torch.cat((d['patch_embed.proj.weight'], noise), dim=1)
87         model.load_state_dict(d)
88         return model
89
90     elif net_architecture == "nnunetv2":
91         all_information = torch.load(weight_path)
92         d = all_information['network weights']
93         shape = list(d['encoder.stages.0.0.convs.0.conv.weight'].shape)
94         shape[1] = in_channels - 1
95         noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['encoder.stages.0.0.convs.0.conv.weight'].device)
96         d['encoder.stages.0.0.convs.0.conv.weight'] = torch.cat((d['encoder.stages.0.0.convs.0.conv.weight'], noise), dim=1)
97         noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['decoder.encoder.stages.0.0.convs.0.conv.weight'].device)
98         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)
99         noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['decoder.encoder.stages.0.0.convs.0.all_modules.0.weight'].device)
100         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)
101         model.load_state_dict(d)
102         return model
103
104     else:
105         raise NotImplementedError(f"not support {net_architecture} yet!")
106
107
108 if __name__ == "__main__":
109
110     parser = argparse.ArgumentParser(description="cardio vessel segmentation")
111     parser.add_argument("--configs", dest="cfg",
112                         help="The configs file.",
113                         default='./configs/heart_server.yaml',
114                         type=str)
115     parser.add_argument("--iters", dest='iters', help='Iterations in training.', type=int, default=None)
116     parser.add_argument("--batch_size", dest='batch_size', help='Mini batch size of one gpu or cpu.', type=int, default=None)
117     parser.add_argument("--learning_rate", dest='learning_rate', help='Learning rate', type=float, default=None)
118     parser.add_argument("--seed", dest='seed', help='seed', type=float, default=0)
119     parser.add_argument("--mode", dest='mode', help='train, infer or both', type=str, default=None)
120
121     parser.add_argument("--experiments_path", dest='experiments_path',
122                         help='experiments_path, all output and checkpoint are saved here.', type=str, default=None)
123
124     parser.add_argument("--pretrain_weight_path", help='pretrain weight for training, testing of inferring', type=str, default=None)

```

```

125 parser.add_argument('--img_path', help='image path', type=str, default=None)
126
127 parser.add_argument('--label_path', help='label path', type=str, default=None)
128 parser.add_argument('--output_path', help='save path', type=str, default=None)
129 parser.add_argument('--persist_path', help='persist path, the path used to save persist data in persist loader',
130                     type=str, default=None)
131 parser.add_argument('--val_set', help='val set, a txt path to select val data, only used if mode is train',
132                     type=str, default=None)
133 parser.add_argument('--train_set', help='train set, a txt path to select train data, only used if mode is train',
134                     type=str, default=None)
135 parser.add_argument('--dataset_information', help='dataset information',
136                     type=str, default=None)
137
138 # ----- multi information ----- #
139 parser.add_argument('--CS_W',
140                     help='pretrain weight from coarse segmentation, testing of inferring', type=str, default=None)
141 parser.add_argument('--CS_M', help='coarse segmentation in main phase', type=str, default=None)
142 parser.add_argument('--CS_A', help='coarse segmentation in auxiliary phase', type=str, default=None)
143 parser.add_argument('--CS_DL', help='discontinuity label generated in main coarse segmentation', type=str,
144                     default=None)
145 parser.add_argument('--CS_DLGT', help='discontinuity label generated in auxiliary coarse segmentation', type=str,
146                     default=None)
147 parser.add_argument('--I_M', help='main image', type=str, default=None)
148 parser.add_argument('--I_A', help='auxiliary image', type=str, default=None)
149 parser.add_argument('--select_file', help='use discontinuity label to determine what data are used', type=str,
150                     default=None)
151 parser.add_argument('--delete_persist', help='pretrain weight from coarse segmentation, testing of inferring', type=str,)
152 parser.add_argument('--view', action='store_true', default=False, help='delete persist cache, because it is too large')
153 # ----- multi information ----- #
154
155 args = parser.parse_args()
156 monai.config.print_config()
157
158 logging.basicConfig(stream=sys.stdout, level=logging.INFO)
159 cfg = Config(
160     args.cfg,
161     learning_rate=args.learning_rate,
162     iters=args.iters,
163     batch_size=args.batch_size,
164     seed=args.seed,
165     mode=args.mode,
166     img_path=args.img_path,
167     I_M=args.I_M,
168     I_A=args.I_A,
169     CS_M=args.CS_M,
170     CS_A=args.CS_A,
171     CS_DL=args.CS_DL,
172     CS_DLGT=args.CS_DLGT,
173     CS_W=args.CS_W,
174     select_file=args.select_file,
175     label_path=args.label_path,
176     output_path=args.output_path,
177     persist_path=args.persist_path,
178     val_set=args.val_set,
179     train_set=args.train_set,
180     experiments_path=args.experiments_path,
181     pretrain_weight_path=args.pretrain_weight_path,
182     dataset_information=args.dataset_information
183 )
184 set_determinism(seed=cfg.seed)
185
186 if cfg.dic['mode'] == 'train':
187     # ----- built transform sequence ----- #
188     # ----- take a look at val_transform on dataset and save a case ----- #
189     cfg.creat_training_require()
190     load_weight_from_coarse_segmentation(in_channels=cfg.dic['model'].get("in_channels"), model=cfg.model,
191                                         net_architecture=cfg.dic['model']['name'], weight_path=cfg.CS_W)
192     train_transforms, val_transforms, save_transform = get_multi_phase_transform_with_image(cfg.dic['transform'])
193     if cfg.dic["train"]["loader"].get("file_path"):
194         files = [load_json(cfg.dic["train"]["loader"]["file_path"])]
195     elif cfg.dic["train"]["loader"].get("val_set"):
196         val_files = prepare_main_auxiliary_with_img_datalist_with_file(main_file=cfg.CS_M,
197                             auxiliary_file=cfg.CS_A,
198                             main_img_file=cfg.I_M,
199                             auxiliary_img_file=cfg.I_A,
200                             label_file=cfg.train_label_path,
201                             broken_file=cfg.CS_DL,
202                             broken_gt_file=cfg.CS_DLGT,
203                             img_name=cfg.val_set,
204                             select_file=cfg.select_file, )
205
206     train_files = prepare_main_auxiliary_with_img_datalist_with_file(main_file=cfg.CS_M,
207                             auxiliary_file=cfg.CS_A,
208                             main_img_file=cfg.I_M,
209                             auxiliary_img_file=cfg.I_A,
210                             label_file=cfg.train_label_path,
211                             broken_file=cfg.CS_DL,
212                             broken_gt_file=cfg.CS_DLGT,
213                             img_name=cfg.train_set,
214                             select_file=cfg.select_file, )
215
216     files = [{"train_files": train_files, "val_files": val_files}]
217
218 else:
219     raise ValueError("please provide a file path or val set and train set")
220 # check_transform_in_data_loader(val_files=files[0]['val_files'], val_transforms=try_transforms)
221
222 # ----- create loss function ----- #
223 # ----- you can crate your own loss or metric here amd replace cfg ----- #
224
225 # dice_metric = DiceMetric(include_background=False, reduction="mean")
226 # loss_function = DiceLoss(to_onehot_y=True, softmax=True)
227
228 # ----- training ----- #
229 metric_record = []
230 experiment_path = os.path.join(cfg.dic['experiments_path'], time.strftime("%d_%m_%Y_%H_%M_%S"))\
231     if cfg.dic.get('time_name', None) else cfg.dic['experiments_path']
232
233 if not os.path.exists(experiment_path):
234     os.makedirs(experiment_path)
235 for i in range(len(files)):
236     if cfg.dic['train']['loader'].get('split_mode') == "five_fold":
237         experiment_path_fold = os.path.join(experiment_path, f"{i}_fold")
238         if not os.path.exists(experiment_path_fold):
239             os.makedirs(experiment_path_fold)
240         else:
241             experiment_path_fold = experiment_path
242         write_data_reference(files[i], experiment_path_fold)
243         save_json(files[i], os.path.join(experiment_path_fold, 'files.txt'))
244         # ----- Create Model, Loss, Optimizer in Config----- #
245         # ----- using config ----- #
246         cfg.save_config(os.path.join(experiment_path_fold, 'configs.yaml'))
247         if cfg.dic['train']['loader'].get('split_mode') == "five_fold":
248             print(f"-----fold{i} start!-----")
249         else:
250             print(f"-----training start!-----")
251         if cfg.dic['train']['loader'].get('persist'):

```



```

252         if cfg.persist_path == 'default':
253             persistent_cache = pathlib.Path(experiment_path_fold, "persistent_cache")
254         else:
255             persistent_cache = pathlib.Path(cfg.persist_path)
256
257         persistent_cache.mkdir(parents=True, exist_ok=True)
258         train_ds = PersistentDataset(data=files[i]['train_files'], transform=train_transforms,
259                                     cache_dir=persistent_cache)
260         val_ds = PersistentDataset(data=files[i]['val_files'], transform=val_transforms,
261                                   cache_dir=persistent_cache)
262     else:
263         train_ds = CacheDataset(
264             data=files[i]['train_files'], transform=train_transforms,
265             cache_rate=cfg.dic['train']['loader']['cache'], num_workers=4)
266         # train_ds = Dataset(data=train_files, transform=train_transforms)
267         # use batch_size=2 to load images and use RandCropByPosNegLabeld
268         val_ds = CacheDataset(
269             data=files[i]['val_files'], transform=val_transforms,
270             cache_rate=cfg.dic['train']['loader']['cache'], num_workers=4)
271
272     # load pretrained model
273     out_channels = cfg.dic['model']['out_channels'] if cfg.dic['model'].get('out_channels', None) else cfg.model.out_channels
274     cardio_vessel_segmentation_multi_phase_with_image_train(cfg=cfg,
275                                                             key=cfg.second_stage_key,
276                                                             model=cfg.model,
277                                                             num_class=out_channels,
278                                                             loss_function=cfg.train_loss,
279                                                             val_metric=cfg.val_metric,
280                                                             optimizer=cfg.optimizer_init,
281                                                             lr_scheduler=cfg.lr_scheduler_init,
282                                                             train_dataset=train_ds,
283                                                             val_dataset=val_ds,
284                                                             experiment_path=experiment_path_fold,
285                                                             device=cfg.device,
286                                                             start_epoch=cfg.start_epoch,
287                                                             sw_batch_size=cfg.train_sw_batch_size,
288                                                             overlap=cfg.train_sw_overlap,
289                                                             mirror_axes=cfg.train_mirror_axes,
290                                                             )
291     if cfg.dic['train']['loader'].get('persist'):
292         import shutil
293         shutil.rmtree(persistent_cache)
294
295 elif cfg.dic['mode'] == 'test':
296     train_transforms, val_transforms, save_transform = get_multi_phase_transform_with_image(cfg.dic['transform'])
297
298     cfg.creat_test_require()
299     cfg.save_config(os.path.join(cfg.test_output_path, 'config.yaml'))
300     if cfg.dic['test']['loader'].get("file_path"):
301         files = load_json(cfg.dic['test']['loader']["file_path"])["val_files"]
302     elif cfg.dic['test']['loader'].get("val_set"):
303         files = prepare_main_auxiliary_with_img_datalist_with_file(main_file=cfg.CS_M,
304                                                                     auxiliary_file=cfg.CS_A,
305                                                                     main_img_file=cfg.I_M,
306                                                                     auxiliary_img_file=cfg.I_A,
307                                                                     label_file=cfg.test_label_path,
308                                                                     broken_file=cfg.CS_DL,
309                                                                     broken_gt_file=cfg.CS_DLGT,
310                                                                     img_name=cfg.val_set,
311                                                                     select_file=cfg.select_file, )
312     else:
313         files = prepare_main_auxiliary_with_img_datalist_with_file(main_file=cfg.CS_M,
314                                                                     auxiliary_file=cfg.CS_A,
315                                                                     main_img_file=cfg.I_M,
316                                                                     auxiliary_img_file=cfg.I_A,
317                                                                     label_file=cfg.test_label_path,
318                                                                     broken_file=cfg.CS_DL,
319                                                                     broken_gt_file=cfg.CS_DLGT,
320                                                                     img_name=None,
321                                                                     select_file=None, )
322     # ----- built transform sequence ----- #
323     total_ds = CacheDataset(
324         data=files, transform=val_transforms, cache_rate=cfg.dic['test']['loader']['cache'], num_workers=2)
325
326     # ----- test ----- #
327     cardio_vessel_segmentation_multi_phase_with_image_test(model=cfg.model,
328                                                             key=['I_M', 'CS_M', 'CS_A', 'CS_DLGT'],
329                                                             val_dataset=total_ds,
330                                                             device=cfg.device,
331                                                             output_path=cfg.test_output_path,
332                                                             window_size=cfg.dic['transform']['patch_size'],
333                                                             save_data=cfg.dic['test']['save_data'])
334
335 elif cfg.dic['mode'] == 'infer':
336     infer_transforms = get_multi_phase_transform_with_image(cfg.dic['transform'], mode='infer')
337
338     cfg.creat_infer_require()
339     cfg.save_config(os.path.join(cfg.infer_output_path, 'config.yaml'))
340
341     # ----- built transform sequence ----- #
342     # files = prepare_image_list(image_path=cfg.dic['infer']['loader']['path'])
343
344     files = prepare_main_auxiliary_with_img_datalist_with_file(main_file=cfg.CS_M,
345                                                                 auxiliary_file=cfg.CS_A,
346                                                                 main_img_file=cfg.I_M,
347                                                                 auxiliary_img_file=cfg.I_A,
348                                                                 label_file=None,
349                                                                 broken_file=cfg.CS_DL,
350                                                                 broken_gt_file=cfg.CS_DLGT,
351                                                                 img_name=None,
352                                                                 select_file=None, )
353     total_ds = CacheDataset(
354         data=files, transform=infer_transforms, cache_rate=cfg.dic['infer']['loader']['cache'], num_workers=2)
355
356     cardio_vessel_segmentation_multi_phase_with_image_infer(model=cfg.model,
357                                                             key=cfg.second_stage_key,
358                                                             val_dataset=total_ds,
359                                                             device=cfg.device,
360                                                             output_path=cfg.infer_output_path,
361                                                             window_size=tuple(cfg.dic['transform']['patch_size']),
362                                                             origin_transforms=infer_transforms,
363                                                             overlap=cfg.infer_sw_overlap,
364                                                             sw_batch_size=cfg.infer_sw_batch_size,
365                                                             mirror_axes=cfg.infer_mirror_axes,
366                                                             )
367 else:
368     raise RuntimeError('Only train and infer mode are supported now')

```

```

1 SparrowLink/second_stage_only_one_phase_main.py
2 from monai.utils import first, set_determinism
3 from monai.data import CacheDataset, DataLoader, Dataset, decollate_batch, ITKReader, PersistentDataset
4 from monai.config import print_config
5 from monai.metrics.meandice import DiceMetric
6 import torch
7 import matplotlib.pyplot as plt
8 import os
9 import argparse
10 import monai
11 import logging
12 from utils.Config import Config
13 import sys
14 import time
15 from data_loader import prepare_datalist, prepare_datalist_with_file, \
16     prepare_image_list, save_json, write_data_reference, load_json, prepare_main_with_img_datalist_with_file
17 from transform_utils import get_transform, get_second_stage_only_one_phase
18 from utils.test import cardio_vessel_segmentation_test, cardio_vessel_segmentation_multi_phase_with_image_test
19 from utils.trainer import cardio_vessel_segmentation_multi_phase_with_image_train
20 from utils.inferer import cardio_vessel_segmentation_infer, cardio_vessel_segmentation_multi_phase_with_image_infer
21 import pathlib
22 import torch.nn as nn
23 from torch.distributions.normal import Normal
24 # print_config()
25
26 torch.multiprocessing.set_sharing_strategy('file_system')
27
28
29 def load_weight_from_coarse_segmentation(in_channels, model, weight_path, net_architecture):
30     """
31     it will change
32     """
33     in_channels = model.in_channels if hasattr(model, "in_channels") else in_channels
34     if net_architecture == 'UNet':
35         # d = torch.load(
36         #     './experiments/Graduate_project/multi_phase/pretrain/ResUnet/checkpoint/best_metric_model.pth')
37         d = torch.load(weight_path)
38         shape = list(d['model.0.conv.unit0.conv.weight'].shape)
39         shape[1] = in_channels - 1
40         if d.get('model.0.residual.weight'):
41             noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['model.0.residual.weight'].device)
42             d['model.0.residual.weight'] = torch.cat((d['model.0.residual.weight'], noise), dim=1)
43             noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['model.0.conv.unit0.conv.weight'].device)
44             d['model.0.conv.unit0.conv.weight'] = torch.cat((d['model.0.conv.unit0.conv.weight'], noise), dim=1)
45             model.load_state_dict(d)
46
47     elif net_architecture == "CS2net":
48         d = torch.load(weight_path)
49         shape = list(d['encoder1.conv1.weight'].shape)
50         shape[1] = in_channels - 1
51         noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['enc_input.conv1.weight'].device)
52         d['enc_input.conv1.weight'] = torch.cat((d['enc_input.conv1.weight'], noise), dim=1)
53         model.load_state_dict(d)
54
55     elif net_architecture == "SkipDenseUnet":
56         d = torch.load(weight_path)
57         shape = list(d['features.conv0.weight'].shape)
58         shape[1] = in_channels - 1
59         noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['features.conv0.weight'].device)
60         d['features.conv0.weight'] = torch.cat((d['features.conv0.weight'], noise), dim=1)
61         model.load_state_dict(d)
62
63     elif net_architecture == "SwinTransformerSys3D":
64         d = torch.load(weight_path)
65         shape = list(d['patch_embed.proj.weight'].shape)
66         shape[1] = in_channels - 1
67         noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['patch_embed.proj.weight'].device)
68         d['patch_embed.proj.weight'] = torch.cat((d['patch_embed.proj.weight'], noise), dim=1)
69         model.load_state_dict(d)
70
71     elif net_architecture == "nnunetv2":
72         all_information = torch.load(weight_path)
73         d = all_information['network_weights']
74         shape = list(d['encoder.stages.0.0.convs.0.conv.weight'].shape)
75         shape[1] = in_channels - 1
76         noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['encoder.stages.0.0.convs.0.conv.weight'].device)
77         d['encoder.stages.0.0.convs.0.conv.weight'] = torch.cat((d['encoder.stages.0.0.convs.0.conv.weight'], noise), dim=1)
78         noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['encoder.stages.0.0.convs.0.all_modules.0.weight'].device)
79         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)
80         noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['decoder.encoder.stages.0.0.convs.0.conv.weight'].device)
81         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)
82         noise = nn.Parameter(Normal(0, 1e-5).sample(shape)).to(d['decoder.encoder.stages.0.0.convs.0.all_modules.0.weight'].device)
83         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)
84         model.load_state_dict(d)
85     return model
86
87
88 if __name__ == "__main__":
89
90     parser = argparse.ArgumentParser(description="cardio vessel segmentation")
91     parser.add_argument("--configs", dest="cfg",
92                         help="The configs file.",
93                         default='./configs/heart_server.yaml',
94                         type=str)
95     parser.add_argument("--iters", dest='iters', help='Iterations in training.', type=int, default=None)
96     parser.add_argument("--batch_size", dest='batch_size', help='Mini batch size of one gpu or cpu.', type=int, default=None)
97     parser.add_argument("--learning_rate", dest='learning_rate', help='Learning rate', type=float, default=None)
98     parser.add_argument("--seed", dest='seed', help='seed', type=float, default=0)
99     parser.add_argument("--mode", dest='mode', help='train, infer or both', type=str, default=None)
100
101     parser.add_argument("--experiments_path", dest='experiments_path',
102                        help='experiments_path, all output and checkpoint are saved here.', type=str, default=None)
103
104     parser.add_argument("--pretrain_weight_path", help='pretrain weight for training, testing of inferring', type=str, default=None)
105     parser.add_argument("--img_path", help='image path', type=str, default=None)
106
107     parser.add_argument("--label_path", help='label path', type=str, default=None)
108     parser.add_argument("--output_path", help='save path', type=str, default=None)
109     parser.add_argument("--persist_path", help='persist path, the path used to save persist data in persist loader',
110                        type=str, default=None)
111     parser.add_argument("--val_set", help='val set, a txt path to select val data, only used if mode is train',
112                        type=str, default=None)
113     parser.add_argument("--train_set", help='train set, a txt path to select train data, only used if mode is train',
114                        type=str, default=None)
115     parser.add_argument("--dataset_information", help='dataset information',
116                        type=str, default=None)
117
118     # ----- multi information ----- #
119     parser.add_argument("--CS_W",
120                        help='pretrain weight from coarse segmentation, testing of inferring', type=str, default=None)
121     parser.add_argument("--CS_M", help='coarse segmentation in main phase', type=str, default=None)
122     parser.add_argument("--CS_A", help='coarse segmentation in auxiliary phase', type=str, default=None)
123     parser.add_argument("--CS_DL", help='discontinuity label generated in main coarse segmentation', type=str,
124                        default=None)

```

```

126 parser.add_argument('--CS-DLGT', help='discontinuity label generated in auxiliary coarse segmentation', type=str,  
    default=None)  
127  
128 parser.add_argument('--I M', help='main image', type=str, default=None)  
129 parser.add_argument('--I A', help='auxiliary image', type=str, default=None)  
130 parser.add_argument('--select file', help='use discontinuity label to determine what data are used', type=str,  
    default=None)  
131  
132 parser.add_argument('--delete_persist', help='preload weight from coarse segmentation, testing of inferring', type=str,  
    action='store_true', default=False, help='delete persist cache, because it is too large')  
133 # ----- multi information ----- #  
134  
135 args = parser.parse_args()  
136 monai.config.print_config()  
137  
138 logging.basicConfig(stream=sys.stdout, level=logging.INFO)  
139 cfg = Config(  
140     args.cfg,  
141     learning_rate=args.learning_rate,  
142     iters=args.iters,  
143     batch_size=args.batch_size,  
144     seed=args.seed,  
145     mode=args.mode,  
146     img_path=args.img_path,  
147     I_M=args.I_M,  
148     I_A=args.I_A,  
149     CS_M=args.CS_M,  
150     CS_A=args.CS_A,  
151     CS_DL=args.CS_DL,  
152     CS_DLGT=args.CS_DLGT,  
153     CS_W=args.CS_W,  
154     select_file=args.select_file,  
155     label_path=args.label_path,  
156     output_path=args.output_path,  
157     persist_path=args.persist_path,  
158     val_set=args.val_set,  
159     train_set=args.train_set,  
160     experiments_path=args.experiments_path,  
161     pretrain_weight_path=args.pretrain_weight_path,  
162     dataset_information=args.dataset_information  
163 )  
164 set_determinism(seed=cfg.seed)  
165  
166 if cfg.dic['mode'] == 'train':  
167  
168     # ----- built transform sequence ----- #  
169     # ----- take a look at val_transform on dataset and save a case ----- #  
170     cfg.create_training_require()  
171     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)  
172     train_transforms, val_transforms, save_transform = get_second_stage_only_one_phase(cfg.dic['transform'])  
173     if cfg.dic['train']['loader'].get("file path"):  
174         files = [load_json(cfg.dic["train"]["loader"] ["file path"]) ]  
175     elif cfg.dic["train"] ["loader"].get ("val set"):   
176         val_files = prepare_main_with_img_dataalist_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, )  
  
177         train_files = prepare_main_with_img_dataalist_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, )  
178         files = [{"train_files": train_files, "val_files": val_files}]  
179  
180 else:  
181     raise ValueError("please provide a file path or val set and train set")  
182     # check_transform_in_dateloader(val_files=files[0] ['val_files'], val_transforms=train_transforms)  
183  
184     # ----- create loss function ----- #  
185     # ----- you can crate your own loss or metric here amd replace cfg ----- #  
186  
187     dice_metric = DiceMetric(include_background=False, reduction="mean")  
188     loss_function = DiceLoss(to_ohot_y=True, softmax=True)  
189  
190     # ----- training ----- #  
191     metric_record = []  
192     experiment_path = os.path.join(cfg.dic['experiments path'], time.strftime("%d_%m_%Y_%H%M%S")) \   
193         if cfg.dic.get ('time_name', None) else cfg.dic['experiments path']  
194  
195     if not os.path.exists(experiment_path):  
196         os.makedirs(experiment_path)  
197     for i in range(len(files)):  
198         if cfg.dic['train'] ['loader'].get ('split_mode') == "five_fold":  
199             experiment_path_fold = os.path.join(experiment_path, f"{i}_fold")  
200             if not os.path.exists(experiment_path_fold):  
201                 os.makedirs(experiment_path_fold)  
202         else:  
203             experiment_path_fold = experiment_path  
204             write_data_reference(files[i], experiment_path_fold)  
205             save_json(files[i], os.path.join(experiment_path_fold, 'files.txt'))  
206             # ----- Create Model, Loss, Optimizer in Config----- #  
207             # ----- using config ----- #  
208             cfg.save_config(os.path.join(experiment_path_fold, 'configs.yaml'))  
209             if cfg.dic['train'] ['loader'].get ('split_mode') == "five_fold":  
210                 print(f"-----fold{i} start!-----")  
211             else:  
212                 print(f"-----training start!-----")  
213             if cfg.dic['train'] ['loader'].get ('persist'):   
214                 if cfg.persist_path == 'default':  
215                     persistent_cache = pathlib.Path(experiment_path_fold, "persistent_cache")  
216                 else:  
217                     persistent_cache = pathlib.Path(cfg.persist_path)  
218             persistent_cache.mkdir(parents=True, exist_ok=True)  
219             train_ds = PersistentDataset(data=files[i] ['train_files'], transform=train_transforms,  
                cache_dir=persistent_cache)  
220             val_ds = PersistentDataset(data=files[i] ['val_files'], transform=val_transforms,  
                cache_dir=persistent_cache)  
221         else:  
222             train_ds = CacheDataset(  
223                 data=files[i] ['train_files'], transform=train transforms,  
224                 cache_rate=cfg.dic['train'] ['loader'] ['cache'], num_workers=4)  
225             # train ds = Dataset(data=train_files, transform=train_transforms)  
226             # use batch size=2 to load images and use RandCropByPosNegLabeld  
227             val_ds = CacheDataset(  
228                 data=files[i] ['val_files'], transform=val transforms,  
229                 cache_rate=cfg.dic['train'] ['loader'] ['cache'], num_workers=4)  
230  
231     # load pretrained model  
232     out_channels = cfg.dic['model'] ['out_channels'] if cfg.dic['model'].get ('out_channels', None) else cfg.model.out_channels  
233     cardio_vessel_segmenatation_multi_phase_with_image_train(cfg=cfg,  
234         key=cg.second stage key,
```

```

252         model=cfg.model,
253         num_class=out_channels,
254         loss_function=cfg.train_loss,
255         val_metric=cfg.val_metric,
256         optimizer=cfg.optimizer_init,
257         lr_scheduler=cfg.lr_scheduler_init,
258         train_dataset=train_ds,
259         val_dataset=val_ds,
260         experiment_path=experiment_path_fold,
261         device=cfg.device,
262         start_epoch=cfg.start_epoch,
263         sw_batch_size=cfg.train_sw_batch_size,
264         overlap=cfg.train_sw_overlap,
265         mirror_axes=cfg.train_mirror_axes,
266     )
267     if cfg.dic['train']['loader'].get('persist'):
268         import shutil
269         shutil.rmtree(persistent_cache)
270
271 elif cfg.dic['mode'] == 'test':
272     train_transforms, val_transforms, save_transform = get_second_stage_only_one_phase(cfg.dic['transform'])
273
274     cfg.creat_test_require()
275     cfg.save_config(os.path.join(cfg.test_output_path, 'config.yaml'))
276     if cfg.dic["test"]["loader"].get("file path"):
277         files = load_json(cfg.dic["test"]["loader"]["file path"])["val_files"]
278     elif cfg.dic["test"]["loader"].get("val set"):
279         files = prepare_main_with_img_datalist_with_file(main_file=cfg.CS_M,
280                                                         main_img_file=cfg.I_M,
281                                                         label_file=cfg.test_label_path,
282                                                         broken_file=cfg.CS_DL,
283                                                         broken_gt_file=cfg.CS_DLGT,
284                                                         img_name=cfg.val_set,
285                                                         select_file=cfg.select_file, )
286     else:
287         files = prepare_main_with_img_datalist_with_file(main_file=cfg.CS_M,
288                                                         main_img_file=cfg.I_M,
289                                                         label_file=cfg.test_label_path,
290                                                         broken_file=cfg.CS_DL,
291                                                         broken_gt_file=cfg.CS_DLGT,
292                                                         img_name=None,
293                                                         select_file=None, )
294     # ----- built transform sequence ----- #
295     total_ds = CacheDataset(
296         data=files, transform=val_transforms, cache_rate=cfg.dic['test']['loader']['cache'], num_workers=2)
297
298     # ----- test ----- #
299     cardio_vessel_segmentation_multi_phase_with_image_test(model=cfg.model,
300                                                            key=['I_M', 'CS_M', 'CS_A', 'CS_DLGT'],
301                                                            val_dataset=total_ds,
302                                                            device=cfg.device,
303                                                            output_path=cfg.test_output_path,
304                                                            window_size=cfg.dic['transform']['patch_size'],
305                                                            save_data=cfg.dic['test']['save_data'])
306
307 elif cfg.dic['mode'] == 'infer':
308     infer_transforms = get_second_stage_only_one_phase(cfg.dic['transform'], mode='infer')
309
310     cfg.creat_infer_require()
311     cfg.save_config(os.path.join(cfg.infer_output_path, 'config.yaml'))
312
313     # ----- built transform sequence ----- #
314     # files = prepare_image_list(image_path=cfg.dic['infer']['loader']['path'])
315
316     files = prepare_main_with_img_datalist_with_file(main_file=cfg.CS_M,
317                                                         main_img_file=cfg.I_M,
318                                                         label_file=None,
319                                                         broken_file=cfg.CS_DL,
320                                                         broken_gt_file=cfg.CS_DLGT,
321                                                         img_name=None,
322                                                         select_file=None, )
323
324     total_ds = CacheDataset(
325         data=files, transform=infer_transforms, cache_rate=cfg.dic['infer']['loader']['cache'], num_workers=2)
326
327     cardio_vessel_segmentation_multi_phase_with_image_infer(model=cfg.model,
328                                                             key=cfg.second_stage_key,
329                                                             val_dataset=total_ds,
330                                                             device=cfg.device,
331                                                             output_path=cfg.infer_output_path,
332                                                             window_size=tuple(cfg.dic['transform']['patch_size']),
333                                                             origin_transforms=infer_transforms,
334                                                             overlap=cfg.infer_sw_overlap,
335                                                             sw_batch_size=cfg.infer_sw_batch_size,
336                                                             mirror_axes=cfg.infer_mirror_axes
337                 )
338 else:
339     raise RuntimeError('Only train and infer mode are supported now')

```

```

0 SparrowLink/slicer_visualization/slicer_mark_up.py
1 import json
2
3
4 def create_extended_plane_markup_json(center, normal, bounds, coordinate_system="LPS", plane_type="pointNormal",
5                                     size_mode="absolute", auto_scaling_factor=1.0, markup_orientation=None,
6                                     orientation=None, object_to_base=None, base_to_node=None, color=[0, 0, 1]):
7     """
8     Create an extended JSON string for a plane markup in 3D Slicer, including various properties.
9
10    Parameters:
11    - center: The center of the plane in world coordinates.
12    - normal: The normal vector of the plane.
13    - size: The size of the plane (length, width, height).
14    - bounds: The bounds of the plane in world coordinates.
15    - coordinate_system: The coordinate system used (default "LPS").
16    - plane_type: The type of the plane (default "pointNormal").
17    - size_mode: The size mode of the plane (default "absolute").
18    - auto_scaling_factor: Auto scaling factor for the plane size (default 1.0).
19    - orientation: The orientation matrix of the plane (3x3 matrix).
20    - object_to_base: The object to base transform matrix (4x4 matrix).
21    - base_to_node: The base to node transform matrix (4x4 matrix).
22
23    Returns:
24    - A JSON string representing the extended markup for 3D Slicer.
25    """
26    # Default values for orientation and transform matrices if not provided
27    if orientation is None:
28        orientation = [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
29    if object_to_base is None:
30        object_to_base = orientation[:3] + [0.0] + orientation[3:6] + [0.0] + orientation[6:] + [0.0, 0.0, 0.0, 0.0, 1.0]
31    if base_to_node is None:
32        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]
33
34    # calculate size
35    length = bounds[1] - bounds[0] # xMax - xMin
36    width = bounds[3] - bounds[2] # yMax - yMin
37    size = [length, width, 0.0]
38
39    markup_json = {
40        "@schema": "https://raw.githubusercontent.com/slicer/slicer/master/Modules/Loadable/Markups/Resources/Schema/markups-schema-v1.0.3.json#",
41        "markups": [
42            {
43                "type": "Plane",
44                "coordinateSystem": coordinate_system,
45                "coordinateUnits": "mm",
46                "locked": False,
47                "fixedNumberOfControlPoints": False,
48                "labelFormat": "%N-%d",
49                "lastUsedControlPointNumber": 1,
50                "planeType": plane_type,
51                "sizeMode": size_mode,
52                "autoScalingFactor": auto_scaling_factor,
53                "center": center,
54                "normal": normal, # "objectToBase": object_to_base,
55                "baseToNode": base_to_node,
56                "orientation": markup_orientation,
57                "size": size,
58                "planeBounds": bounds,
59                "display":
60                    {
61                        "color": color,
62                    },
63                "controlPoints": [
64                    {
65                        "id": "1",
66                        "label": "P-1",
67                        "position": center,
68                        "orientation": orientation,
69                        "selected": True,
70                        "locked": False,
71                        "visibility": True,
72                        "positionStatus": "defined"
73                    }
74                ],
75                "measurements": [
76                    {
77                        "name": "area",
78                        "enabled": False,
79                        "units": "cm2",
80                        "printFormat": "%-#4.4g%s"
81                    }
82                ]
83            }
84        ]
85    }
86
87    return json.dumps(markup_json, indent=4)

```

```

0 SparrowLink/slicer_visulization/slicer_mvf.py
1 import nrrd
2 import numpy as np
3
4
5 def get_space_full_name(space):
6     assert len(space) == 3, "space should be a 3-letter string"
7     full_name = []
8     for s in space:
9         if s == "L":
10             full_name.append("left")
11         elif s == "R":
12             full_name.append("right")
13         elif s == "A":
14             full_name.append("anterior")
15         elif s == "P":
16             full_name.append("posterior")
17         elif s == "S":
18             full_name.append("superior")
19         elif s == "I":
20             full_name.append("inferior")
21     # link with _ to form full name
22     return "-".join(full_name)
23
24
25 def get_direction(space):
26     """3D-slicer use default RAS space, so we need to convert the space to RAS"""
27     direction = [1, 1, 1]
28     if space[0] == "L":
29         direction[0] = -1
30     if space[1] == "P":
31         direction[1] = -1
32     if space[2] == "I":
33         direction[2] = -1
34     return direction
35
36
37 def save_mvf(mvf, save_path, affine, space, scale_factor=10):
38     offset = np.array((affine[:3, 3]))
39     direction = np.array((affine[:3, :3])).tolist()
40     """Seems like mvf in 3D slicer do not consider the direction of the space,
41     so we need to multiply the mvf with the direction matrix to make it consistent with the space"""
42     mvf_direction = get_direction(space)
43     mvf[:, :, :, 0] = mvf[:, :, :, 0] * mvf_direction[0]
44     mvf[:, :, :, 1] = mvf[:, :, :, 1] * mvf_direction[1]
45     mvf[:, :, :, 2] = mvf[:, :, :, 2] * mvf_direction[2]
46     header = {
47         'endian': 'little',
48         'encoding': 'raw',
49         'space': get_space_full_name(space),
50         'space directions': direction + [None],
51         'space origin': offset,
52         'kinds': ['domain', 'domain', 'domain', 'vector'],
53     }
54     nrrd.write(save_path, mvf * scale_factor, header=header)
55
56
57 def save_image(image, save_path, affine, space):
58     offset = np.array((affine[:3, 3]))
59     direction = np.array((affine[:3, :3])).tolist()
60     header = {
61         'endian': 'little',
62         'encoding': 'raw',
63         'space': get_space_full_name(space),
64         'space directions': direction,
65         'space origin': offset,
66         'kinds': ['domain', 'domain', 'domain'],
67     }
68     nrrd.write(save_path, image, header=header)

```

```

0 SparrowLink/SparrowLink_metric.py
1 from post_processing.fracture_detection import sparrowlink_metric
2 import numpy as np
3 import SimpleITK as sitk
4 import json
5 from openpyxl import Workbook
6
7 class SparrowLinkMetric(object):
8     def __init__(self,
9                 max_broken_size_percentage=0.1,
10                 min_sphere_radius_percentage=0.02,
11                 sphere_dilation_percentage=0.01,
12                 region_select=4,
13                 cube_min_size=(10, 10, 10),
14                 skeleton_refine_times=3,
15                 region_threshold=5,
16                 angle_threshold_1=0.0,
17                 angle_threshold_2=0.0,
18                 view=False
19             ):
20         self.max_broken_size_percentage = max_broken_size_percentage
21         self.min_sphere_radius_percentage = min_sphere_radius_percentage
22         self.sphere_dilation_percentage = sphere_dilation_percentage
23         self.region_select = region_select
24         self.cube_min_size = cube_min_size
25         self.skeleton_refine_times = skeleton_refine_times
26         self.region_threshold = region_threshold
27         self.angle_threshold_1 = angle_threshold_1
28         self.angle_threshold_2 = angle_threshold_2
29         self.view = view
30
31         """
32         :param max_broken_size_percentage: the max broken size percentage, used in constriction on distance
33         :param min_sphere_radius_percentage: the min sphere radius percentage, used for some small fracture
34         :param sphere_dilation_percentage: the sphere dilation percentage, to consider the structure around fracture area
35         :param region_select: the region select, matching points consist of start point and paired point, start point is...
36         ...constricted in the region with region_select.
37         :param cube_min_size: the min size of the cube, used for small fracture
38         :param skeleton_refine_times: the refine times of the skeleton, zhang algorithm might cause some small branch in the...
39         ...skeleton, so we need to refine the skeleton to delete the small branch
40         :param region_threshold: the region threshold, used to delete the small region in the segmentation
41         :param angle_threshold_1: the angle threshold 1, used for constriction on orientation
42         :param angle_threshold_2: the angle threshold 2, used for constriction on orientation
43         """
44     def __call__(self,
45                 seg_path: str = None,
46                 gt_path: str = None,
47                 save_path: str = None
48             ):
49         seg, spacing, origin, direction = self.read_image(seg_path)
50         self.spacing, self.origin, self.direction = spacing, origin, direction
51         gt, _, _ = self.read_image(gt_path)
52         result_dict = sparrowlink_metric(
53             label=seg,
54             GT=gt,
55             spacing=spacing,
56             direction=direction,
57             origin=origin,
58             save_path=save_path,
59             view=self.view,
60             max_broken_size_percentage=self.max_broken_size_percentage,
61             min_sphere_radius_percentage=self.min_sphere_radius_percentage,
62             sphere_dilation_percentage=self.sphere_dilation_percentage,
63             cube_min_size=self.cube_min_size,
64             skeleton_refine_times=self.skeleton_refine_times,
65             region_threshold=self.region_threshold,
66             angle_threshold_1=self.angle_threshold_1,
67             angle_threshold_2=self.angle_threshold_2,
68         )
69         d = {
70             "name": pathlib.Path(seg_path).name[:9],
71             "num_gt": result_dict["num_gt"],
72         }
73         ##### permanent #####
74         if "_CS_M" in str(seg_path):
75             print(1)
76             sphere_gt = result_dict["mask_sphere_gt"]
77             gt_segment = ((sphere_gt * gt) > 0).astype(np.uint16)
78             self.save_image(gt_segment, save_path=seg_path.replace(".nii.gz", "_sphere_gt_segment.nii.gz"))
79             self.save_image(sphere_gt, save_path=seg_path.replace(".nii.gz", "_sphere_gt.nii.gz"))
80             path = seg_path.replace("_CS_M.nii.gz", "_RCS_NEW.nii.gz")
81             rcs_new = self.read_image(path)[0]
82             rcs_new_gt_sphere_segment = ((sphere_gt * rcs_new) > 0).astype(np.uint16)
83
84             self.save_image(rcs_new_gt_sphere_segment,
85                             save_path=seg_path.replace(".nii.gz", "_rcs_new_gt_sphere_segment.nii.gz"))
86             cs_m_gt_sphere_segment = ((sphere_gt * seg) > 0).astype(np.uint16)
87             self.save_image(cs_m_gt_sphere_segment,
88                             save_path=seg_path.replace(".nii.gz", "_CS_M_gt_sphere_segment.nii.gz"))
89
90         return d
91
92     def save_image(self, image, save_path, save_type=np.uint16):
93         sitk_image = sitk.GetImageFromArray(image.astype(save_type))
94         sitk_image.SetSpacing(self.spacing)
95         sitk_image.SetOrigin(self.origin)
96         sitk_image.SetDirection(self.direction)
97         sitk_image.WriteImage(sitk_image, save_path)
98
99     @staticmethod

```

```

98     def read_image(path):
99         sitk_image = sitk.ReadImage(path)
100         image = sitk.GetArrayFromImage(sitk_image)
101         spacing = sitk_image.GetSpacing()
102         origin = sitk_image.GetOrigin()
103         direction = sitk_image.GetDirection()
104         return image, spacing, origin, direction
105
106
107     def update(pbar, record, result):
108         pbar.update()
109         record.append(result)
110
111
112     def error_back(err):
113         print(err)
114
115
116     if __name__ == "__main__":
117         import argparse
118         import tqdm
119         import pathlib
120         from multiprocessing import Pool
121
122         import warnings
123         warnings.filterwarnings("ignore") # ignore from np.int16 to np.uint8
124
125         parser = argparse.ArgumentParser()
126         parser.add_argument("--seg", type=str, default=None)
127         parser.add_argument("--seg_find", type=str, default="*.nii.gz")
128         parser.add_argument("--gt", type=str, default=None)
129         parser.add_argument("--gt_find", type=str, default="*.nii.gz")
130         parser.add_argument("--multiprocess", action='store_true', default=False)
131         parser.add_argument("--metric_postfix", type=str, default=None)
132         args = parser.parse_args()
133
134         assert args.seg is not None, "seg_path is None"
135         assert args.gt is not None, "label_path is None"
136         if args.metric_postfix is None:
137             metric_result_path = str(pathlib.Path(args.seg) / "sparrowlink_metric.xlsx")
138         else:
139             metric_result_path = str(pathlib.Path(args.seg) / f"sparrowlink_metric_{args.metric_postfix}.xlsx")
140
141         wb = Workbook()
142         ws = wb.active
143         ws.append(['name', 'num_gt'])
144         metric_list = []
145
146         seg = pathlib.Path(args.seg)
147         gt = pathlib.Path(args.gt)
148         seg_list = list(seg.glob(f"{args.seg_find}"))
149         gt_list = list(gt.glob(f"{args.gt_find}"))
150         seg_list.sort()
151         gt_list.sort()
152         print(f"\033[96m SparrowLink Metric Calculating for {args.seg_find} \033[00m")
153         pbar = tqdm.tqdm(total=len(seg_list), colour="#87cefa")
154         pbar.set_description("SparrowLink Processing")
155         metric_record = []
156         pool = Pool(14)
157         for seg_path, gt_path in zip(seg_list, gt_list):
158             name = seg_path.name[:9]
159             metric_calculator = SparrowLinkMetric(view=False)
160             if not args.multiprocess:
161                 result_dict = metric_calculator(
162                     seg_path=str(seg_path),
163                     gt_path=str(gt_path),
164                     save_path=None
165                 )
166                 metric_record.append(result_dict)
167                 pbar.update()
168             else:
169                 kwargs = {
170                     "seg_path": str(seg_path),
171                     "gt_path": str(gt_path),
172                     "save_path": None
173                 }
174                 pool.apply_async(
175                     func=metric_calculator,
176                     kwds=kwargs,
177                     callback=lambda x: update(pbar, metric_record, x),
178                     error_callback=error_back
179                 )
180         metric_record.sort(key=lambda x: x.get("name"))
181         if args.multiprocess:
182             pool.close()
183             pool.join()
184
185         for result in metric_record:
186             ws.append([result["name"], result["num_gt"]])
187
188         mean_metric = np.array([metric_record[i].get('num_gt') for i in range(len(metric_record))]).mean(axis=0).tolist()
189         ws.append(["mean", mean_metric])
190         wb.save(metric_result_path)

```


[illegible]

```

107         pre_mask_array=cs_dl,
108         spacing=self.spacing,
109         origin=self.origin,
110         direction=self.direction,
111         save_path=None,
112     )
113     cs_m_selected_sphere = result_dict.get("mask_sphere")
114     cs_m_selected_refined_num = result_dict.get("num")
115     assert cs_m_selected_sphere is not None, "rcs_selected_sphere is None"
116     assert cs_m_selected_refined_num is not None, "rcs_selected_refined_num is None"
117     rcs_selected = self.merge(m1=cs_m, m2=rs, dl=cs_m_selected_sphere)
118
119     # 7. select_two_biggest_connected_region on rcs_select -> rcs_selected_two
120     rcs_selected_two = select_two_biggest_connected_region(rcs_selected)
121
122     # 8. merge
123
124     arcs_selected = self.selected_final_merge(rcs_select=rcs_selected,
125         arcs=arcs,
126         dl1=cs_dl,
127         dl2=rcs_sphere)
128
129     # 9. select_two_biggest_connected_region on arcs_selected -> arcs_selected_two
130     arcs_selected_two = select_two_biggest_connected_region(arcs_selected)
131
132     # 10. selectively merge rs and cs_m without cs_dl -> rcs_new
133     result_dict = self.discontinuity_detection(seg_array=cs_m,
134         gt_array=None,
135         auxiliary_array=rs,
136         pre_mask_array=None,
137         spacing=self.spacing,
138         origin=self.origin,
139         direction=self.direction,
140         save_path=None,
141     )
142
143     cs_m_new_sphere = result_dict.get("mask_sphere")
144     cs_m_new_refined_num = result_dict.get("num")
145     assert cs_m_new_sphere is not None, "rcs_new_sphere is None"
146     assert cs_m_new_refined_num is not None, "rcs_new_refined_num is None"
147     rcs_new = self.merge(m1=cs_m, m2=rs, dl=cs_m_new_sphere)
148
149     # 11. select_two_biggest_connected_region on rcs_select -> rcs_selected_two
150     rcs_new_two = select_two_biggest_connected_region(rcs_new)
151
152     # 12. merge
153
154     arcs_new = self.selected_final_merge(rcs_select=rcs_new,
155         arcs=arcs,
156         dl1=cs_dl,
157         dl2=rcs_sphere)
158
159     # 13. select_two_biggest_connected_region on arcs_selected -> arcs_selected_two
160     arcs_new_two = select_two_biggest_connected_region(arcs_new)
161
162
163     # 14. save
164     self.save_image(rcs, self.save_subdir / f"{self.name}_RCS.nii.gz")
165     self.save_image(rcs_two, self.save_subdir / f"{self.name}_RCS_TWO.nii.gz")
166     self.save_image(arcs, self.save_subdir / f"{self.name}_ARCS.nii.gz")
167     self.save_image(arcs_two, self.save_subdir / f"{self.name}_ARCS_TWO.nii.gz")
168
169     self.save_image(rcs_selected, self.save_subdir / f"{self.name}_RCS_SELECTED.nii.gz")
170     self.save_image(rcs_selected_two, self.save_subdir / f"{self.name}_RCS_SELECTED_TWO.nii.gz")
171     self.save_image(cs_m_two, self.save_subdir / f"{self.name}_CS_M_TWO.nii.gz")
172     self.save_image(arcs_selected, self.save_subdir / f"{self.name}_ARCS_SELECTED.nii.gz")
173     self.save_image(arcs_selected_two, self.save_subdir / f"{self.name}_ARCS_SELECTED_TWO.nii.gz")
174     self.save_image(rcs_new, self.save_subdir / f"{self.name}_RCS_NEW.nii.gz")
175     self.save_image(rcs_new_two, self.save_subdir / f"{self.name}_RCS_NEW_TWO.nii.gz")
176     self.save_image(arcs_new, self.save_subdir / f"{self.name}_ARCS_NEW.nii.gz")
177     self.save_image(arcs_new_two, self.save_subdir / f"{self.name}_ARCS_NEW_TWO.nii.gz")
178
179     # save little segment and sphere for visualization
180     self.save_image(cs_m_selected_sphere, self.save_subdir / f"{self.name}_cs_m_selected_sphere.nii.gz")
181     self.save_image(rcs_sphere, self.save_subdir / f"{self.name}_rcs_sphere.nii.gz")
182     self.save_image(rcs * cs_dl, self.save_subdir / f"{self.name}_cs_m_merge_segment.nii.gz")
183     self.save_image(cs_m_selected_sphere * rs, self.save_subdir / f"{self.name}_rcs_selected_merge_segment.nii.gz")
184     self.save_image(rcs_sphere * cs_a, self.save_subdir / f"{self.name}_rcs_merge_segment.nii.gz")
185     self.save_image(cs_m_new_sphere, self.save_subdir / f"{self.name}_cs_m_new_sphere.nii.gz")
186     self.save_image(cs_m_new_sphere * rcs_new, self.save_subdir / f"{self.name}_rcs_new_merge_segment.nii.gz")
187
188     # move cs_a, cs_m, cs_dl, rs, gt to save_subdir for visualization
189     shutil.copy(self.cs_a_path, self.save_subdir / f"{self.name}_CS_A.nii.gz")
190     shutil.copy(self.cs_m_path, self.save_subdir / f"{self.name}_CS_M.nii.gz")
191     shutil.copy(self.cs_dl_path, self.save_subdir / f"{self.name}_CS_DL.nii.gz")
192     shutil.copy(self.rs_path, self.save_subdir / f"{self.name}_RS.nii.gz")
193
194     # save discontinuity metric
195     d = {
196         "name": self.name,
197         "cs_m_selected_refined_num": cs_m_selected_refined_num,
198         "arcs_improve": rcs_refined_num,
199         "cs_m_new_refined_num": cs_m_new_refined_num,
200     }
201     return d
202
203 def run_without_cs_a(self):
204     """
205     Run the post processing pipeline.
206     1. load the data
207     2. directly merge rs and cs_m with cs_dl -> rcs
208     3. select two biggest connected region on rcs -> rcs_two
209     4. run discontinuity detection on rcs with cs_a and cs_dl -> arcs
210     5. select_two_biggest_connected_region on arcs -> arcs_two
211
212     if selected merge:
213     6. selectively merge rs and cs_m with cs_dl -> rcs_selected
214     7. select_two_biggest_connected_region on rcs_select -> rcs_selected_two
215     8. run final merge on rcs_select with cs_a and cs_dl -> arcs_selected
216     9. select_two_biggest_connected_region on arcs_selected -> arcs_selected_two

```

```

216
217     if new merge:
218         10. selectively merge rs and cs_m without cs_dl -> rcs_new
219         11. select_two_biggest_connected_region on rcs_new -> rcs_new_two
220         12. run final merge on rcs_new with cs_a and cs_dl -> arcs_new
221         13. select_two_biggest_connected_region on arcs_new -> arcs_new_two
222     """
223     cs_a, rs, cs_dl, cs_m = None, None, None, None
224     # 1. load the data
225     cs_m, spacing, origin, direction = self.read_image(self.cs_m_path)
226     self.spacing, self.origin, self.direction = spacing, origin, direction
227
228     rs, _, _, _ = self.read_image(self.rs_path)
229     cs_dl, _, _, _ = self.read_image(self.cs_dl_path)
230
231     rs_two = select_two_biggest_connected_region(rs)
232
233     # 2. directly merge rs and cs_m with cs_dl -> rcs
234     rcs = self.merge(m1=cs_m, m2=rs, dl=cs_dl)
235
236     # 3. select_two_biggest_connected_region on rcs -> rcs_two
237     rcs_two = select_two_biggest_connected_region(rcs)
238     cs_m_two = select_two_biggest_connected_region(cs_m)
239
240     # 6. selectively merge rs and cs_m with cs_dl -> rcs_selected
241     result_dict = self.discontinuity_detection_default(seg_array=cs_m,
242                                                         gt_array=None,
243                                                         auxiliary_array=rs,
244                                                         pre_mask_array=cs_dl,
245                                                         spacing=self.spacing,
246                                                         origin=self.origin,
247                                                         direction=self.direction,
248                                                         save_path=None,
249                                                         )
250
251     cs_m_selected_sphere = result_dict.get("mask_sphere")
252     cs_m_selected_refined_num = result_dict.get("num")
253
254     assert cs_m_selected_sphere is not None, "rcs_selected_sphere is None"
255     assert cs_m_selected_refined_num is not None, "rcs_selected_refined_num is None"
256     rcs_selected = self.merge(m1=cs_m, m2=rs, dl=cs_m_selected_sphere)
257
258     # 7. select_two_biggest_connected_region on rcs_selected -> rcs_selected_two
259     rcs_selected_two = select_two_biggest_connected_region(rcs_selected)
260
261     # 10. selectively merge rs and cs_m without cs_dl -> rcs_new
262     result_dict = self.discontinuity_detection(seg_array=cs_m,
263                                                         gt_array=None,
264                                                         auxiliary_array=rs,
265                                                         pre_mask_array=None,
266                                                         spacing=self.spacing,
267                                                         origin=self.origin,
268                                                         direction=self.direction,
269                                                         save_path=None,
270                                                         )
271
272     cs_m_new_sphere = result_dict.get("mask_sphere")
273     cs_m_new_refined_num = result_dict.get("num")
274     assert cs_m_new_sphere is not None, "rcs_new_sphere is None"
275     assert cs_m_new_refined_num is not None, "rcs_new_refined_num is None"
276     rcs_new = self.merge(m1=cs_m, m2=rs, dl=cs_m_new_sphere)
277
278     # 11. select_two_biggest_connected_region on rcs_selected -> rcs_selected_two
279     rcs_new_two = select_two_biggest_connected_region(rcs_new)
280
281     # 13. select_two_biggest_connected_region on arcs_selected -> arcs_selected_two
282
283     # 14. save
284     self.save_image(rcs, self.save_subdir / f"{self.name}_RCS.nii.gz")
285     self.save_image(rcs_two, self.save_subdir / f"{self.name}_RCS_TWO.nii.gz")
286     self.save_image(rcs_selected, self.save_subdir / f"{self.name}_RCS_SELECTED.nii.gz")
287     self.save_image(rcs_selected_two, self.save_subdir / f"{self.name}_RCS_SELECTED_TWO.nii.gz")
288     self.save_image(cs_m_two, self.save_subdir / f"{self.name}_CS_M_TWO.nii.gz")
289     self.save_image(rcs_new, self.save_subdir / f"{self.name}_RCS_NEW.nii.gz")
290     self.save_image(rcs_new_two, self.save_subdir / f"{self.name}_RCS_NEW_TWO.nii.gz")
291     self.save_image(rs_two, self.save_subdir / f"{self.name}_RS_TWO.nii.gz")
292
293     # save little segment and sphere for visualization
294     self.save_image(cs_m_selected_sphere, self.save_subdir / f"{self.name}_cs_m_selected_sphere.nii.gz")
295     self.save_image(cs_m_selected_sphere * rs, self.save_subdir / f"{self.name}_rcs_selected_merge_segment.nii.gz")
296     self.save_image(cs_m_new_sphere, self.save_subdir / f"{self.name}_cs_m_new_sphere.nii.gz")
297
298     # move cs_a, cs_m, cs_dl, rs, gt to save_subdir for visualization
299     shutil.copy(self.cs_m_path, self.save_subdir / f"{self.name}_CS_M.nii.gz")
300     shutil.copy(self.cs_dl_path, self.save_subdir / f"{self.name}_CS_DL.nii.gz")
301     shutil.copy(self.rs_path, self.save_subdir / f"{self.name}_RS.nii.gz")
302
303     # save discontinuity metric
304     d = {
305         "name": self.name,
306         "cs_m_selected_refined_num": cs_m_selected_refined_num,
307         "cs_m_new_refined_num": cs_m_new_refined_num,
308     }
309     return d
310
311
312 def save_image(self, image, save_path, save_type=np.uint16):
313     sitk_image = sitk.GetImageFromArray(image.astype(save_type))
314     sitk_image.SetSpacing(self.spacing)
315     sitk_image.SetOrigin(self.origin)
316     sitk_image.SetDirection(self.direction)
317     sitk_image.WriteImage(sitk_image, save_path)
318
319 @staticmethod
320 def merge(m1, m2, dl):
321     m = m1 * (1 - dl) + m2 * dl
322     return m
323
324 @staticmethod

```

```

325 def read_image(path):
326     sitk_image = sitk.ReadImage(path)
327     image = sitk.GetArrayFromImage(sitk_image)
328     spacing = sitk_image.GetSpacing()
329     origin = sitk_image.GetOrigin()
330     direction = sitk_image.GetDirection()
331     return image, spacing, origin, direction
332
333 @staticmethod
334 def selected_final_merge(rcs_select, arcs, dl1, dl2):
335     """TODO: selected_final_merge."""
336     """
337     :param rcs_select: path
338     :param arcs: merge cs rs and a
339     :param dl1: first stage discontinuity label
340     :param dl2: second stage discontinuity label
341     :param save_path: save path
342     :param save_postfix: save postfix
343     """
344     if np.sum(dl2) != 0:
345         id_map = cc3d.connected_components(dl1, connectivity=26)
346         remain_id = np.unique(id_map[dl2 > 0])
347         remain_index = np.zeros_like(dl1)
348         for id in remain_id:
349             if id > 0:
350                 remain_index[id_map == id] = 1
351         arcs_selected = rcs_select * (1 - remain_index) + arcs * remain_index
352     else:
353         arcs_selected = rcs_select
354
355     return arcs_selected
356
357
358 class SparrowLinkDiscontinuityDetection:
359     def __init__(self,
360                 max_broken_size_percentage=0.1,
361                 min_sphere_radius_percentage=0.015,
362                 sphere_dilation_percentage=0.2, # 0.1
363                 region_select=4,
364                 cube_min_size=(10, 10, 10),
365                 skeleton_refine_times=3,
366                 angle_threshold_1=0.0,
367                 angle_threshold_2=0.0,
368                 view=False,
369                 region_threshold: Union[int, None] = 5,
370                 ):
371         self.max_broken_size_percentage = max_broken_size_percentage
372         self.min_sphere_radius_percentage = min_sphere_radius_percentage
373         self.sphere_dilation_percentage = sphere_dilation_percentage
374         self.region_select = region_select
375         self.cube_min_size = cube_min_size
376         self.skeleton_refine_times = skeleton_refine_times
377         self.region_threshold = region_threshold
378         self.angle_threshold_1 = angle_threshold_1
379         self.angle_threshold_2 = angle_threshold_2
380         self.view = view
381
382     """
383     :param max_broken_size_percentage: the max broken size percentage, used in constriction on distance
384     :param min_sphere_radius_percentage: the min sphere radius percentage, used for some small fracture
385     :param sphere_dilation_percentage: the sphere dilation percentage, to consider the structure around fracture area
386     :param region_select: the region select, matching points consist of start point and paired point, start point is...
387     ...constricted in the region with region_select.
388     :param cube_min_size: the min size of the cube, used for small fracture
389     :param skeleton_refine_times: the refine times of the skeleton, zhang algorithm might cause some small branch in the...
390     ...skeleton, so we need to refine the skeleton to delete the small branch
391     :param region_threshold: the region threshold, used to delete the small region in the segmentation
392     :param angle_threshold_1: the angle threshold 1, used for constriction on orientation
393     :param angle_threshold_2: the angle threshold 2, used for constriction on orientation
394     """
395     def __call__(self,
396                 seg_array: np.array = None,
397                 gt_array: np.array = None,
398                 auxiliary_array: np.array = None,
399                 pre_mask_array: np.array = None,
400                 spacing: Sequence[float] = None,
401                 origin: Sequence[float] = None,
402                 direction: Sequence[float] = None,
403                 save_path: str = None,
404                 ):
405         result_dict = discontinuity_detection(
406             label=seg_array,
407             GT=gt_array,
408             auxiliary=auxiliary_array,
409             pre_mask=pre_mask_array,
410             spacing=spacing,
411             direction=direction,
412             origin=origin,
413             save_path=save_path,
414             view=self.view,
415             max_broken_size_percentage=self.max_broken_size_percentage,
416             min_sphere_radius_percentage=self.min_sphere_radius_percentage,
417             sphere_dilation_percentage=self.sphere_dilation_percentage,
418             region_select=self.region_select,
419             cube_min_size=self.cube_min_size,
420             skeleton_refine_times=self.skeleton_refine_times,
421             region_threshold=self.region_threshold,
422             angle_threshold_1=self.angle_threshold_1,
423             angle_threshold_2=self.angle_threshold_2,
424         )
425         return result_dict
426
427 def update(pbar, record, result):
428     pbar.update()
429     record.append(result)
430
431
432 def error_back(err):
433     print(err)

```

```

434
435
436 if __name__ == "__main__":
437     import argparse
438     import tqdm
439     import pathlib
440     from multiprocessing import Pool
441
442     import warnings
443     warnings.filterwarnings("ignore") # ignore from np.int16 to np.uint8
444
445     parser = argparse.ArgumentParser()
446     parser.add_argument("--CS_M", type=str, default=None)
447     parser.add_argument("--CS_DL", type=str, default=None)
448     parser.add_argument("--RS", type=str, default=None)
449     parser.add_argument("--CS_A", type=str, default=None)
450     parser.add_argument("--save_dir", type=str, default=None)
451     parser.add_argument("--max_broken_size_percentage", type=float, default=0.3)
452     parser.add_argument("--multiprocess", action='store_true', default=False)
453     args = parser.parse_args()
454     CS_M_path = pathlib.Path(args.CS_M)
455     CS_DL_path = pathlib.Path(args.CS_DL)
456     RS_path = pathlib.Path(args.RS)
457     CS_A_path = pathlib.Path(args.CS_A) if args.CS_A is not None else None
458     data_list = list(CS_M_path.glob("*.nii.gz"))
459     print(f"\033[96m SparrowLink Postprocessing \033[00m")
460     pbar = tqdm.tqdm(total=len(list(data_list)), colour="#87cefa")
461     pbar.set_description("SparrowLink Processing")
462     process_record = []
463
464     if args.CS_A is not None:
465         if args.multiprocess:
466             poor = Pool(14)
467             for file in data_list:
468                 name = file.name
469                 processor = SparrowLinkPostProcess(
470                     cs_m_path=str(CS_M_path / name),
471                     cs_dl_path=str(CS_DL_path / name),
472                     rs_path=str(RS_path / name),
473                     cs_a_path=str(CS_A_path / name),
474                     save_path=args.save_dir,
475                 )
476                 if not args.multiprocess:
477                     process_record.append(processor.run())
478                     pbar.update()
479                 else:
480                     poor.apply_async(
481                         func=processor.run,
482                         callback=lambda x: update(pbar, process_record, x),
483                         error_callback=error_back
484                     )
485             if args.multiprocess:
486                 poor.close()
487                 poor.join()
488
489             process_record.sort(key=lambda x: x.get("name"))
490             with open(args.save_dir + "/process_record.json", "w") as f:
491                 json.dump(process_record, f, indent=4)
492
493         else:
494             if args.multiprocess:
495                 poor = Pool(14)
496                 for file in data_list:
497                     name = file.name
498                     processor = SparrowLinkPostProcess(
499                         cs_m_path=str(CS_M_path / name),
500                         cs_dl_path=str(CS_DL_path / name),
501                         rs_path=str(RS_path / name),
502                         cs_a_path=None,
503                         save_path=args.save_dir,
504                         max_broken_size_percentage=args.max_broken_size_percentage,
505                     )
506                     if not args.multiprocess:
507                         process_record.append(processor.run_without_cs_a())
508                     pbar.update()
509                     else:
510                         poor.apply_async(
511                             func=processor.run_without_cs_a,
512                             callback=lambda x: update(pbar, process_record, x),
513                             error_callback=error_back
514                         )
515             if args.multiprocess:
516                 poor.close()
517                 poor.join()
518
519             process_record.sort(key=lambda x: x.get("name"))
520             with open(args.save_dir + "/process_record.json", "w") as f:
521                 json.dump(process_record, f, indent=4)

```

```

0 SparrowLink/transform/Anatomical_augmentation_for_CCTA_images.md
1 # Anatomy-Informed Data Augmentation for Coronary Artery in CCTA Image
2 * Video of the on-the-fly anatomy-based data augmentation (slice view)
3 ![1709195253734_00h00m00s-00h00m07s] (https://github.com/xxsxxsxxs666/SparrowLink/assets/61532031/5443c459-72b7-480e-8b30-06ff231b956d)
4
5 * Video of the on-the-fly anatomy-based data augmentation. (3D)
6 ![3D Slicer 5 2 1 2024-02-29 15-57-46_00h00m00s-00h00m11s] (https://github.com/xxsxxsxxs666/SparrowLink/assets/61532031/c203e8bf-a892-4227-98cb-060fc3c40671)
7
8 * real CCTA image video:
9 ![1706926881079_00h00m00s-00h00m08s] (https://github.com/xxsxxsxxs666/SparrowLink/assets/61532031/c6cdce80-186f-44d0-bcc9-4abb36e9f1ef)
10
11 * You can use our anatomy-based data augmentation tool by simply plugging it into MONAI transform architecture:
12
13 ```python
14 save_transform = Compose(
15     [
16         LoadImaged(keys=["image", "label", "heart"]),
17         EnsureChannelFirstd(keys=["image", "label", "heart"]),
18         ArteryTransformD(keys=["image", "label"], image_key="image", artery_key="label", p_anatomy_per_sample=1,
19             p_contrast_per_sample=1,
20             contrast_reduction_factor_range=(0.6, 1), mask_blur_range=(3, 6),
21             mvf_scale_factor_range=(1, 2), mode=("bilinear", "nearest")),
22         # HeartTransformD(keys=["image", "label", "heart"], artery_key="label", heart_key="heart",
23             # p_anatomy_heart=0, p_anatomy_artery=1,
24             # dil_ranges=((-10, 10), (-5, -3)), directions_of_trans=((1, 1, 1), (1, 1, 1)), blur=(32, 8),
25             # mode=("bilinear", "nearest", "nearest"), visualize=True, batch_interpolate=True,
26             # threshold=(-1, 0.5, 0.5)),
27         # CASTransformD(keys=["image", "label", "heart"], label_key="label", heart_key="heart", p_anatomy_per_sample=1,
28             # dil_ranges=((-30, -40), (-300, -500)), directions_of_trans=((1, 1, 1), (1, 1, 1)), blur=[4, 32],
29             # mode=("bilinear", "nearest", "nearest"),),
30         SaveImaged(keys=["image"], output_dir=save_dir, output_postfix='spatial_transform_image',
31             print_log=True, padding_mode="zeros"),
32         SaveImaged(keys=["label"], output_dir=save_dir, output_postfix='spatial_transform_label',
33             print_log=True, padding_mode="zeros"),
34     ]
35 )
36 ```
37
38 ```python
39 save_transform = Compose(
40     [
41         LoadImaged(keys=["image", "label", "heart"]),
42         EnsureChannelFirstd(keys=["image", "label", "heart"]),
43         ArteryTransformD(keys=["image", "label"], image_key="image", artery_key="label", p_anatomy_per_sample=1,
44             p_contrast_per_sample=1,
45             contrast_reduction_factor_range=(0.6, 1), mask_blur_range=(3, 6),
46             mvf_scale_factor_range=(1, 2), mode=("bilinear", "nearest")),
47         RandCropByPosNegLabeld(
48             keys=["image", "label"],
49             label_key="label",
50             spatial_size=(128, 128, 128),
51             pos=3,
52             neg=1,
53             num_samples=4,
54             image_key="image",
55             image_threshold=0,
56         ),
57         SaveImaged(keys=["image"], output_dir=save_dir, output_postfix='spatial_transform_image',
58             print_log=True, padding_mode="zeros"),
59         SaveImaged(keys=["label"], output_dir=save_dir, output_postfix='spatial_transform_label',
60             print_log=True, padding_mode="zeros"),
61     ]
62 )
63 ```

```

```

0 SparrowLink/transform/AnatomyTransformD.py
1 import numpy as np
2 import torch
3 from monai.utils import TransformBackends, convert_to_tensor, ensure_tuple
4 from monai.data.meta_obj import get_track_meta
5 from monai.config import KeysCollection
6 from monai.transforms import MapTransform, RandomizableTransform, Randomizable, SpatialCrop, TraceableTransform
7
8 from typing import Dict, Hashable, List, Mapping, Optional, Sequence, Union, Tuple, Any
9 from batchgenerators.augmentations.utils import create_zero_centered_coordinate_mesh, elastic_deform_coordinates, \
10     interpolate_img, \
11     rotate_coords_2d, rotate_coords_3d, scale_coords, resize_segmentation, resize_multichannel_image, \
12     elastic_deform_coordinates_2
13 import warnings
14 from monai.transforms.utils_pytorch_numpy_unification import unravel_index
15 from monai.transforms.utils import correct_crop_centers, map_binary_to_indices, convert_to_dst_type, create_translate, \
16     ensure_tuple_rep, ensure_tuple
17 from monai.data.meta_tensor import MetaTensor
18 from copy import deepcopy
19 import time
20 from torch.nn.functional import grid_sample
21 from scipy.ndimage.filters import gaussian_filter
22 from skimage.morphology import skeletonize
23 from post_processing.fracture_detection import get_point_orientation
24 import cc3d
25
26
27 def get_mvf_by_gaussian_gradient(mask, spacing, blur, dil_magnitude, directions_of_trans, anisotropy_safety):
28     shape = mask.shape
29     coords = create_zero_centered_coordinate_mesh(shape)
30     t, u, v = get_organ_gradient_field(mask,
31                                       spacing=spacing,
32                                       blur=blur)
33     n_factor = np.sqrt(2 * np.pi)
34     if directions_of_trans[0]:
35         coords[0, :, :, :] = coords[0, :, :, :] + t * blur * dil_magnitude * n_factor
36     if directions_of_trans[1]:
37         coords[1, :, :, :] = coords[1, :, :, :] + u * blur * dil_magnitude * n_factor
38     if directions_of_trans[2]:
39         coords[2, :, :, :] = coords[2, :, :, :] + v * blur * dil_magnitude * n_factor
40     deformation_record = (t * dil_magnitude * n_factor, u * dil_magnitude * n_factor, v * dil_magnitude * n_factor)
41     for d in range(3):
42         ctr = shape[d] / 2 # !!!
43         coords[d] += ctr - 0.5 # !!!
44
45     if anisotropy_safety:
46         coords[0, 0, :, :][coords[0, 0, :, :] < 0] = 0.0
47         coords[0, 1, :, :][coords[0, 1, :, :] < 0] = 0.0
48         coords[0, -1, :, :][coords[0, -1, :, :] > (shape[-2] - 1)] = shape[-2] - 1
49         coords[0, -2, :, :][coords[0, -2, :, :] > (shape[-2] - 1)] = shape[-2] - 1
50
51     return coords, deformation_record
52
53
54 def find_random_one_numpy(image, patch_size=None):
55     # 找到值为1的所有点的坐标
56     # copy the image to avoid changing the original image
57     if patch_size is not None:
58         # set image to 0 in the corner
59         # copy the image to avoid changing the original image
60         image_copy = image.copy()
61         image_copy[:patch_size[0] // 2, :, :] = 0
62         image_copy[-patch_size[0] // 2:, :, :] = 0
63         image_copy[:, :patch_size[1] // 2, :] = 0
64         image_copy[:, -patch_size[1] // 2:, :] = 0
65         image_copy[:, :, :patch_size[2] // 2] = 0
66         image_copy[:, :, -patch_size[2] // 2:] = 0
67         ones_indices = np.where(image_copy > 0)
68     else:
69         ones_indices = np.where(image == 1)
70     # 转换坐标为列表形式 [(x1, y1), (x2, y2), ...]
71     ones_list = list(zip(ones_indices[0], ones_indices[1], ones_indices[2]))
72
73     if not ones_list:
74         return None # 如果没有找到值为1的点, 则返回None
75
76     return ones_list[np.random.randint(len(ones_list))]
77
78
79 def generate_slice_by_center_and_patch_size(center, patch_size):
80     x_slice = slice(center[0] - patch_size[0] // 2, center[0] + patch_size[0] // 2)
81     y_slice = slice(center[1] - patch_size[1] // 2, center[1] + patch_size[1] // 2)
82     z_slice = slice(center[2] - patch_size[2] // 2, center[2] + patch_size[2] // 2)
83
84     return (x_slice, y_slice, z_slice)
85
86 def get_organ_gradient_field(organ, spacing=(1, 1, 1), blur=32):
87     """
88     from batchgenerators.augmentations.utils, but data shape is (H, W, D) instead of (D, H, W)
89     The returned MVF is in image coordinates.
90     """
91     u_ratio = spacing[0] / spacing[1]
92     v_ratio = spacing[0] / spacing[2]
93
94     organ_blurred = gaussian_filter(organ.astype(float),
95                                   sigma=(blur, blur * u_ratio, blur * v_ratio),
96                                   order=0,
97                                   mode='nearest')
98     t, u, v = np.gradient(organ_blurred)
99     t = t
100     u = u * u_ratio
101     v = v * v_ratio
102
103     return t, u, v
104
105

```

```

106 def interpolator(data, coords, mode, border_mode=None, border_cval=None, keep_meta=True):
107     data = convert_to_tensor(data, track_meta=get_track_meta())
108     meta = data.meta.copy()
109     data_result = np.zeros_like(data)
110     if isinstance(mode, int):
111         for channel_id in range(data.shape[0]):
112             data_result[channel_id] = interpolate_img(np.array(data[channel_id]), coords, mode,
113                                                         border_mode, cval=border_cval)
114     data_result = MetaTensor(data_result, meta=meta) if keep_meta else data_result
115 else:
116     # import time
117     # tic = time.time()
118     data_batch = data.unsqueeze(0)
119     data_result = \
120         grid_sample(data_batch, coords, mode=mode, padding_mode="zeros", align_corners=False)[0]
121     data_result = MetaTensor(data_result, meta=meta) if keep_meta else data_result
122     # toc = time.time()
123     # print(f"interpolator time: {toc - tic}")
124     return data_result
125
126
127 def coords_numpy2torch(coords, shape, change_coords=True):
128     """Prepare for torch's grid-sampling"""
129     h, w, d = shape
130     if change_coords:
131         coords_torch = torch.from_numpy(coords)
132     else:
133         coords_torch = torch.from_numpy(coords).clone()
134     coords_permute = torch.flip(coords_torch, dims=[0]).unsqueeze(0).permute(0, 2, 3, 4, 1).to(torch.float32)
135     coords_permute[:, :, :, :, 0] /= (d - 1)
136     coords_permute[:, :, :, :, 1] /= (w - 1)
137     coords_permute[:, :, :, :, 2] /= (h - 1)
138     coords_permute -= 0.5
139     coords_permute *= 2
140     return coords_permute
141
142
143 def mesh_generator_tensor(patch_size, rand_state, p_el_per_sample: float = 1, p_rot_per_sample: float = 1,
144                             p_scale_per_sample: float = 1, do_elastic_deform=True,
145                             alpha=(0., 1000.), sigma=(10., 13.), do_rotation=True,
146                             angle_x=(0, 2 * np.pi), angle_y=(0, 2 * np.pi), angle_z=(0, 2 * np.pi),
147                             p_rot_per_axis: float = 1, do_scale=True, scale=(0.75, 1.25),
148                             independent_scale_for_each_axis=False, p_independent_scale_per_axis: int = 1,
149                             num_samples: int = 1):
150     """TODO: change numpy to torch"""
151     dim = len(patch_size)
152     coords_list = []
153     modified_coords_list = []
154     for _ in range(num_samples):
155         coords = create_zero_centered_coordinate_mesh(patch_size)
156         modified_coords = False
157         if do_elastic_deform and rand_state.uniform() < p_el_per_sample:
158             a = np.random.uniform(alpha[0], alpha[1])
159             s = np.random.uniform(sigma[0], sigma[1])
160             coords = elastic_deform_coordinates(coords, a, s)
161             modified_coords = True
162
163         if do_rotation and rand_state.uniform() < p_rot_per_sample:
164
165             if np.random.uniform() <= p_rot_per_axis:
166                 a_x = rand_state.uniform(angle_x[0], angle_x[1])
167             else:
168                 a_x = 0
169
170             if dim == 3:
171                 if np.random.uniform() <= p_rot_per_axis:
172                     a_y = rand_state.uniform(angle_y[0], angle_y[1])
173                 else:
174                     a_y = 0
175
176                 if np.random.uniform() <= p_rot_per_axis:
177                     a_z = rand_state.uniform(angle_z[0], angle_z[1])
178                 else:
179                     a_z = 0
180                 coords = rotate_coords_3d(coords, a_x, a_y, a_z)
181             else:
182                 coords = rotate_coords_2d(coords, a_x)
183             modified_coords = True
184         if do_scale and rand_state.uniform() < p_scale_per_sample:
185             if independent_scale_for_each_axis and rand_state.uniform() < p_independent_scale_per_axis:
186                 sc = []
187                 for _ in range(dim):
188                     if rand_state.random() < 0.5 and scale[0] < 1:
189                         sc.append(rand_state.uniform(scale[0], 1))
190                     else:
191                         sc.append(rand_state.uniform(max(scale[0], 1), scale[1]))
192             else:
193                 if rand_state.random() < 0.5 and scale[0] < 1:
194                     sc = rand_state.uniform(scale[0], 1)
195                 else:
196                     sc = rand_state.uniform(max(scale[0], 1), scale[1])
197             coords = scale_coords(coords, sc)
198             modified_coords = True
199         coords_list.append(coords)
200         modified_coords_list.append(modified_coords)
201     return coords_list, modified_coords_list
202
203
204
205
206 def generate_pos_neg_label_crop_centers(
207     label,
208     spatial_size: Union[Sequence[int], int],
209     num_samples: int,
210     pos_ratio: float,
211     label_spatial_shape: Sequence[int],
212     fg_indices = None,
213     bg_indices = None,
214     rand_state: Optional[np.random.RandomState] = None

```



```

214         rand_state: Optional[np.random.RandomState] = None,
215         allow_smaller: bool = False,
216     ) -> List[List[int]]:
217         """
218         Generate valid sample locations based on the label with option for specifying foreground ratio
219         Valid: samples sitting entirely within image, expected input shape: [C, H, W, D] or [C, H, W]
220
221     Args:
222         label: used for generating coordinates
223         image: for checking correction of cropping
224         spatial_size: spatial size of the ROIs to be sampled.
225         num_samples: total sample centers to be generated.
226         pos_ratio: ratio of total locations generated that have center being foreground.
227         label_spatial_shape: spatial shape of the original label data to unravel selected centers.
228         fg_indices: pre-computed foreground indices in 1 dimension.
229         bg_indices: pre-computed background indices in 1 dimension.
230         rand_state: numpy randomState object to align with other modules.
231         allow_smaller: if 'False', an exception will be raised if the image is smaller than
232             the requested ROI in any dimension. If 'True', any smaller dimensions will be set to
233             match the cropped size (i.e., no cropping in that dimension).
234
235     Raises:
236         ValueError: When the proposed roi is larger than the image.
237         ValueError: When the foreground and background indices lengths are 0.
238
239     """
240     if rand_state is None:
241         rand_state = np.random.random.__self__ # type: ignore
242
243     if fg_indices is None or bg_indices is None:
244         fg_indices, bg_indices = map_binary_to_indices(label, image=None)
245
246     centers = []
247     fg_indices = np.asarray(fg_indices) if isinstance(fg_indices, Sequence) else fg_indices
248     bg_indices = np.asarray(bg_indices) if isinstance(bg_indices, Sequence) else bg_indices
249     if len(fg_indices) == 0 and len(bg_indices) == 0:
250         raise ValueError("No sampling location available.")
251
252     if len(fg_indices) == 0 or len(bg_indices) == 0:
253         pos_ratio = 0 if len(fg_indices) == 0 else 1
254         warnings.warn(
255             f"Num foregrounds {len(fg_indices)}, Num backgrounds {len(bg_indices)}, "
256             f"unable to generate class balanced samples, setting `pos_ratio` to {pos_ratio}."
257         )
258
259     for _ in range(num_samples):
260         indices_to_use = fg_indices if rand_state.rand() < pos_ratio else bg_indices
261         random_int = rand_state.randint(len(indices_to_use))
262         idx = indices_to_use[random_int]
263         center = unravel_index(idx, label_spatial_shape).tolist()
264         # shift center to range of valid centers
265         centers.append(correct_crop_centers(center, spatial_size, label_spatial_shape, allow_smaller))
266
267     return centers
268
269
270 class CASTransformD(Randomizable, MapTransform):
271     """
272     Add three components in deformable transformation
273     1. Muscle squeezing and muscle relaxation -> based on vessel segmentation, using normal vector of the surface.
274     2. Heart motion -> based on chambers segmentation
275     3. Cardiac Motion or curve deformation -> based on centerline of segmentation
276     Args:
277         `dil_ranges`: dilation range per organs
278         `modalities`: on which input channels should the transformation be applied
279         `directions_of_trans`: to which directions should the organs be dilated per organs
280         `p_per_sample`: probability of the transformation per organs
281         `spacing_ratio`: ratio of the transversal plane spacing and the slice thickness, in our case it was 0.3125/3
282         `blur`: Gaussian kernel parameter, we used the value 32 for 0.3125mm transversal plane spacing
283         `anisotropy_safety`: it provides a certain protection against transformation artifacts in 2 slices from the image border
284         `max_annotation_value`: the value that should be still relevant for the main task
285         `replace_value`: segmentation values larger than the `max_annotation_value` will be replaced with
286     """
287     backend = [TransformBackends.NUMPY, TransformBackends.TORCH]
288
289     def __init__(self,
290                 keys: KeysCollection,
291                 label_key: str = 'label',
292                 heart_key: str = 'heart',
293                 p_anatomy_per_sample: float = 0.5,
294                 dil_ranges: Tuple[Tuple[int, int], Tuple[int, int]] = ((0, 0), (0, 0)),
295                 directions_of_trans: Tuple[Tuple[int, int, int], Tuple[int, int, int]] = ((1, 1, 1), (1, 1, 1)),
296                 spacing_ratio: float = 0.334/0.5,
297                 blur: List = [32, 32],
298                 anisotropy_safety: bool = True,
299                 max_annotation_value: int = 1,
300                 allow_missing_keys: bool = False,
301                 mode: Union[Sequence[int], int, Sequence[str], str] = 1,
302                 border_mode: str = 'constant',
303                 cval: float = 0.0,):
304         MapTransform.__init__(self, keys, allow_missing_keys)
305         self.label_key = label_key
306         self.heart_key = heart_key
307         self.p_anatomy = p_anatomy_per_sample
308         self.dilation_ranges = dil_ranges
309         self.directions_of_trans = directions_of_trans
310         self.spacing_ratio = spacing_ratio
311         self.blur = blur
312         self.anisotropy_safety = anisotropy_safety
313         self.max_annotation_value = max_annotation_value
314         self.border_mode = ensure_tuple_rep(border_mode, len(self.keys))
315         self.border_cval = ensure_tuple_rep(cval, len(self.keys))
316         self.mode = ensure_tuple_rep(mode, len(self.keys))
317
318     def __call__(self, data: Mapping[Hashable, torch.Tensor]) -> Dict[Hashable, torch.Tensor]:
319         """TODO: if num_samples > 1, this function uses for loop to interpolate the data, which is not efficient."""
320         d: Dict = dict(data)
321         mask = [d[self.label_key],]
322         if self.heart_key is not None:

```

```

323         mask.append(d[self.heart_key])
324
325     m, deformation_record = self.randomize(mask=mask)
326
327     deformation_shape = list(d[self.label_key].shape)
328     deformation_shape[0] = 3
329     meta = d[self.label_key].meta.copy()
330     label_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
331     label_deformation_field[0, :, :, :] = torch.tensor(deformation_record[0][0])
332     label_deformation_field[1, :, :, :] = torch.tensor(deformation_record[0][1])
333     label_deformation_field[2, :, :, :] = torch.tensor(deformation_record[0][2])
334     d['label_df'] = MetaTensor(label_deformation_field, meta=meta)
335     if self.heart_key is not None:
336         heart_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
337         heart_deformation_field[0, :, :, :] = torch.tensor(deformation_record[1][0])
338         heart_deformation_field[1, :, :, :] = torch.tensor(deformation_record[1][1])
339         heart_deformation_field[2, :, :, :] = torch.tensor(deformation_record[1][2])
340         d['heart_df'] = MetaTensor(heart_deformation_field, meta=meta)
341         heart_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
342         heart_deformation_field[0, :, :, :] = torch.tensor(deformation_record[2][0])
343         heart_deformation_field[1, :, :, :] = torch.tensor(deformation_record[2][1])
344         heart_deformation_field[2, :, :, :] = torch.tensor(deformation_record[2][2])
345         d['heart_df1'] = MetaTensor(heart_deformation_field, meta=meta)
346         heart_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
347         heart_deformation_field[0, :, :, :] = torch.tensor(deformation_record[3][0])
348         heart_deformation_field[1, :, :, :] = torch.tensor(deformation_record[3][1])
349         heart_deformation_field[2, :, :, :] = torch.tensor(deformation_record[3][2])
350         d['heart_df2'] = MetaTensor(heart_deformation_field, meta=meta)
351         heart_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
352         heart_deformation_field[0, :, :, :] = torch.tensor(deformation_record[4][0])
353         heart_deformation_field[1, :, :, :] = torch.tensor(deformation_record[4][1])
354         heart_deformation_field[2, :, :, :] = torch.tensor(deformation_record[4][2])
355         d['heart_df3'] = MetaTensor(heart_deformation_field, meta=meta)
356         heart_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
357         heart_deformation_field[0, :, :, :] = torch.tensor(deformation_record[5][0])
358         heart_deformation_field[1, :, :, :] = torch.tensor(deformation_record[5][1])
359         heart_deformation_field[2, :, :, :] = torch.tensor(deformation_record[5][2])
360         d['heart_df4'] = MetaTensor(heart_deformation_field, meta=meta)
361         heart_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
362         heart_deformation_field[0, :, :, :] = torch.tensor(deformation_record[6][0])
363         heart_deformation_field[1, :, :, :] = torch.tensor(deformation_record[6][1])
364         heart_deformation_field[2, :, :, :] = torch.tensor(deformation_record[6][2])
365         d['heart_df5'] = MetaTensor(heart_deformation_field, meta=meta)
366         heart_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
367         heart_deformation_field[0, :, :, :] = torch.tensor(deformation_record[7][0])
368         heart_deformation_field[1, :, :, :] = torch.tensor(deformation_record[7][1])
369         heart_deformation_field[2, :, :, :] = torch.tensor(deformation_record[7][2])
370         d['heart_df6'] = MetaTensor(heart_deformation_field, meta=meta)
371
372     for key, border_mode, border_cval, order in self.key_iterator(d, self.border_mode, self.border_cval, self.mode):
373         d[key] = self.interpolator(data=d[key], coords=m, border_mode=border_mode, border_cval=border_cval,
374                                   mode=order,)
375
376     return d
377
378 def randomize(self,
379              mask: List = None,
380              spacing=None,):
381     """
382     return coordinate mesh and dilation magnitude
383     """
384     deformation_record = []
385     if self.R.uniform() < self.p_anatomy:
386         shape = mask[0].shape
387         coords = create_zero_centered_coordinate_mesh(shape[1:])
388         if spacing is not None:
389             # TODO: spacing in 1, 2 dimension may be different
390             spacing_ratio = spacing[0] / spacing[2]
391         else:
392             spacing_ratio = self.spacing_ratio
393
394         for i in range(len(mask)):
395             dil_magnitude = np.random.uniform(low=self.dilation_ranges[i][0], high=self.dilation_ranges[i][1])
396             if i == 0:
397                 t, u, v = get_organ_gradient_field(mask[i][0] > 0,
398                                                    spacing_ratio=spacing_ratio,
399                                                    blur=self.blur[i])
400             elif i == 1:
401                 t, u, v = get_organ_gradient_field(mask[i][0] > 0,
402                                                    spacing_ratio=spacing_ratio,
403                                                    blur=self.blur[i])
404             t_1, u_1, v_1 = get_organ_gradient_field(mask[i][0] == 1,
405                                                    spacing_ratio=spacing_ratio,
406                                                    blur=self.blur[i])
407             t_2, u_2, v_2 = get_organ_gradient_field(mask[i][0] == 2,
408                                                    spacing_ratio=spacing_ratio,
409                                                    blur=self.blur[i])
410             t_3, u_3, v_3 = get_organ_gradient_field(mask[i][0] == 3,
411                                                    spacing_ratio=spacing_ratio,
412                                                    blur=self.blur[i])
413             t_4, u_4, v_4 = get_organ_gradient_field(mask[i][0] == 4,
414                                                    spacing_ratio=spacing_ratio,
415                                                    blur=self.blur[i])
416             t_5, u_5, v_5 = get_organ_gradient_field(mask[i][0] == 5,
417                                                    spacing_ratio=spacing_ratio,
418                                                    blur=self.blur[i])
419             t_6, u_6, v_6 = get_organ_gradient_field(mask[i][0] == 6,
420                                                    spacing_ratio=spacing_ratio,
421                                                    blur=self.blur[i])
422
423             sigma = self.blur[i]
424             if self.directions_of_trans[i][0]:
425                 coords[0, :, :, :] = coords[0, :, :, :] + t * (sigma ** 2) * dil_magnitude
426             if self.directions_of_trans[i][1]:
427                 coords[1, :, :, :] = coords[1, :, :, :] + u * (sigma ** 2) * dil_magnitude
428             if self.directions_of_trans[i][2]:
429                 coords[2, :, :, :] = coords[2, :, :, :] + v * (sigma ** 2) * dil_magnitude
430             if self.directions_of_trans[i][3]:

```

```

430     self.directions_of_trans[1][2].
431     coords[2, :, :, :] = coords[2, :, :, :] + v * (sigma ** 2) * dil_magnitude
432
433     deformation_record.append((t * dil_magnitude, u * dil_magnitude, v * dil_magnitude)) # * spacing_ratio))
434     if i == 1:
435         deformation_record.append((t_1 * dil_magnitude, u_1 * dil_magnitude, v_1 * dil_magnitude))
436         deformation_record.append((t_2 * dil_magnitude, u_2 * dil_magnitude, v_2 * dil_magnitude))
437         deformation_record.append((t_3 * dil_magnitude, u_3 * dil_magnitude, v_3 * dil_magnitude))
438         deformation_record.append((t_4 * dil_magnitude, u_4 * dil_magnitude, v_4 * dil_magnitude))
439         deformation_record.append((t_5 * dil_magnitude, u_5 * dil_magnitude, v_5 * dil_magnitude))
440         deformation_record.append((t_6 * dil_magnitude, u_6 * dil_magnitude, v_6 * dil_magnitude))
441
442     for d in range(3):
443         ctr = shape[d + 1] / 2 # !!!
444         coords[d] += ctr - 0.5 # !!!
445
446     if self.anisotropy_safety:
447         coords[0, 0, :, :][coords[0, 0, :, :] < 0] = 0.0
448         coords[0, 1, :, :][coords[0, 1, :, :] < 0] = 0.0
449         coords[0, -1, :, :][coords[0, -1, :, :] > (shape[-3] - 1)] = shape[-3] - 1
450         coords[0, -2, :, :][coords[0, -2, :, :] > (shape[-3] - 1)] = shape[-3] - 1
451     else:
452         coords = None
453
454     return coords, deformation_record
455
456 def interpolator(self, data, coords, mode, border_mode=None, border_cval=None):
457     data = convert_to_tensor(data, track_meta=get_track_meta())
458     meta = data.meta.copy()
459     data_result = np.zeros_like(data)
460     if isinstance(mode, int):
461         for channel_id in range(data.shape[0]):
462             data_result[channel_id] = interpolate_img(np.array(data[channel_id]), coords, mode,
463                                                         border_mode, cval=border_cval)
464         data_result = MetaTensor(data_result, meta=meta)
465     else:
466         h, w, d = data.shape[1], data.shape[2], data.shape[3]
467         coords = coords[::1, :, :, :].copy() # xyz -> zyx
468         coords_permute = torch.from_numpy(coords).unsqueeze(0).permute(0, 2, 3, 4, 1).to(torch.float32)
469         data_batch = data.unsqueeze(0)
470         # coords_norm = (coords_permute + 1) / torch.tensor([h, w, d], dtype=torch.float32).reshape(1, 1, 1, 1, 3)
471         coords_norm = torch.zeros_like(coords_permute)
472         coords_norm[:, :, :, 0, :] = 0, coords_norm[:, :, :, 1, :] = 1, coords_norm[:, :, :, 2, :] = \
473             (coords_permute[:, :, :, 0] + 1) / d, (coords_permute[:, :, :, 1] + 1) / w, \
474             (coords_permute[:, :, :, 2] + 1) / h
475         coords_norm = (coords_norm - 0.5) * 2
476         data_result = \
477             grid_sample(data_batch, coords_norm, mode=mode, padding_mode="zeros", align_corners=False)[0]
478         data_result = MetaTensor(data_result, meta=meta)
479     return data_result
480
481
482
483 class HeartTransformD(Randomizable, MapTransform):
484     """
485     Add three components in deformable transformation
486     1. Cardiac muscle squeezing and muscle relaxation -> based on vessel segmentation, using normal vector of the surface.
487     Args:
488     `dil_ranges`: dilation range per organs
489     `modalities`: on which input channels should the transformation be applied
490     `directions_of_trans`: to which directions should the organs be dilated per organs
491     `p_per_sample`: probability of the transformation per organs
492     `spacing_ratio`: ratio of the transversal plane spacing and the slice thickness, in our case it was 0.3125/3
493     `blur`: Gaussian kernel parameter, we used the value 32 for 0.3125mm transversal plane spacing
494     `anisotropy_safety`: it provides a certain protection against transformation artifacts in 2 slices from the image border
495     `max_annotation_value`: the value that should be still relevant for the main task
496     `replace_value`: segmentation values larger than the `max_annotation_value` will be replaced with
497     `heart_select`: select which chamber to do dilation and shrink, In this work [1, 2, 3, 4, 5, 6] represent
498     [inner left-aorta, left-ventricular, right-aorta, right-ventricular, main-artery, left-aorta]
499     `heart_key`: heart segmentation
500     `label_key`: coronary artery segmentation
501     """
502     backend = [TransformBackends.NUMPY, TransformBackends.TORCH]
503
504     def __init__(self,
505                  keys: KeysCollection,
506                  heart_key: str = 'heart',
507                  artery_key: str = None,
508
509                  heart_select: Tuple[Union[int, Tuple[int]]] = ((5, ), (2, 4), (3, 6)),
510                  p_anatomy_heart: float = 0.5,
511                  p_anatomy_artery: float = 0.5,
512                  dil_ranges: Tuple[Tuple[float, float], Tuple[float, float]] = ((0, 0), (0, 0)),
513                  directions_of_trans: Tuple[Tuple[int, int, int], Tuple[int, int, int]] = ((1, 1, 1), (1, 1, 1)),
514                  blur: Tuple = (32, 32),
515                  anisotropy_safety: bool = True,
516                  max_annotation_value: int = 1,
517                  allow_missing_keys: bool = False,
518                  mode: Union[Sequence[int], int, Sequence[str], str] = 1,
519                  batch_interpolate: bool = False,
520                  threshold: Tuple = None,
521                  border_mode: str = 'constant',
522                  cval: float = 0.0,
523                  visualize: bool = False,
524                  del_heart: bool = True,
525                  ):
526         MapTransform.__init__(self, keys, allow_missing_keys)
527         self.heart_key = heart_key
528         self.heart_select = heart_select
529         self.artery_key = artery_key
530         self.p_anatomy_heart = p_anatomy_heart
531         self.p_anatomy_artery = p_anatomy_artery
532         self.dilation_ranges = dil_ranges
533         self.directions_of_trans = directions_of_trans
534         self.blur = blur
535         self.anisotropy_safety = anisotropy_safety
536         self.max_annotation_value = max_annotation_value
537         self.border_mode = ensure_tuple_rep(border_mode, len(self.keys))
538         self.border_cval = ensure_tuple_rep(cval, len(self.keys))
539         self.mode = ensure_tuple_rep(mode, len(self.keys))

```

```

539 self.visualize = visualize
540
541 self.do_heart_transformation = False
542 self.do_artery_transformation = False
543 self.dil_magnitude_heart = 0
544 self.dil_magnitude_artery = 0
545 self.random_index = None # random index to select chambers
546 self.threshold = ensure_tuple_rep(threshold, len(self.keys))
547 self.batch_interpolate = batch_interpolate
548
549 self.del_heart = del_heart
550
551 def __call__(self, data: Mapping[Hashable, torch.Tensor]) -> Dict[Hashable, torch.Tensor]:
552     """TODO: if num_samples > 1, this function uses for loop to interpolate the data, which is not efficient."""
553     d: Dict = dict(data)
554     spacing = d[self.heart_key].meta['pixdim'][1:4]
555     shape = d[self.heart_key].shape
556     self.randomize()
557     if self.do_heart_transformation:
558         mask = [d[self.heart_key]]
559         mask_dilation = torch.zeros_like(mask[0][0])
560         for idx in self.random_index:
561             mask_dilation += mask[0][0] == idx
562
563         m, deformation_heart = get_mvf_by_gaussian_gradient(mask=mask_dilation > 0,
564                                                         spacing=spacing,
565                                                         blur=self.blur[0],
566                                                         dil_magnitude=self.dil_magnitude_heart,
567                                                         directions_of_trans=self.directions_of_trans[0],
568                                                         anisotropy_safety=self.anisotropy_safety)
569         if not isinstance(self.mode[0], int): # assert all modes are the same, either int or str
570             m = coords_numpy2torch(m, shape[1:], change_coords=True)
571         if self.batch_interpolate:
572             # only support torch
573             self.batch_interpolator(d=d, m=m, shape=shape[1:])
574         else:
575             for key, border_mode, border_cval, order in \
576                 self.key_iterator(d, self.border_mode, self.border_cval, self.mode):
577                 # use torch's grid-sampling to interpolate the data
578                 d[key] = interpolator(data=d[key], coords=m, border_mode=border_mode, border_cval=border_cval,
579                                     mode=order, )
580
581         if self.visualize:
582             deformation_shape = list(d[self.heart_key].shape)
583             deformation_shape[0] = 3
584             meta = d[self.heart_key].meta.copy()
585             heart_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
586             # change to world coordinates
587             heart_deformation_field[0, :, :, :] = torch.tensor(deformation_heart[0]) * spacing[0]
588             heart_deformation_field[1, :, :, :] = torch.tensor(deformation_heart[1]) * spacing[1]
589             heart_deformation_field[2, :, :, :] = torch.tensor(deformation_heart[2]) * spacing[2]
590             d['heart_df'] = MetaTensor(heart_deformation_field, meta=meta)
591
592     if self.do_artery_transformation:
593         m, deformation_artery = get_mvf_by_gaussian_gradient(mask=d[self.artery_key][0] > 0,
594                                                         spacing=spacing,
595                                                         blur=self.blur[1],
596                                                         dil_magnitude=self.dil_magnitude_artery,
597                                                         directions_of_trans=self.directions_of_trans[1],
598                                                         anisotropy_safety=self.anisotropy_safety)
599         if not isinstance(self.mode[0], int): # assert all modes are the same, either int or str
600             m = coords_numpy2torch(m, shape[1:], change_coords=True)
601
602         if self.batch_interpolate:
603             self.batch_interpolator(d=d, m=m, shape=shape[1:])
604         else:
605             for key, border_mode, border_cval, order in \
606                 self.key_iterator(d, self.border_mode, self.border_cval, self.mode):
607                 d[key] = interpolator(data=d[key], coords=m, border_mode=border_mode, border_cval=border_cval,
608                                     mode=order, )
609
610         if self.visualize:
611             deformation_shape = list(d[self.artery_key].shape)
612             deformation_shape[0] = 3
613             meta = d[self.artery_key].meta.copy()
614             label_deformation_field = torch.zeros(size=deformation_shape, dtype=torch.float32)
615             label_deformation_field[0, :, :, :] = torch.tensor(deformation_artery[0]) * spacing[0]
616             label_deformation_field[1, :, :, :] = torch.tensor(deformation_artery[1]) * spacing[1]
617             label_deformation_field[2, :, :, :] = torch.tensor(deformation_artery[2]) * spacing[2]
618             d['label_df'] = MetaTensor(label_deformation_field, meta=meta)
619
620     if self.del_heart:
621         d.pop(self.heart_key) # delete heart segmentation to save memory
622     return d
623
624 def batch_interpolator(self, d, m, shape, device="cpu"):
625     i_h, i_w, i_d = shape
626     c = len(self.mode)
627     batch_data = torch.zeros((1, c, i_h, i_w, i_d))
628     meta_dict = {}
629
630     for i, key in enumerate(self.key_iterator(d)):
631         batch_data[0][i] = d[key]
632         meta_dict[key] = d[key].meta
633
634     with torch.no_grad():
635         data_result = \
636             grid_sample(batch_data.to(device), m.to(device), mode=self.mode[0], padding_mode="zeros",
637                         align_corners=False)[0]
638
639     for key, t, i in self.key_iterator(d, self.threshold, range(c)):
640         if t > 0:
641             d[key] = MetaTensor((data_result[i] > t).unsqueeze(0),
642                                 meta=meta_dict[key])
643         else:
644             d[key] = MetaTensor(data_result[i].unsqueeze(0), meta=meta_dict[key])
645
646

```

```

647 def randomize(self, data: Any = None):
648     """ return coordinate mesh and dilation magnitude"""
649     if self.R.uniform() < self.p_anatomy_heart:
650         self.do_heart_transformation = True
651         self.dil_magnitude_heart = self.R.uniform(low=9, high=self.dilation_ranges[0][1])
652         self.random_index = self.heart_select[self.R.randint(0, len(self.heart_select))]
653     else:
654         self.do_heart_transformation = False
655
656     if self.artery_key is not None:
657         if self.R.uniform() < self.p_anatomy_artery:
658             self.do_artery_transformation = True
659             self.dil_magnitude_artery = self.R.uniform(low=self.dilation_ranges[1][0],
660                                                         high=self.dilation_ranges[1][1])
661             print(f"artery dilation magnitude: {self.dil_magnitude_artery}")
662         else:
663             self.do_artery_transformation = False
664
665     # shape = mask[0].shape
666     # coords = create_zero_centered_coordinate_mesh(shape[1:])
667     # random_index = self.heart_select[np.random.randint(0, len(self.heart_select))]
668     # mask_dilation = torch.zeros_like(mask[0][0])
669     # for idx in random_index:
670     #     mask_dilation += mask[0][0] == idx
671     # t_h, u_h, v_h = get_organ_gradient_field(mask_dilation > 0,
672     #                                         spacing=spacing,
673     #                                         blur=self.blur[0])
674     # dil_magnitude_heart = np.random.uniform(low=self.dilation_ranges[0][0], high=self.dilation_ranges[0][1])
675     # dil_magnitude_artery = None
676     # t_a, u_a, v_a = None, None, None
677
678     # if self.artery_key is not None:
679     #     assert len(mask) > 0 and len(self.blur) > 0 and len(self.dilation_ranges) > 0, \
680     #         "artery label is not available or parameters are not set"
681     #     dil_magnitude_artery = np.random.uniform(low=self.dilation_ranges[0][0], high=self.dilation_ranges[0][1])
682     #     t_a, u_a, v_a = get_organ_gradient_field(mask[1][0] > 0,
683     #                                             spacing=spacing,
684     #                                             blur=self.blur[1])
685
686     # n_factor = np.sqrt(2 * np.pi)
687     # if self.directions_of_trans[0][0]:
688     #     coords[0, :, :, :] = coords[0, :, :, :] + t_h * self.blur[0] * dil_magnitude_heart * n_factor
689     # if self.directions_of_trans[0][1]:
690     #     coords[1, :, :, :] = coords[1, :, :, :] + u_h * self.blur[0] * dil_magnitude_heart * n_factor
691     # if self.directions_of_trans[0][2]:
692     #     coords[2, :, :, :] = coords[2, :, :, :] + v_h * self.blur[0] * dil_magnitude_heart * n_factor
693
694     # deformation_record.append((t_h * dil_magnitude_heart * n_factor, u_h * dil_magnitude_heart * n_factor,
695     #                             v_h * dil_magnitude_heart * n_factor))
696
697     # if self.artery_key is not None:
698     #     if self.directions_of_trans[0][0]:
699     #         coords[0, :, :, :] = coords[0, :, :, :] + t_a * self.blur[1] * dil_magnitude_artery * n_factor
700     #     if self.directions_of_trans[0][1]:
701     #         coords[1, :, :, :] = coords[1, :, :, :] + u_a * self.blur[1] * dil_magnitude_artery * n_factor
702     #     if self.directions_of_trans[0][2]:
703     #         coords[2, :, :, :] = coords[2, :, :, :] + v_a * self.blur[1] * dil_magnitude_artery * n_factor
704     #     deformation_record.append(
705     #         (t_a * dil_magnitude_artery * n_factor, u_a * dil_magnitude_artery * n_factor,
706     #          v_a * dil_magnitude_artery * n_factor))
707
708     # for d in range(3):
709     #     ctr = shape[d + 1] / 2 # !!!
710     #     coords[d] += ctr - 0.5 # !!!
711
712     # if self.anisotropy_safety:
713     #     coords[0, 0, :, :][coords[0, 0, :, :] < 0] = 0.0
714     #     coords[0, 1, :, :][coords[0, 1, :, :] < 0] = 0.0
715     #     coords[0, -1, :, :][coords[0, -1, :, :] > (shape[-3] - 1)] = shape[-3] - 1
716     #     coords[0, -2, :, :][coords[0, -2, :, :] > (shape[-3] - 1)] = shape[-3] - 1
717
718     # else:
719     #     coords = None
720
721     # return coords, deformation_record
722
723 def adjust_contrast(patch_image, contrast_reduction_factor, patch_patch_slice, mask_blur=4):
724     # generate weighted mask
725     patch_patch_mask = np.ones_like(patch_image)
726     patch_patch_mask[patch_patch_slice] = contrast_reduction_factor
727     patch_patch_mask_gaussian = gaussian_filter(patch_patch_mask, sigma=mask_blur)
728     min_max = patch_patch_mask_gaussian.max() - patch_patch_mask_gaussian.min()
729     assert min_max > 0.00001, f"min_max should be larger than 0, {patch_patch_mask_gaussian.max(), patch_patch_mask_gaussian.min()}" \
730         f"{patch_patch_mask.max(), patch_patch_mask.min()}" \
731         f"{contrast_reduction_factor}" \
732         f"{patch_patch_slice}"
733     patch_min = patch_patch_mask_gaussian.min()
734     patch_patch_mask_gaussian -= patch_min
735     patch_patch_mask_gaussian *= ((1 - contrast_reduction_factor) / min_max)
736     patch_patch_mask_gaussian += contrast_reduction_factor
737     # patch_patch_mask_gaussian = (patch_patch_mask_gaussian - patch_min) * (1 - contrast_reduction_factor) / min_max \
738     #     + contrast_reduction_factor
739
740     patch_mean = patch_image.mean()
741     patch_image = (patch_image - patch_mean) * patch_patch_mask_gaussian + patch_mean
742     return torch.from_numpy(patch_image)
743
744 def check_shape(a, b):
745     flag = True
746     for i, j in zip(a, b):
747         if i > j:
748             flag = False
749     return flag
750
751
752 def scale_tuple(a, t):
753     return tuple([a[i]*t for i in range(len(a))])
754
755

```

```

755
756 def generate_random_vector_perpendicular_to(n):
757     n = n / np.linalg.norm(n)
758
759     # 生成一个随机向量
760     r = np.random.rand(3)
761
762     # 计算垂直于n的向量
763     v = np.cross(n, r)
764
765     # 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
766     while np.linalg.norm(v) == 0:
767         r = np.random.rand(3)
768         v = np.cross(n, r)
769     # 标准化v
770     v = v / np.linalg.norm(v)
771
772     return v
773
774
775 def get_sphere_by_center_and_radius(center, radius, spacing, shape):
776     """
777     center: sphere center (h, w, d)
778     radius: sphere radius, mm
779     spacing: voxel spacing, mm
780     shape: image shape or patch shape
781     """
782     h, w, d = shape
783     tmp = tuple([np.arange(i) for i in (h, w, d)])
784     coords = np.array(np.meshgrid(*tmp, indexing='ij')).astype(float)
785     coords -= np.array(center)[:, None, None, None]
786     coords = coords * np.array(spacing)[:, None, None, None]
787     distance = np.linalg.norm(coords, axis=0)
788     sphere = np.zeros((h, w, d))
789     sphere[distance <= radius] = 1.0
790     return sphere
791
792
793 def generate_mvf_vector(center, artery, centerline, direction_image, spacing, shape, scale_factor, radius, sigma=4):
794     orientation_centerline_spacing = \
795         get_point_orientation(point=center, centerline=centerline, spacing=spacing, size=3,
796                               is_end_point=False)[0]
797     if orientation_centerline_spacing is None:
798         return None
799     direction_image_com = np.array([direction_image[0], direction_image[4], direction_image[8]])
800     orientation_centerline_world = orientation_centerline_spacing * direction_image_com
801     random_direction_3d = generate_random_vector_perpendicular_to(orientation_centerline_world)
802     h, w, d = shape
803     mvf_world = np.zeros((h, w, d, 3))
804     mvf_world += random_direction_3d
805     center_point = center[0]
806
807     mvf_mask = get_sphere_by_center_and_radius(center=center_point, radius=radius, spacing=spacing, shape=shape)
808     mvf_mask = mvf_mask * (artery > 0)
809     connected_region = cc3d.connected_components(mvf_mask) # only select the region where center locates
810     area_id = connected_region[center_point[0], center_point[1], center_point[2]]
811     mvf_mask = mvf_mask * (connected_region == area_id)
812     mvf_weight = gaussian_filter(mvf_mask, sigma=sigma) * scale_factor * sigma
813
814     mvf_world = mvf_world * mvf_weight[..., None]
815     mvf_image = (mvf_world / np.array(direction_image_com) / np.array(spacing))
816     h, w, d = shape
817     tmp = tuple([np.arange(i) for i in (h, w, d)])
818     coords = np.array(np.meshgrid(*tmp, indexing='ij')).astype(float)
819     return coords + mvf_image.transpose(3, 0, 1, 2)
820
821
822 class ArteryTransformD(Randomizable, MapTransform):
823     """
824     1. local contrast change
825     2. artery shift
826     3. local vessel shrink or dilation
827     """
828     backend = [TransformBackends.NUMPY, TransformBackends.TORCH]
829
830     def __init__(self,
831                  keys: KeysCollection,
832                  image_key: str = None,
833                  contrast_patch_patch_size_range: Union[Tuple, Tuple[Tuple]] = ((10, 30), (10, 30), (10, 30)),
834                  deform_patch_patch_size_range: Tuple = (5.0, 10.0),
835                  contrast_reduction_factor_range: Tuple = (0.6, 1),
836                  mvf_scale_factor_range: Tuple = (-2, 2),
837                  mask_blur_range: Tuple = (3, 6),
838                  artery_key: str = None,
839                  centerline_key: str = None,
840                  allow_missing_keys: bool = False,
841                  p_anatomy_per_sample: float = 0.0,
842                  p_contrast_per_sample: float = 0.0,
843                  mode: Union[Sequence[int], int, Sequence[str], str] = 1,
844                  border_mode: str = 'constant',
845                  cval: float = 0.0,
846                  visualize: bool = False):
847         MapTransform.__init__(self, keys, allow_missing_keys)
848
849         self.artery_key = artery_key
850         self.image_key = image_key
851         self.centerline_key = centerline_key
852         self.p_anatomy = p_anatomy_per_sample
853         self.p_contrast = p_contrast_per_sample
854         self.border_mode = ensure_tuple_rep(border_mode, len(self.keys))
855         self.border_cval = ensure_tuple_rep(cval, len(self.keys))
856         self.mode = ensure_tuple_rep(mode, len(self.keys))
857         self.visualize = visualize
858         self.do_artery_deformation = False
859         self.do_local_contrast_change = False
860         self.contrast_reduction_factor_range = contrast_reduction_factor_range
861         self.contrast_reduction_factor = None
862         self.mask_blur_range = mask_blur_range
863         self.mask_blur = None

```



```

863 self.contrast_patch_patch_size_range = ensure_tuple_rep(contrast_patch_patch_size_range, 3)
864 self.deform_patch_patch_size_range = deform_patch_patch_size_range
865 self.mvf_mask_radius = None
866 self.patch_patch_size_x_1, self.patch_patch_size_y_1, self.patch_patch_size_z_1 = None, None, None
867 self.mvf_scale_factor_range = mvf_scale_factor_range
868 self.mvf_scale_factor = None
869
870 def __call__(self, data: Mapping[Hashable, torch.Tensor]) -> Dict[Hashable, torch.Tensor]:
871     """TODO: if num_samples > 1, this function uses for loop to interpolate the data, which is not efficient."""
872     d: Dict = dict(data)
873     # check whether this patch has label
874     self.randomize()
875     if self.do_artery_deformation or self.do_local_contrast_change:
876         artery = d[self.artery_key]
877         shape = artery.shape[1:]
878         original_affine = artery.meta.get('original_affine')
879         direction_image = np.sign(original_affine[:3, :3].reshape(-1,)).tolist()
880         # spacing = artery.meta.get('pixdim')[1:4]
881         r_matrix = artery.meta.get('affine')[:3, :3].reshape(-1,).tolist()
882         spacing = np.array([np.abs(r_matrix[0]), np.abs(r_matrix[4]), np.abs(r_matrix[8])], dtype=np.float32)
883         origin = original_affine[:3, 3]
884         if artery.sum() < 100: # if no artery, return directly. We assume the artery should be larger than 100
885             return d
886         if self.centerline_key is not None:
887             """get centerline"""
888             centerline = data[self.centerline_key]
889         else:
890             centerline = skeletonize(d[self.artery_key][0] > 0)[None, ]
891     else:
892         return d
893     if self.do_local_contrast_change:
894         patch_patch_size = (self.patch_patch_size_x_1, self.patch_patch_size_y_1, self.patch_patch_size_z_1)
895         center = np.array(find_random_one_numpy(centerline[0], patch_size=patch_patch_size))[None,]
896         if center[0] is not None:
897             patch_slice = generate_slice_by_center_and_patch_size(center[0], patch_size=patch_patch_size)
898             assert patch_slice[-1].start >= 0, f"patch_slice is None, {center[0]}, {patch_patch_size}"
899             d[self.image_key][0] = adjust_contrast(patch_image=d[self.image_key][0],
900                                                  patch_patch_slice=patch_slice,
901                                                  mask_blur=self.mask_blur,
902                                                  contrast_reduction_factor=self.contrast_reduction_factor)
903     if self.do_artery_deformation:
904         patch_patch_size = (int(self.mvf_mask_radius / spacing[0]), int(self.mvf_mask_radius / spacing[1]),
905                             int(self.mvf_mask_radius / spacing[2]))
906         # check the patch_patch_size is suitable
907         if check_shape(a=scale_tuple(patch_patch_size, 3), b=shape):
908             center = np.array(find_random_one_numpy(centerline[0],
909                                                     patch_size=scale_tuple(patch_patch_size, 3)))[None,]
910             # times 3 here to reduce deformation effect in the edge
911             if center[0] is not None:
912                 mvf = generate_mvf_vector(center=center, artery=artery[0], centerline=centerline[0] > 0,
913                                          scale_factor=self.mvf_scale_factor, radius=self.mvf_mask_radius,
914                                          direction_image=direction_image, spacing=spacing, shape=shape)
915                 if mvf is None:
916                     RuntimeError("MVF is None, probably because the center is too close to the edge or isolated")
917                 return d
918             if not isinstance(self.mode[0], int): # assert all modes are the same, either int or str
919                 mvf = coords_numpy2torch(mvf, shape, change_coords=True)
920             for key, border_mode, border_cval, order in \
921                 self.key_iterator(d, self.border_mode, self.border_cval, self.mode):
922                 # use torch's grid-sampling to interpolate the data
923                 d[key] = interpolator(data=d[key], coords=mvf, border_mode=border_mode, border_cval=border_cval,
924                                     mode=order, )
925     return d
926
927 def randomize(self, data: Any = None):
928     if self.R.uniform() < self.p_anatomy:
929         self.do_artery_deformation = True
930         self.mvf_mask_radius = self.R.uniform(self.deform_patch_patch_size_range[0],
931                                              self.deform_patch_patch_size_range[1])
932         self.mvf_scale_factor = self.R.uniform(self.mvf_scale_factor_range[0],
933                                              self.mvf_scale_factor_range[1])
934     else:
935         self.do_artery_deformation = False
936
937     if self.R.uniform() < self.p_contrast:
938         self.do_local_contrast_change = True
939         self.contrast_reduction_factor = self.R.uniform(self.contrast_reduction_factor_range[0],
940                                                      self.contrast_reduction_factor_range[1])
941         self.patch_patch_size_x_1 = self.R.randint(self.contrast_patch_patch_size_range[0][0],
942                                                  self.contrast_patch_patch_size_range[0][1])
943         self.patch_patch_size_y_1 = self.R.randint(self.contrast_patch_patch_size_range[1][0],
944                                                  self.contrast_patch_patch_size_range[1][1])
945         self.patch_patch_size_z_1 = self.R.randint(self.contrast_patch_patch_size_range[2][0],
946                                                  self.contrast_patch_patch_size_range[2][1])
947         self.mask_blur = self.R.uniform(self.mask_blur_range[0], self.mask_blur_range[1])
948     else:
949         self.do_local_contrast_change = False

```

```

0 SparrowLink/transform/cardiac_transformation.py
1 import numpy as np
2 import torch
3 from monai.utils import TransformBackends
4 from typing import Any, Callable, Dict, Hashable, List, Mapping, Optional, Sequence, Union
5 from monai.config import IndexSelection, KeysCollection, SequenceStr, NdarrayOrTensor, DtypeLike, KeysCollection
6 from monai.utils.type_conversion import convert_data_type, convert_to_dst_type, convert_to_tensor, get_equivalent_dtype
7 from monai.data.meta_tensor import MetaTensor
8 from monai.metrics.utils import get_surface_distance, get_mask_edges
9
10
11 from monai.transforms import (
12     MapTransform,
13 )
14 from openpyxl import Workbook, load_workbook
15 import os
16 from scipy import ndimage
17 import numpy as np
18
19
20 class UseHeartsegDeleteInformationd(MapTransform):
21
22     backend = [TransformBackends.TORCH]
23
24     def __init__(
25         self,
26         keys: KeysCollection,
27         heart_key: str = 'heart',
28         vessel_key: str = 'vessel',
29         heart_dilation_time: int = 1,
30         vessel_dilation_time: int = 1,
31         dilation_struct: int = 1,
32         allow_missing_keys: bool = False,
33     ) -> None:
34         """
35         test_key: show whether your mask is reserved totally.
36         """
37         self.heart_dilation_time = heart_dilation_time
38         self.vessel_dilation_time = vessel_dilation_time
39         self.vessel_key = vessel_key
40         self.heart_key = heart_key
41         self.dilation_struct = dilation_struct
42
43         super().__init__(keys, allow_missing_keys=allow_missing_keys)
44
45     def __call__(self, data: Mapping[Hashable, torch.Tensor]) -> Dict[Hashable, torch.Tensor]:
46         d = dict(data)
47         heart_region = convert_to_tensor(data=d[self.heart_key])
48         vessel_region = convert_to_tensor(data=d[self.vessel_key])
49         heart_region = heart_region.clip(min=0, max=1) > 0
50         vessel_region = vessel_region.clip(min=0, max=1) > 0
51         if self.heart_dilation_time > 1:
52             heart_region = self.dilation(x=heart_region, dilation_time=self.heart_dilation_time)
53         if self.vessel_dilation_time > 1:
54             vessel_region = self.dilation(x=vessel_region, dilation_time=self.vessel_dilation_time)
55         region = (heart_region + vessel_region) > 0
56         d[self.heart_key] = MetaTensor(region, meta=d[self.heart_key].meta)
57         for key in self.key_iterator(d):
58             if key not in [self.heart_key, self.vessel_key]:
59                 d[key] = d[key] * region
60         return d
61
62     def dilation(self, x: torch.Tensor, dilation_time: int = 1):
63         shape = x.shape
64         if len(shape) > 3:
65             x = x.squeeze()
66             x = np.array(x.squeeze())
67             struct1 = ndimage.generate_binary_structure(3, self.dilation_struct)
68             x = ndimage.binary_dilation(x, structure=struct1, iterations=dilation_time).astype(x.dtype)
69         if len(shape) > 3:
70             return torch.tensor(x).unsqueeze(dim=0)
71         else:
72             return torch.tensor(x)

```



```

0 SparrowLink/transform/fast_crop.py
1 import numpy as np
2
3 def fast_index(img, point):
4     """point: n * 3"""
5     H, W, D = img.shape
6     point = point[:, 2] + D * point[:, 1] + W * D * point[:, 0]
7     batch_crop = img.reshape(-1)[point]
8     return batch_crop
9
10
11 def fast_crop(img, point):
12     """
13     point: n * h * w * d * 3
14     return: n, h, w, d
15     """
16     n, h, w, d, _ = point.shape
17     point = point.reshape(n * h * w * d, 3)
18     batch_crop = fast_index(img, point)
19     batch_crop = batch_crop.reshape(n, h, w, d)
20
21     return batch_crop
22
23
24 def generate_cube(point, h, w, d):
25     """
26     point: n * 3
27     return: n * h * w * d * 3
28     """
29
30     """
31     n, _ = point.shape
32     x = np.arange(-h, h+1)
33     y = np.arange(-w, w+1)
34     z = np.arange(-d, d+1)
35
36     [Y, X, Z] = np.meshgrid(y, x, z)
37     crop_region_index = np.zeros((x.shape[0] * y.shape[0] * z.shape[0], 3))
38     crop_region_index[:, 0], crop_region_index[:, 1], crop_region_index[:, 2] = \
39         X.reshape(-1), Y.reshape(-1), Z.reshape(-1)
40     region_index = point[:, None, :] + crop_region_index[None, :, :]
41     region_index = region_index.reshape(n, x.shape[0], y.shape[0], z.shape[0], 3)
42     return region_index.astype(np.uint16)
43
44
45 def fancy_indexing(imgs, centers, pw, ph):
46     n = imgs.shape[0]
47     img_i, RGB, x, y = np.ogrid[:n, :3, :pw, :ph]
48     corners = centers - [pw//2, ph//2]
49     x_i = x + corners[:, 0, None, None, None]
50     y_i = y + corners[:, 1, None, None, None]
51     return imgs[img_i, RGB, x_i, y_i]
52
53
54 if __name__ == '__main__':
55     H = 8
56     W = 8
57     D = 8
58     img = np.arange(H * W * D).reshape(H, W, D)
59     point = np.array([[3, 3, 3],
60                       [2, 2, 2],
61                       [1, 1, 1], ])
62
63     cube_point = generate_cube(point, h=1, w=1, d=1)
64
65     crop = fast_crop(img, point=cube_point)
66
67     print(img)
68     print(crop)

```

```

0 SparrowLink/transform/IntensityTransformD.py
1 import numpy as np
2 import torch
3 from monai.utils import TransformBackends
4 from typing import Any, Callable, Dict, Hashable, List, Mapping, Optional, Sequence, Union
5 from monai.config import IndexSelection, KeysCollection, SequenceStr, NdarrayOrTensor, DtypeLike, KeysCollection
6 from monai.data.meta_obj import get_track_meta
7 from monai.utils.type_conversion import convert_data_type, convert_to_dst_type, convert_to_tensor, get_equivalent_dtype
8 from monai.transforms.utils_pytorch_numpy_unification import clip
9 from monai.data.meta_tensor import MetaTensor
10 from monai.metrics.utils import get_surface_distance, get_mask_edges
11 from monai.transforms import MapTransform, RandomizableTransform, Randomizable
12
13 from batchgenerators.augmentations.noise_augmentations import augment_gaussian_blur, augment_gaussian_noise, \
14     augment_rician_noise
15 from batchgenerators.augmentations.color_augmentations import augment_contrast, augment_brightness_additive, \
16     augment_brightness_multiplicative, augment_gamma, augment_illumination, augment_PCA_shift
17 from batchgenerators.augmentations.resample_augmentations import augment_linear_downsampling_scipy
18 from typing import Any, Callable, Dict, Hashable, List, Mapping, Optional, Sequence, Union, Tuple
19
20
21 class CTNormalizedD(MapTransform):
22     """
23     Dictionary-based wrapper of :py:class:`monai.transforms.NormalizeIntensity`.
24     This transform can normalize only non-zero values or entire image, and can also calculate
25     mean and std on each channel separately.
26     """
27
28     backend = [TransformBackends.TORCH, TransformBackends.NUMPY]
29
30     def __init__(
31         self,
32         keys: KeysCollection,
33         mean_intensity: float = None,
34         std_intensity: float = None,
35         lower_bound: float = None,
36         upper_bound: float = None,
37         allow_missing_keys: bool = False,
38     ) -> None:
39         super().__init__(keys, allow_missing_keys)
40         self.mean_intensity = mean_intensity
41         self.std_intensity = std_intensity
42         self.lower_bound = lower_bound
43         self.upper_bound = upper_bound
44
45     @staticmethod
46     def normalize_intensity(
47         img: NdarrayOrTensor,
48         mean_intensity: float = None,
49         std_intensity: float = None,
50         lower_bound: float = None,
51         upper_bound: float = None,
52         dtype: DtypeLike = np.float32,
53     ):
54         img = convert_to_tensor(img, track_meta=get_track_meta())
55         dtype = dtype or img.dtype
56         img = convert_to_tensor(img, track_meta=get_track_meta())
57         img = clip(img, a_min=lower_bound, a_max=upper_bound)
58         img = (img - mean_intensity) / max(std_intensity, 1e-8)
59         # img = MetaTensor(img, meta=img.meta)
60         ret: NdarrayOrTensor = convert_data_type(img, dtype=dtype)[0]
61         return ret
62
63     def __call__(self, data: Mapping[Hashable, NdarrayOrTensor]) -> Dict[Hashable, NdarrayOrTensor]:
64         d = dict(data)
65         for key in self.keys:
66             d[key] = self.normalize_intensity(img=d[key],
67                                             mean_intensity=self.mean_intensity,
68                                             std_intensity=self.std_intensity,
69                                             lower_bound=self.lower_bound,
70                                             upper_bound=self.upper_bound,
71                                             )
72         return d
73
74
75 class BrightnessMultiplicativeD(RandomizableTransform, MapTransform):
76     """
77     Adds additive Gaussian Noise
78     :param keys: selecting the keys to be transformed
79     :param prob: probability of the noise being added, per sample
80     :param prob_per_channel: probability of the noise being added, per channel
81     CAREFUL: This transform will modify the value range of your data!
82     """
83
84     backend = [TransformBackends.TORCH, TransformBackends.NUMPY]
85
86     def __init__(
87         self,
88         keys: KeysCollection,
89         prob: float = 0.1,
90         prob_per_channel: float = 1.0,
91         per_channel: bool = True,
92         multiplier_range: Tuple[float, float] = (0.5, 2),
93         allow_missing_keys: bool = False,
94     ) -> None:

```

```

95     MapTransform.__init__(self, keys, allow_missing_keys)
96     RandomizableTransform.__init__(self, prob=prob)
97     self.keys = keys
98     self.prob = prob
99     self.multiplier_range = multiplier_range
100    self.per_channel = per_channel
101    self.prob_per_channel = prob_per_channel
102
103    def set_random_state(
104        self, seed: Optional[int] = None, state: Optional[np.random.RandomState] = None
105    ) -> "BrightnessMultiplicativeD":
106        super().set_random_state(seed=seed, state=state)
107        return self
108
109    def __call__(self, data: Mapping[Hashable, NdarrayOrTensor]) -> Dict[Hashable, NdarrayOrTensor]:
110        d = dict(data)
111        self.randomize(None)
112        for key in self.keys:
113            if self._do_transform:
114                meta = None
115                if get_track_meta():
116                    meta = d[key].meta
117                dtype = d[key].dtype
118                shape = d[key].shape
119                assert len(shape) == 4, "img should be 4D array, (c, w, h, d)"
120
121                img = np.array(d[key])
122                img = augment_brightness_multiplicative(
123                    data_sample=img,
124                    multiplier_range=self.multiplier_range,
125                    per_channel=self.per_channel,
126                )
127                if meta:
128                    img = MetaTensor(img, meta=meta)
129                d[key] = convert_data_type(img, dtype=dtype)[0]
130
131        return d
132
133
134    class ContrastAugmentationD(RandomizableTransform, MapTransform):
135        """
136        Adds additive Gaussian Noise
137        :param keys: selecting the keys to be transformed
138        :param prob: probability of the noise being added, per sample
139        :param prob_per_channel: probability of the noise being added, per channel
140        CAREFUL: This transform will modify the value range of your data!
141        """
142        backend = [TransformBackends.TORCH, TransformBackends.NUMPY]
143
144        def __init__(
145            self,
146            keys: KeysCollection,
147            prob: float = 0.1,
148            prob_per_channel: float = 1.0,
149            per_channel: bool = True,
150            contrast_range: Tuple[float, float] = (0.75, 1.25),
151            preserve_range: bool = True,
152            allow_missing_keys: bool = False,
153        ) -> None:
154
155            MapTransform.__init__(self, keys, allow_missing_keys)
156            RandomizableTransform.__init__(self, prob=prob)
157            self.keys = keys
158            self.prob = prob
159            self.contrast_range = contrast_range
160            self.preserve_range = preserve_range
161            self.per_channel = per_channel
162            self.prob_per_channel = prob_per_channel
163
164        def set_random_state(
165            self, seed: Optional[int] = None, state: Optional[np.random.RandomState] = None
166        ) -> "ContrastAugmentationD":
167            super().set_random_state(seed=seed, state=state)
168            return self
169
170        def __call__(self, data: Mapping[Hashable, NdarrayOrTensor]) -> Dict[Hashable, NdarrayOrTensor]:
171            d = dict(data)
172            self.randomize(None)
173            for key in self.keys:
174                if self._do_transform:
175                    meta = None
176                    if get_track_meta():
177                        meta = d[key].meta
178                    dtype = d[key].dtype
179                    shape = d[key].shape
180                    assert len(shape) == 4, "img should be 4D array, (c, w, h, d)"
181
182                    img = np.array(d[key])
183                    img = augment_contrast(
184                        data_sample=img,
185                        contrast_range=self.contrast_range,
186                        preserve_range=self.preserve_range,
187                        per_channel=self.per_channel,
188                    )
189                    if meta:
190                        img = MetaTensor(img, meta=meta)
191                    d[key] = convert_data_type(img, dtype=dtype)[0]
192
193            return d

```

```

191         d[key] = convert_data_type(img, dtype=dtype)[0]
192
193     return d
194
195
196 class SimulateLowResolutionD(RandomizableTransform, MapTransform):
197     """
198     Adds additive Gaussian Noise
199     :param keys: selecting the keys to be transformed
200     :param prob: probability of the noise being added, per sample
201     :param prob_per_channel: probability of the noise being added, per channel
202     CAREFUL: This transform will modify the value range of your data!
203     """
204     backend = [TransformBackends.TORCH, TransformBackends.NUMPY]
205
206     def __init__(
207         self,
208         keys: KeysCollection,
209         prob: float = 0.1,
210         prob_per_channel: float = 1.0,
211         per_channel: bool = True,
212         zoom_range: Tuple[float, float] = (0.5, 1),
213         order_downsample: int = 0,
214         order_upsample: int = 3,
215         ignore_axes: bool = None,
216         allow_missing_keys: bool = False,
217     ) -> None:
218
219         MapTransform.__init__(self, keys, allow_missing_keys)
220         RandomizableTransform.__init__(self, prob=prob)
221         self.keys = keys
222         self.prob = prob
223         self.zoom_range = zoom_range
224         self.order_downsample = order_downsample
225         self.order_upsample = order_upsample
226         self.ignore_axes = ignore_axes
227         self.per_channel = per_channel
228         self.prob_per_channel = prob_per_channel
229
230     def set_random_state(
231         self, seed: Optional[int] = None, state: Optional[np.random.RandomState] = None
232     ) -> "SimulateLowResolutionD":
233         super().set_random_state(seed=seed, state=state)
234         return self
235
236     def __call__(self, data: Mapping[Hashable, NdarrayOrTensor]) -> Dict[Hashable, NdarrayOrTensor]:
237         d = dict(data)
238         self.randomize(None)
239         for key in self.keys:
240             if self._do_transform:
241                 meta = None
242                 if get_track_meta():
243                     meta = d[key].meta
244                 dtype = d[key].dtype
245                 shape = d[key].shape
246                 assert len(shape) == 4, "img should be 4D array, (c, w, h, d)"
247
248                 img = np.array(d[key])
249                 img = augment_linear_downsampling_scipy(
250                     data_sample=img,
251                     zoom_range=self.zoom_range,
252                     order_downsample=self.order_downsample,
253                     order_upsample=self.order_upsample,
254                     ignore_axes=self.ignore_axes,
255                     per_channel=self.per_channel,
256                 )
257
258                 if meta:
259                     img = MetaTensor(img, meta=meta)
260                 d[key] = convert_data_type(img, dtype=dtype)[0]
261         return d
262
263
264 class GammaD(RandomizableTransform, MapTransform):
265     """
266     Adds additive Gaussian Noise
267     :param keys: selecting the keys to be transformed
268     :param prob: probability of the noise being added, per sample
269     :param prob_per_channel: probability of the noise being added, per channel
270     CAREFUL: This transform will modify the value range of your data!
271     """
272     backend = [TransformBackends.TORCH, TransformBackends.NUMPY]
273
274     def __init__(
275         self,
276         keys: KeysCollection,
277         prob: float = 0.1,
278         prob_per_channel: float = 1.0,
279         per_channel: bool = True,
280         gamma_range: Tuple[float, float] = (0.5, 2),
281         invert_image: bool = False,
282         retain_stats: bool = False,
283         allow_missing_keys: bool = False,
284     ) -> None:
285
286         MapTransform.__init__(self, keys, allow_missing_keys)
287         RandomizableTransform.__init__(self, prob=prob)

```

```

288     self.keys = keys
289     self.prob = prob
290     self.gamma_range = gamma_range
291     self.invert_image = invert_image
292     self.retain_stats = retain_stats
293     self.per_channel = per_channel
294     self.prob_per_channel = prob_per_channel
295
296     def set_random_state(
297         self, seed: Optional[int] = None, state: Optional[np.random.RandomState] = None
298     ) -> "GammaD":
299         super().set_random_state(seed=seed, state=state)
300         return self
301
302     def __call__(self, data: Mapping[Hashable, NdarrayOrTensor]) -> Dict[Hashable, NdarrayOrTensor]:
303         d = dict(data)
304         self.randomize(None)
305         for key in self.keys:
306             if self._do_transform:
307                 meta = None
308                 if get_track_meta():
309                     meta = d[key].meta
310                 dtype = d[key].dtype
311                 shape = d[key].shape
312                 assert len(shape) == 4, "img should be 4D array, (c, w, h, d)"
313
314                 img = np.array(d[key])
315                 img = augment_gamma(
316                     data_sample=img,
317                     gamma_range=self.gamma_range,
318                     invert_image=self.invert_image,
319                     retain_stats=self.retain_stats,
320                     per_channel=self.per_channel,
321                 )
322                 if meta:
323                     img = MetaTensor(img, meta=meta)
324                 d[key] = convert_data_type(img, dtype=dtype)[0]
325
326         return d

```

```

0 SparrowLink/transform/NoiseTransformD.py
1 import numpy as np
2 import torch
3 from monai.utils import TransformBackends
4 from typing import Any, Callable, Dict, Hashable, List, Mapping, Optional, Sequence, Union
5 from monai.config import IndexSelection, KeysCollection, SequenceStr, NdarrayOrTensor, DtypeLike, KeysCollection
6 from monai.data.meta_obj import get_track_meta
7 from monai.utils.type_conversion import convert_data_type, convert_to_dst_type, convert_to_tensor, get_equivalent_dtype
8 from monai.transforms.utils_pytorch_numpy_unification import clip
9 from monai.data.meta_tensor import MetaTensor
10 from monai.metrics.utils import get_surface_distance, get_mask_edges
11 from monai.transforms import MapTransform, RandomizableTransform, Randomizable
12
13 from batchgenerators.augmentations.noise_augmentations import augment_gaussian_blur, augment_gaussian_noise, \
14     augment_riician_noise
15 from typing import Any, Callable, Dict, Hashable, List, Mapping, Optional, Sequence, Union, Tuple
16
17
18 class GaussianNoiseD(RandomizableTransform, MapTransform):
19     """
20     Adds additive Gaussian Noise
21     :param keys: selecting the keys to be transformed
22     :param prob: probability of the noise being added, per sample
23     :param prob_per_channel: probability of the noise being added, per channel
24     CAREFUL: This transform will modify the value range of your data!
25     """
26     backend = [TransformBackends.TORCH, TransformBackends.NUMPY]
27
28     def __init__(
29         self,
30         keys: KeysCollection,
31         prob: float = 0.1,
32         prob_per_channel: float = 1.0,
33         per_channel: bool = True,
34         noise_variance: Tuple[float, float] = (0.0, 0.1),
35         allow_missing_keys: bool = False,
36     ) -> None:
37
38         MapTransform.__init__(self, keys, allow_missing_keys)
39         RandomizableTransform.__init__(self, prob=prob)
40         self.keys = keys
41         self.prob = prob
42         self.noise_variance = noise_variance
43         self.per_channel = per_channel
44         self.prob_per_channel = prob_per_channel
45
46     def set_random_state(
47         self, seed: Optional[int] = None, state: Optional[np.random.RandomState] = None
48     ) -> "GaussianNoiseD":
49         super().set_random_state(seed=seed, state=state)
50         return self
51
52     def __call__(self, data: Mapping[Hashable, NdarrayOrTensor]) -> Dict[Hashable, NdarrayOrTensor]:
53         d = dict(data)
54         self.randomize(None)
55         for key in self.keys:
56             if self._do_transform:
57                 meta = None
58                 if get_track_meta():
59                     meta = d[key].meta
60                 dtype = d[key].dtype
61                 shape = d[key].shape
62                 assert len(shape) == 4, "img should be 4D array, (c, w, h, d)"
63
64                 img = np.array(d[key])
65                 img = augment_gaussian_noise(data_sample=img,
66                                             noise_variance=self.noise_variance,
67                                             p_per_channel=self.prob_per_channel,
68                                             per_channel=self.per_channel,
69                                             )
70                 if meta:
71                     img = MetaTensor(img, meta=meta)
72                 d[key] = convert_data_type(img, dtype=dtype)[0]
73
74         return d
75
76
77 class GaussianBlurD(RandomizableTransform, MapTransform):
78     """
79     Adds additive Gaussian Noise
80     :param keys: selecting the keys to be transformed
81     :param prob: probability of the noise being added
82     CAREFUL: This transform will modify the value range of your data!
83
84     """
85     backend = [TransformBackends.TORCH, TransformBackends.NUMPY]
86
87     def __init__(
88         self,
89         keys: KeysCollection,
90         blur_sigma: Tuple[float, float] = (0.0, 0.1),
91         prob: float = 0.2,
92         prob_per_channel: float = 1.0,
93         per_channel: bool = True,
94         allow_missing_keys: bool = False,
95     ) -> None:

```

```

95     MapTransform.__init__(self, keys, allow_missing_keys)
96     RandomizableTransform.__init__(self, prob=prob)
97     self.prob = prob
98     self.blur_sigma = blur_sigma
99     self.per_channel = per_channel
100     self.prob_per_channel = prob_per_channel
101
102     def set_random_state(
103         self, seed: Optional[int] = None, state: Optional[np.random.RandomState] = None
104     ) -> "GaussianBlurD":
105         super().set_random_state(seed=seed, state=state)
106         return self
107
108     def __call__(self, data: Mapping[Hashable, NdarrayOrTensor]) -> Dict[Hashable, NdarrayOrTensor]:
109         d = dict(data)
110         self.randomize(None)
111         for key in self.keys:
112             if self._do_transform:
113                 meta = None
114                 if get_track_meta():
115                     meta = d[key].meta
116                 dtype = d[key].dtype
117                 shape = d[key].shape
118                 assert len(shape) == 4, "img should be 4D array, (c, w, h, d)"
119
120                 img = np.array(d[key])
121
122                 img = augment_gaussian_blur(data_sample=img,
123                                           sigma_range=self.blur_sigma,
124                                           per_channel=self.per_channel,
125                                           p_per_channel=self.prob_per_channel,
126                                           )
127
128                 if meta:
129                     img = MetaTensor(img, meta=meta)
130                 d[key] = convert_data_type(img, dtype=dtype)[0]
131         return d

```

```

0 SparrowLink/transform/SpatialTransformationD.py
1 import numpy as np
2 import torch
3 from monai.utils import TransformBackends
4 from typing import Any, Callable, Dict, Hashable, List, Mapping, Optional, Sequence, Union
5 from monai.config import IndexSelection, KeysCollection, SequenceStr, NdarrayOrTensor, DtypeLike, KeysCollection
6 from monai.data.meta_obj import get_track_meta
7 from monai.utils.type_conversion import convert_data_type, convert_to_dst_type, convert_to_tensor, get_equivalent_dtype
8 from monai.transforms.utils_pytorch_numpy_unification import clip
9 from monai.data.meta_tensor import MetaTensor
10 from monai.metrics.utils import get_surface_distance, get_mask_edges
11 from monai.transforms import MapTransform, RandomizableTransform, Randomizable, InvertibleTransform
12
13 from batchgenerators.augmentations.noise_augmentations import augment_gaussian_blur, augment_gaussian_noise, \
14     augment_rician_noise
15 from batchgenerators.augmentations.color_augmentations import augment_contrast, augment_brightness_additive, \
16     augment_brightness_multiplicative, augment_gamma, augment_illumination, augment_PCA_shift
17 from batchgenerators.augmentations.resample_augmentations import augment_linear_downsampling_scipy
18 from typing import Any, Callable, Dict, Hashable, List, Mapping, Optional, Sequence, Union, Tuple
19
20
21 class Permuted(MapTransform, InvertibleTransform):
22     """
23     Dictionary-based wrapper of :py:class:`monai.transforms.Orientation`.
24
25     This transform assumes the channel-first input format.
26     In the case of using this transform for normalizing the orientations of images,
27     it should be used before any anisotropic spatial transforms.
28     """
29
30     backend = [TransformBackends.NUMPY, TransformBackends.TORCH]
31
32     def __init__(
33         self,
34         keys: KeysCollection,
35         axcodes: Optional[str] = None,
36         as_closest_canonical: bool = False,
37         labels: Optional[Sequence[Tuple[str, str]]] = (("L", "R"), ("P", "A"), ("I", "S")),
38         meta_keys: Optional[KeysCollection] = None,
39         meta_key_postfix: str = "meta_dict",
40         allow_missing_keys: bool = False,
41     ) -> None:
42         """
43         Args:
44             axcodes: N elements sequence for spatial ND input's orientation.
45                     e.g. axcodes='RAS' represents 3D orientation:
46                     (Left, Right), (Posterior, Anterior), (Inferior, Superior).
47                     default orientation labels options are: 'L' and 'R' for the first dimension,
48                     'P' and 'A' for the second, 'I' and 'S' for the third.
49             as_closest_canonical: if True, load the image as closest to canonical axis format.
50             labels: optional, None or sequence of (2,) sequences
51                     (2,) sequences are labels for (beginning, end) of output axis.
52                     Defaults to ``(('L', 'R'), ('P', 'A'), ('I', 'S'))``.
53             allow_missing_keys: don't raise exception if key is missing.
54
55         See Also:
56             `nibabel.orientations.ornt2axcodes`.
57
58         """
59         super().__init__(keys, allow_missing_keys)
60         self.ornt_transform = Orientation(axcodes=axcodes, as_closest_canonical=as_closest_canonical, labels=labels)
61
62     def __call__(self, data: Mapping[Hashable, torch.Tensor]) -> Dict[Hashable, torch.Tensor]:
63         d: Dict = dict(data)
64         for key in self.key_iterator(d):
65             d[key] = self.ornt_transform(d[key])
66         return d
67
68     def inverse(self, data: Mapping[Hashable, torch.Tensor]) -> Dict[Hashable, torch.Tensor]:
69         d = dict(data)
70         for key in self.key_iterator(d):
71             d[key] = self.ornt_transform.inverse(d[key])
72         return d

```



```

0 SparrowLink/transform/utils.py
1 import numpy as np
2 import torch
3 from monai.utils import TransformBackends
4 from typing import Any, Callable, Dict, Hashable, List, Mapping, Optional, Sequence, Union
5 from monai.config import IndexSelection, KeysCollection, SequenceStr, NdarrayOrTensor, DtypeLike, KeysCollection
6 from monai.data.meta_obj import get_track_meta
7 from monai.utils.type_conversion import convert_data_type, convert_to_dst_type, convert_to_tensor, get_equivalent_dtype
8 from monai.transforms.utils_pytorch_numpy_unification import clip
9 from monai.data.meta_tensor import MetaTensor
10 from monai.metrics.utils import get_surface_distance, get_mask_edges
11
12
13 from monai.transforms import (
14     EnsureChannelFirstd,
15     Compose,
16     CropForegroundd,
17     LoadImaged,
18     Orientationd,
19     RandCropByPosNegLabeld,
20     SaveImaged,
21     ScaleIntensityRanged,
22     Spacingd,
23     RandAffined,
24     MapTransform,
25     RandFlipd,
26     RandRotated,
27 )
28 from openpyxl import Workbook, load_workbook
29 import os
30 from scipy import ndimage
31 import numpy as np
32
33 from transform.IntensityTransformD import CTNormalizedD, BrightnessMultiplicativeD, ContrastAugmentationD, \
34     SimulateLowResolutionD, GammaD
35 from transform.NoiseTransformD import GaussianNoiseD, GaussianBlurD
36
37
38 class UseHeartsegDeleteInformationD(MapTransform):
39
40     backend = [TransformBackends.TORCH]
41
42     def __init__(
43         self,
44         keys: KeysCollection,
45         dilation_key: str,
46         dilation_time: int = 1,
47         dilation_struct: int = 1,
48         test_key: str = None,
49         caculate_surface_disdanc: bool = False,
50         vessel_key: str = None,
51         chamber_key: str = None,
52         save_path: str = None,
53         allow_missing_keys: bool = False,
54         **pad_kwargs,
55     ) -> None:
56         """
57         test_key: show whether your mask is reserved totally.
58         """
59         self.dilation_key = dilation_key
60         self.dilation_time = dilation_time
61         self.dilation_struct = dilation_struct
62         self.test_key = test_key
63         self.caculate_surface_disdanc = caculate_surface_disdanc
64         self.vessel_key = vessel_key
65         self.chamber_key = chamber_key
66         self.save_path = save_path
67         self.error_list = []
68         self.wb = None
69
70         if self.caculate_surface_disdanc:
71             row = ('image name', 'surface disdanc', 'error')
72             self.wb = Workbook()
73             self.excel_writer(row=row, file_name=self.save_path)
74
75         super().__init__(keys, allow_missing_keys=allow_missing_keys)
76
77     def __call__(self, data: Mapping[Hashable, torch.Tensor]) -> Dict[Hashable, torch.Tensor]:
78         d = dict(data)
79         region = convert_to_tensor(data=d[self.dilation_key])
80         if self.dilation_time > 1:
81             region = region.clip(min=0, max=1) > 0
82             region = self.dilation(x=region) # bool
83
84         distance = -1
85         error = 0
86         for key in self.key_iterator(d):
87             if self.caculate_surface_disdanc:
88                 vessel_edge, chamber_edge = get_mask_edges(d[self.vessel_key], d[self.chamber_key] > 0)
89                 distance = get_surface_distance(vessel_edge, chamber_edge)
90                 distance = distance.max()
91
92             if key == self.dilation_key:
93                 d[key] = MetaTensor(region, meta=d[key].meta)
94             else:
95                 # record the img information if label has some voxels out of dilation region
96                 if key == self.test_key and (d[key] * (~region)).sum() > 0:
97                     self.error_list.append(d[key].meta['filename_or_obj'])
98                     error = 1
99                     print("-----error crop-----")
100                     print(d[key].meta['filename_or_obj'])
101
102                 d[key] = d[key] * region
103
104         # record image dilation information in Excel
105         if self.wb:
106             row = (os.path.split(d[self.test_key].meta['filename_or_obj'])[-1], distance, error)
107             self.excel_writer(row=row, file_name=self.save_path)
108
109     else:

```

```

110         # if you choose to set dilation before training process
111         region = region.clip(min=0, max=1) > 0
112         for key in self.key_iterator(d):
113             if key != self.dilation_key:
114                 d[key] = d[key] * region
115         return d
116
117     def excel_writer(self, row, file_name):
118         ws = self.wb.active
119         ws.append(row)
120         self.wb.save(file_name)
121
122     def dilation(self, x: torch.Tensor):
123         shape = x.shape
124         if len(shape) > 3:
125             x = x.squeeze()
126             x = np.array(x.squeeze())
127             struct1 = ndimage.generate_binary_structure(3, self.dilation_struct)
128             x = ndimage.binary_dilation(x, structure=struct1, iterations=self.dilation_time).astype(x.dtype)
129         if len(shape) > 3:
130             return torch.tensor(x).unsqueeze(dim=0)
131         else:
132             return torch.tensor(x)
133
134
135 class DilationD(MapTransform):
136
137     backend = [TransformBackends.TORCH]
138
139     def __init__(
140         self,
141         keys: KeysCollection,
142         dilation_time: int = 1,
143         dilation_struct: int = 1,
144         allow_missing_keys: bool = False,
145         **pad_kwargs,
146     ) -> None:
147         """
148         test_key: show whether your mask is reserved totally.
149         """
150         self.dilation_time = dilation_time
151         self.dilation_struct = dilation_struct
152         super().__init__(keys, allow_missing_keys=allow_missing_keys)
153
154     def __call__(self, data: Mapping[Hashable, torch.Tensor]) -> Dict[Hashable, torch.Tensor]:
155         d = dict(data)
156         for key in self.keys:
157             region = convert_to_tensor(data=d[key])
158             region = self.dilation(region)
159             d[key] = MetaTensor(region, meta=d[key].meta)
160         return d
161
162     def dilation(self, x: torch.Tensor):
163         shape = x.shape
164         if len(shape) > 3:
165             x = x.squeeze()
166             x = np.array(x.squeeze())
167             struct1 = ndimage.generate_binary_structure(3, self.dilation_struct)
168             x = ndimage.binary_dilation(x, structure=struct1, iterations=self.dilation_time).astype(x.dtype)
169         if len(shape) > 3:
170             return torch.tensor(x).unsqueeze(dim=0)
171         else:
172             return torch.tensor(x)
173
174
175     def get_transform(transform_dic, mode='train'):
176         """dic is transform block of config.yaml"""
177         """
178         only patch_size and normalization is controlled by config.yaml
179         """
180         if transform_dic.get('use_config') is True:
181             return get_config_transform(transform_dic, mode=mode)
182         else:
183             return get_default_transform(transform_dic, mode=mode)
184
185
186     def get_default_transform(transform_dic, mode='train'):
187         """dic is transform block of config.yaml"""
188         """
189         only patch_size and normalization is controlled by config.yaml
190         """
191         if mode == 'train':
192             train_transforms = Compose(
193                 [
194                     LoadImaged(keys=["image", "label"], image_only=False),
195                     EnsureChannelFirstd(keys=["image", "label"]),
196                     ScaleIntensityRanged(
197                         keys=["image"], a_min=-57, a_max=400,
198                         b_min=0.0, b_max=1.0, clip=True,
199                     ),
200                     CTNormalizedD(keys=["image"],
201                                   mean_intensity=transform_dic["normalize"]["mean"],
202                                   std_intensity=transform_dic["normalize"]["std"],
203                                   lower_bound=transform_dic["normalize"]["min"],
204                                   upper_bound=transform_dic["normalize"]["max"], ),
205                     CropForegroundd(keys=["image", "label"], source_key="image"),
206                     Orientationd(keys=["image", "label"], axcodes="RAI"),
207                     # 记得改
208                     Spacingd(keys=["image", "label"], pixdim=transform_dic["spacing"]
209                               , mode=("bilinear", "nearest")),
210                     RandRotated(
211                         keys=["image", "label"],
212                         range_x=np.pi / 6,
213                         range_y=np.pi / 6,
214                         range_z=np.pi / 6,
215                         prob=1,
216                         padding_mode="zeros",
217                         mode=("bilinear", "nearest")
218                     ),
219                     RandCropByPosNegLabeld(
220                         keys=["image", "label"],
221                         label_key="label",
222

```

```

222         spatial_size=tuple(transform_dic["patch_size"]),
223         pos=3,
224         neg=1,
225         num_samples=4,
226         image_key="image",
227         image_threshold=0,
228     ),
229     GaussianNoised(keys=["image"], noise_variance=(0, 0.1), prob=0.1),
230     GaussianBlurD(keys=["image"], blur_sigma=(0.5, 1.15), prob=0.1),
231     BrightnessMultiplicativeD(keys=["image"], prob=0.15, multiplier_range=(0.7, 1.25)),
232     ContrastAugmentationD(keys=["image"], prob=0.15),
233     SimulateLowResolutionD(keys=["image"], zoom_range=(0.5, 1), prob=0.20,
234         order_upsample=3, order_downsample=0, ignore_axes=None),
235     GammaD(keys=["image"], gamma_range=(0.7, 1.5), invert_image=True,
236         per_channel=True, retain_stats=True, prob=0.1),
237     GammaD(keys=["image"], gamma_range=(0.7, 1.5), invert_image=True,
238         per_channel=True, retain_stats=True, prob=0.3),
239     RandFlipD(keys=["image", "label"], prob=0.5, spatial_axis=0),
240     RandFlipD(keys=["image", "label"], prob=0.5, spatial_axis=1),
241     RandFlipD(keys=["image", "label"], prob=0.5, spatial_axis=2),
242     ]
243 )
244 val_transforms = Compose(
245     [
246         LoadImaged(keys=["image", "label"], image_only=False),
247         EnsureChannelFirstD(keys=["image", "label"]),
248         # ScaleIntensityRanged(
249         #     keys=["image"], a_min=-57, a_max=400,
250         #     b_min=0.0, b_max=1.0, clip=True,
251         # ),
252         CTNormalizedD(keys=["image"],
253             mean_intensity=transform_dic["normalize"]["mean"],
254             std_intensity=transform_dic["normalize"]["std"],
255             lower_bound=transform_dic["normalize"]["min"],
256             upper_bound=transform_dic["normalize"]["max"], ),
257         CropForegroundD(keys=["image", "label"], source_key="image"), # default: value>0
258         OrientationD(keys=["image", "label"], axcodes="RAI"),
259         SpacingD(keys=["image", "label"], pixdim=transform_dic["spacing"],
260             mode=("bilinear", "nearest")), # Now only Sequential_str, How to add spline
261     ]
262 )
263 save_transform = Compose(
264     [
265         LoadImaged(keys=["image", "label"], image_only=False),
266         EnsureChannelFirstD(keys=["image", "label"]),
267         SaveImaged(keys=["image"], output_dir='./transform/test_transform', output_postfix='origin_image',
268             print_log=False),
269         SaveImaged(keys=["label"], output_dir='./transform/test_transform', output_postfix='origin_label',
270             print_log=False),
271         CTNormalizedD(keys=["image"],
272             mean_intensity=transform_dic["normalize"]["mean"],
273             std_intensity=transform_dic["normalize"]["std"],
274             lower_bound=transform_dic["normalize"]["min"],
275             upper_bound=transform_dic["normalize"]["max"], ),
276         CropForegroundD(keys=["image", "label"], source_key="image"),
277         OrientationD(keys=["image", "label"], axcodes="RAI"),
278         SpacingD(keys=["image", "label"], pixdim=transform_dic["spacing"],
279             mode=("bilinear", "nearest")),
280         GaussianNoised(keys=["image"], noise_variance=(0, 0.1), prob=1.0),
281         GaussianBlurD(keys=["image"], blur_sigma=(0.5, 1.15), prob=1.0),
282         SaveImaged(keys=["image"], output_dir='./transform/test_transform', output_postfix='tran_image',
283             print_log=False),
284         SaveImaged(keys=["label"], output_dir='./transform/test_transform', output_postfix='tran_label',
285             print_log=False),
286     ]
287 )
288 return train_transforms, val_transforms, save_transform
289 elif mode == 'infer':
290     infer_transforms = Compose(
291         [
292             LoadImaged(keys=["image"], image_only=False),
293             EnsureChannelFirstD(keys=["image"]),
294             # ScaleIntensityRanged(
295             #     keys=["image"], a_min=-57, a_max=400,
296             #     b_min=0.0, b_max=1.0, clip=True,
297             # ),
298             CTNormalizedD(keys=["image"],
299                 mean_intensity=transform_dic["normalize"]["mean"],
300                 std_intensity=transform_dic["normalize"]["std"],
301                 lower_bound=transform_dic["normalize"]["min"],
302                 upper_bound=transform_dic["normalize"]["max"], ),
303             # CropForegroundD(keys=["image"], source_key="image"), # default: value>0
304             OrientationD(keys=["image"], axcodes="RAI"),
305             SpacingD(keys=["image"], pixdim=transform_dic["spacing"], mode=("bilinear",)), # Now only Sequential_str, How to add spline
306         ]
307     )
308     return infer_transforms
309 else:
310     raise RuntimeError(f"{mode} is not supported yet")
311
312
313 def get_config_transform(transform_dic, mode='train'):
314     """dic is transform block of config.yaml"""
315     """
316     you can further control your random transformation
317     """
318     if transform_dic.get("image_resample") is not None:
319         image_resample_mode = transform_dic["image_resample"]["mode"]
320         image_resample_padding_mode = transform_dic["image_resample"]["padding_mode"]
321     else:
322         image_resample_mode = "bilinear"
323         image_resample_padding_mode = "border"
324
325     if mode == 'train':
326         fixed_transforms = [
327             LoadImaged(keys=["image", "label"]),
328             EnsureChannelFirstD(keys=["image", "label"]),
329             CTNormalizedD(keys=["image"],
330                 mean_intensity=transform_dic["normalize"]["mean"],
331                 std_intensity=transform_dic["normalize"]["std"],
332                 lower_bound=transform_dic["normalize"]["min"],
333                 upper_bound=transform_dic["normalize"]["max"], ),
334             CropForegroundD(keys=["image", "label"], source_key="image"),
335             OrientationD(keys=["image", "label"], axcodes="RAI"),
336             SpacingD(keys=["image", "label"], pixdim=transform_dic["spacing"], mode=("bilinear", "nearest")),
337             # Now only Sequential_str, How to add spline
338         ]

```

```

334 cropForeground(keys=["image", "label"], source_key="image"),
335 Orientationd(keys=["image", "label"], axcodes="RAI"),
336 # 记得改
337 Spacingd(keys=["image", "label"], pixdim=transform_dic["spacing"])
338 , mode=(image_resample_mode, "nearest"), padding_mode=(image_resample_padding_mode, "border")),
339 ]
340 print("-----training_fixed_transform-----")
341 if transform_dic.get("fixed") is not None and len(transform_dic.get("fixed")) > 0:
342     for d in transform_dic["fixed"]:
343         key = d.get('name')
344         value = d.get('parameter')
345         assert key is not None, "name of fixed transform is not defined in config.yaml"
346         assert value is not None, "parameter of fixed transform is not defined in config.yaml"
347         print(f"name: {key}, parameter: {value}")
348         trans_class = eval(key)
349         trans_module = trans_class(**value)
350         fixed_transforms.append(trans_module)
351
352 random_transforms = []
353 # assert transform_dic.get("random") is not None, "random transform is not defined in config.yaml"
354 if transform_dic.get("random") is None or len(transform_dic.get("random")) == 0:
355     print("warning: random transform is not defined in config.yaml")
356 else:
357     print("-----training_random_transform-----")
358     for d in transform_dic["random"]:
359         key = d.get('name')
360         value = d.get('parameter')
361         assert key is not None, "name of random transform is not defined in config.yaml"
362         assert value is not None, "parameter of random transform is not defined in config.yaml"
363         print(f"name: {key}, parameter: {value}")
364         trans_class = eval(key)
365         trans_module = trans_class(**value)
366         random_transforms.append(trans_module)
367
368 train_transforms = Compose(fixed_transforms + random_transforms)
369 val_transforms = Compose(fixed_transforms)
370 save_transform = None
371 return train_transforms, val_transforms, save_transform
372 elif mode == 'infer':
373     infer_transforms = [
374         LoadImaged(keys=["image"]),
375         EnsureChannelFirstd(keys=["image"]),
376         CTNormalizedD(keys=["image"],
377                        mean_intensity=transform_dic["normalize"]["mean"],
378                        std_intensity=transform_dic["normalize"]["std"],
379                        lower_bound=transform_dic["normalize"]["min"],
380                        upper_bound=transform_dic["normalize"]["max"], ),
381         CropForegroundd(keys=["image"], source_key="image"), # default: value>0
382         Orientationd(keys=["image"], axcodes="RAI"),
383         Spacingd(keys=["image"], pixdim=transform_dic["spacing"], mode=image_resample_mode,
384                  padding_mode=image_resample_padding_mode),
385     ]
386     if transform_dic.get("infer") is not None and len(transform_dic.get("infer")) > 0:
387         for d in transform_dic["infer"]:
388             key = d.get('name')
389             value = d.get('parameter')
390             assert key is not None, "name of infer transform is not defined in config.yaml"
391             assert value is not None, "parameter of infer transform is not defined in config.yaml"
392             trans_class = eval(key)
393             trans_module = trans_class(**value)
394             infer_transforms.append(trans_module)
395
396     infer_transforms = Compose(infer_transforms)
397     return infer_transforms
398 else:
399     raise RuntimeError(f"{mode} is not supported yet")
400
401
402 def print_config_transforms(transforms_list):
403     import inspect
404     for transforms in transforms_list:
405         print(f"name: {inspect.getmodule(transforms).__name__}, parameter: {inspect.getmembers(transforms)}")
406
407
408 def get_multi_phase_transform_with_image(transform_dic, mode='train'):
409     """dic is transform block of config.yaml"""
410     """
411     only patch_size and normalization is controlled by config.yaml
412     """
413     if transform_dic.get('use_config') is True:
414         return get_multi_phase_transform_with_image_config(transform_dic, mode=mode)
415     else:
416         return get_multi_phase_transform_with_image_default(transform_dic, mode=mode)
417
418
419 def get_multi_phase_transform_with_image_default(transform_dic, mode='train'):
420     """dic is transform block of config.yaml"""
421     if mode == 'train':
422         train_transforms = Compose(
423             [
424                 LoadImaged(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"]),
425                 EnsureChannelFirstd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"]),
426                 Orientationd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], axcodes="RAI"),
427
428                 CTNormalizedD(keys=["I_M", "I_A"],
429                              mean_intensity=transform_dic["normalize"]["mean"],
430                              std_intensity=transform_dic["normalize"]["std"],
431                              lower_bound=transform_dic["normalize"]["min"],
432                              upper_bound=transform_dic["normalize"]["max"], ),
433                 Spacingd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], pixdim=transform_dic["spacing"],
434                          mode=("nearest", "nearest", "nearest", "bilinear", "bilinear", "nearest", "nearest")),
435                 RandCropByPosNegLabeld(
436                     keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"],
437                     label_key="CS_DLGT",
438                     spatial_size=tuple(transform_dic['patch_size']),
439                     pos=1,
440                     neg=0,
441                     num_samples=2,
442                     image_key="I_M",
443                     image_threshold=0,
444                     allow_smaller=True,
445                 ),

```

```

446         # RandFlipd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], prob=0.5, spatial_axis=0),
447         # RandFlipd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], prob=0.5, spatial_axis=1),
448         # RandFlipd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], prob=0.5, spatial_axis=2),
449         # # user can also add other random transforms
450         # RandAffined(
451         #     keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"],
452         #     mode=('nearest', 'nearest', 'nearest', 'bilinear', 'bilinear', "nearest", "nearest"),
453         #     prob=0.5,
454         #     rotate_range=(0, 0, np.pi / 15),
455         #     scale_range=(0.1, 0.1, 0.1),
456         #     padding_mode="zeros"
457         # ),
458     ]
459 )
460 val_transforms = Compose(
461     [
462         LoadImaged(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"]),
463         EnsureChannelFirstd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"]),
464         Orientationd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], axcodes="RAI"),
465         CTNormalizedD(keys=["I_M", "I_A"],
466             mean_intensity=transform_dic["normalize"]["mean"],
467             std_intensity=transform_dic["normalize"]["std"],
468             lower_bound=transform_dic["normalize"]["min"],
469             upper_bound=transform_dic["normalize"]["max"], ),
470         Spacingd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], pixdim=transform_dic["spacing"],
471             mode=("nearest", "nearest", "nearest", "bilinear", "bilinear", "nearest", "nearest")),
472     ]
473 )
474 save_transform = Compose(
475     [
476     ]
477 )
478 return train_transforms, val_transforms, save_transform
479 elif mode == 'infer':
480     infer_transforms = Compose(
481         [
482             LoadImaged(keys=["CS_M", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"]),
483             EnsureChannelFirstd(keys=["CS_M", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"]),
484             Orientationd(keys=["CS_M", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], axcodes="RAI"),
485             CTNormalizedD(keys=["I_M", "I_A"],
486                 mean_intensity=transform_dic["normalize"]["mean"],
487                 std_intensity=transform_dic["normalize"]["std"],
488                 lower_bound=transform_dic["normalize"]["min"],
489                 upper_bound=transform_dic["normalize"]["max"], ),
490             Spacingd(keys=["CS_M", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], pixdim=transform_dic["spacing"],
491                 mode=("nearest", "nearest", "bilinear", "bilinear", "nearest", "nearest")),
492         ]
493     )
494     return infer_transforms
495 else:
496     raise RuntimeError(f"{mode} is not supported yet")
497
498 def get_multi_phase_transform_with_image_config(transform_dic, mode='train'):
499     """dic is transform block of config.yaml"""
500     if transform_dic.get("image_resample") is not None:
501         image_resample_mode = transform_dic["image_resample"]["mode"]
502         image_resample_padding_mode = transform_dic["image_resample"]["padding_mode"]
503     else:
504         image_resample_mode = "bilinear"
505         image_resample_padding_mode = "border"
506     if mode == 'train':
507         fixed_transforms = [
508             LoadImaged(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"]),
509             EnsureChannelFirstd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"]),
510             Orientationd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], axcodes="RAI"),
511             CTNormalizedD(keys=["I_M", "I_A"],
512                 mean_intensity=transform_dic["normalize"]["mean"],
513                 std_intensity=transform_dic["normalize"]["std"],
514                 lower_bound=transform_dic["normalize"]["min"],
515                 upper_bound=transform_dic["normalize"]["max"], ),
516             Spacingd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], pixdim=transform_dic["spacing"],
517                 mode=("nearest", "nearest", "nearest", image_resample_mode, image_resample_mode, "nearest", "nearest")),
518         ]
519     print("-----second-stage fixed transform-----")
520     if transform_dic.get("fixed") is not None and len(transform_dic.get("fixed")) > 0:
521         for d in transform_dic["fixed"]:
522             key = d.get('name')
523             value = d.get('parameter')
524             assert key is not None, "name of fixed transform is not defined in config.yaml"
525             assert value is not None, "parameter of fixed transform is not defined in config.yaml"
526             print(f"name: {key}, parameter: {value}")
527             trans_class = eval(key)
528             trans_module = trans_class(**value)
529             fixed_transforms.append(trans_module)
530
531     random_transforms = []
532     # assert transform_dic.get("random") is not None, "random transform is not defined in config.yaml"
533     if transform_dic.get("random") is None or len(transform_dic.get("random")) == 0:
534         print("warning: random transform is not defined in config.yaml")
535     else:
536         print("-----second-stage random transform-----")
537         for d in transform_dic["random"]:
538             key = d.get('name')
539             value = d.get('parameter')
540             assert key is not None, "name of random transform is not defined in config.yaml"
541             assert value is not None, "parameter of random transform is not defined in config.yaml"
542             print(f"name: {key}, parameter: {value}")
543             trans_class = eval(key)
544             trans_module = trans_class(**value)
545             random_transforms.append(trans_module)
546
547     train_transforms = Compose(fixed_transforms + random_transforms)
548     val_transforms = Compose(fixed_transforms)
549     save_transform = None
550     return train_transforms, val_transforms, save_transform
551 elif mode == 'infer':
552     infer_transforms = [
553         LoadImaged(keys=["CS_M", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"]),
554         EnsureChannelFirstd(keys=["CS_M", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"]),
555         Orientationd(keys=["CS_M", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], axcodes="RAI"),
556         CTNormalizedD(keys=["I_M", "I_A"],

```

```

558         mean_intensity=transform_dic["normalize"]["mean"],
559         std_intensity=transform_dic["normalize"]["std"],
560         lower_bound=transform_dic["normalize"]["min"],
561         upper_bound=transform_dic["normalize"]["max"], ),
562         Spacingd(keys=["CS_M", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], pixdim=transform_dic["spacing"],
563         mode=("nearest", "nearest", image_resample_mode, image_resample_mode, "nearest", "nearest")),
564     ]
565     print("-----second-stage infer_transform-----")
566     if transform_dic.get("infer") is not None and len(transform_dic.get("infer")) > 0:
567         for d in transform_dic["infer"]:
568             key = d.get('name')
569             value = d.get('parameter')
570             assert key is not None, "name of infer transform is not defined in config.yaml"
571             assert value is not None, "parameter of infer transform is not defined in config.yaml"
572             print(f"name: {key}, parameter: {value}")
573             trans_class = eval(key)
574             trans_module = trans_class(**value)
575             infer_transforms.append(trans_module)
576
577     infer_transforms = Compose(infer_transforms)
578     return infer_transforms
579 else:
580     raise RuntimeError(f"{mode} is not supported yet")
581
582
583 def get_second_stage_only_one_phase(transform_dic, mode='train'):
584     """dic is transform block of config.yaml"""
585     """
586     only patch_size and normalization is controlled by config.yaml
587     """
588     if transform_dic.get('use_config') is True:
589         return get_second_stage_only_one_phase_default(transform_dic, mode=mode)
590     else:
591         return get_second_stage_only_one_phase_config(transform_dic, mode=mode)
592
593
594 def get_second_stage_only_one_phase_default(transform_dic, mode='train'):
595     """dic is transform block of config.yaml"""
596     if mode == 'train':
597         train_transforms = Compose(
598             [
599                 LoadImaged(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"]),
600                 EnsureChannelFirstd(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"]),
601                 Orientationd(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"], axcodes="RAI"),
602
603                 CTNormalizedD(keys=["I_M"],
604                             mean_intensity=transform_dic["normalize"]["mean"],
605                             std_intensity=transform_dic["normalize"]["std"],
606                             lower_bound=transform_dic["normalize"]["min"],
607                             upper_bound=transform_dic["normalize"]["max"], ),
608                 Spacingd(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"], pixdim=transform_dic['spacing'],
609                 , mode=("nearest", "nearest", "bilinear", "nearest", "nearest")),
610                 RandCropByPosNegLabeld(
611                     keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"],
612                     label_key="CS_DLGT",
613                     spatial_size=tuple(transform_dic['patch_size']),
614                     pos=1,
615                     neg=0,
616                     num_samples=2,
617                     image_key="I_M",
618                     image_threshold=0,
619                     allow_smaller=True,
620                 ),
621                 # RandFlipd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], prob=0.5, spatial_axis=0),
622                 # RandFlipd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], prob=0.5, spatial_axis=1),
623                 # RandFlipd(keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"], prob=0.5, spatial_axis=2),
624                 # # user can also add other random transforms
625                 # RandAffined(
626                 #     keys=["CS_M", "label", "CS_A", "I_M", "I_A", "CS_DL", "CS_DLGT"],
627                 #     mode=('nearest', 'nearest', 'nearest', 'bilinear', 'bilinear', "nearest", "nearest"),
628                 #     prob=0.5,
629                 #     rotate_range=(0, 0, np.pi / 15),
630                 #     scale_range=(0.1, 0.1, 0.1),
631                 #     padding_mode="zeros"
632                 # ),
633             ]
634         )
635         val_transforms = Compose(
636             [
637                 LoadImaged(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"]),
638                 EnsureChannelFirstd(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"]),
639                 Orientationd(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"], axcodes="RAI"),
640                 CTNormalizedD(keys=["I_M"],
641                             mean_intensity=transform_dic["normalize"]["mean"],
642                             std_intensity=transform_dic["normalize"]["std"],
643                             lower_bound=transform_dic["normalize"]["min"],
644                             upper_bound=transform_dic["normalize"]["max"], ),
645                 Spacingd(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"], pixdim=transform_dic["spacing"],
646                 , mode=("nearest", "nearest", "bilinear", "nearest", "nearest")),
647             ]
648         )
649         save_transform = Compose(
650             [
651                 ]
652         )
653         return train_transforms, val_transforms, save_transform
654     elif mode == 'infer':
655         infer_transforms = Compose(
656             [
657                 LoadImaged(keys=["CS_M", "I_M", "CS_DL", "CS_DLGT"]),
658                 EnsureChannelFirstd(keys=["CS_M", "I_M", "CS_DL", "CS_DLGT"]),
659                 Orientationd(keys=["CS_M", "I_M", "CS_DL", "CS_DLGT"], axcodes="RAI"),
660                 CTNormalizedD(keys=["I_M"],
661                             mean_intensity=transform_dic["normalize"]["mean"],
662                             std_intensity=transform_dic["normalize"]["std"],
663                             lower_bound=transform_dic["normalize"]["min"],
664                             upper_bound=transform_dic["normalize"]["max"], ),
665                 Spacingd(keys=["CS_M", "I_M", "CS_DL", "CS_DLGT"], pixdim=transform_dic["spacing"],
666                 , mode=("nearest", "bilinear", "nearest", "nearest")),
667             ]
668         )
669         return infer_transforms

```



```

670     else:
671         raise RuntimeError(f"{mode} is not supported yet")
672
673
674 def get_second_stage_only_one_phase_config(transform_dic, mode='train'):
675     """dic is transform block of config.yaml"""
676     if transform_dic.get("image_resample") is not None:
677         image_resample_mode = transform_dic["image_resample"]["mode"]
678         image_resample_padding_mode = transform_dic["image_resample"]["padding_mode"]
679     else:
680         image_resample_mode = "bilinear"
681         image_resample_padding_mode = "border"
682     if mode == 'train':
683         fixed_transforms = [
684             LoadImaged(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"]),
685             EnsureChannelFirstd(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"]),
686             Orientationd(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"], axcodes="RAI"),
687             CTNormalized(keys=["I_M"],
688                 mean_intensity=transform_dic["normalize"]["mean"],
689                 std_intensity=transform_dic["normalize"]["std"],
690                 lower_bound=transform_dic["normalize"]["min"],
691                 upper_bound=transform_dic["normalize"]["max"], ),
692             Spacingd(keys=["CS_M", "label", "I_M", "CS_DL", "CS_DLGT"], pixdim=transform_dic["spacing"],
693                 , mode=("nearest", "nearest", image_resample_mode, "nearest", "nearest")),
694         ]
695     print("-----second-stage fixed transform-----")
696     if transform_dic.get("fixed") is not None and len(transform_dic.get("fixed")) > 0:
697         for d in transform_dic["fixed"]:
698             key = d.get('name')
699             value = d.get('parameter')
700             assert key is not None, "name of fixed transform is not defined in config.yaml"
701             assert value is not None, "parameter of fixed transform is not defined in config.yaml"
702             print(f"name: {key}, parameter: {value}")
703             trans_class = eval(key)
704             trans_module = trans_class(**value)
705             fixed_transforms.append(trans_module)
706
707     random_transforms = []
708     # assert transform_dic.get("random") is not None, "random transform is not defined in config.yaml"
709     if transform_dic.get("random") is None or len(transform_dic.get("random")) == 0:
710         print("warning: random transform is not defined in config.yaml")
711     else:
712         print("-----second-stage random transform-----")
713         for d in transform_dic["random"]:
714             key = d.get('name')
715             value = d.get('parameter')
716             assert key is not None, "name of random transform is not defined in config.yaml"
717             assert value is not None, "parameter of random transform is not defined in config.yaml"
718             print(f"name: {key}, parameter: {value}")
719             trans_class = eval(key)
720             trans_module = trans_class(**value)
721             random_transforms.append(trans_module)
722
723     train_transforms = Compose(fixed_transforms + random_transforms)
724     val_transforms = Compose(fixed_transforms)
725     save_transform = None
726     return train_transforms, val_transforms, save_transform
727 elif mode == 'infer':
728     infer_transforms = [
729         LoadImaged(keys=["CS_M", "I_M", "CS_DL", "CS_DLGT"]),
730         EnsureChannelFirstd(keys=["CS_M", "I_M", "CS_DL", "CS_DLGT"]),
731         Orientationd(keys=["CS_M", "I_M", "CS_DL", "CS_DLGT"], axcodes="RAI"),
732         CTNormalized(keys=["I_M"],
733             mean_intensity=transform_dic["normalize"]["mean"],
734             std_intensity=transform_dic["normalize"]["std"],
735             lower_bound=transform_dic["normalize"]["min"],
736             upper_bound=transform_dic["normalize"]["max"], ),
737         Spacingd(keys=["CS_M", "I_M", "CS_DL", "CS_DLGT"], pixdim=transform_dic["spacing"],
738             mode=("nearest", image_resample_mode, "nearest")),
739     ]
740     print("-----second-stage infer transform-----")
741     if transform_dic.get("infer") is not None and len(transform_dic.get("infer")) > 0:
742         for d in transform_dic["infer"]:
743             key = d.get('name')
744             value = d.get('parameter')
745             assert key is not None, "name of infer transform is not defined in config.yaml"
746             assert value is not None, "parameter of infer transform is not defined in config.yaml"
747             print(f"name: {key}, parameter: {value}")
748             trans_class = eval(key)
749             trans_module = trans_class(**value)
750             infer_transforms.append(trans_module)
751
752     infer_transforms = Compose(infer_transforms)
753     return infer_transforms
754 else:
755     raise RuntimeError(f"{mode} is not supported yet")

```

tree.txt

```
0 SparrowLink/
1 |─ ablation_nnunet.sh
2 |─ caculate_metric.py
3 |─ configs/
4 |   |─ two_stage2/
5 |   |   |─ first_stage/
6 |   |   |   |─ ResUnet_dice_with_random.yaml
7 |   |   |   |─ second_stage/
8 |   |   |   |   |─ nnunetv2.yaml
9 |   |   |   |   |─ ResUnet_dice_with_random.yaml
10 |   |   |   |   |─ ResUnet_dice_with_random_1.yaml
11 |─ data/
12 |   |─ loader.py
13 |─ loss_zoo/
14 |   |─ cldice.py
15 |   |─ soft_skeleton.py
16 |─ main.py
17 |─ model/
18 |   |─ CS2net.py
19 |   |─ denseunet_skip.py
20 |   |─ swin_unet.py
21 |─ modify_key_in_config.py
22 |─ post_processing/
23 |   |─ fracture_detection.py
24 |   |─ move_file.py
25 |   |─ select_two_region.py
26 |─ README.md
27 |─ registration/
28 |   |─ basic_registration.py
29 |   |─ label_registration.py
30 |   |─ SparrowLinkv3.sh
31 |   |─ SparrowLinkv3_nnUnet.sh
32 |─ second_stage_main.py
33 |─ second_stage_only_one_phase_main.py
34 |─ slicer_visulization/
35 |   |─ slicer_mark_up.py
36 |   |─ slicer_mvf.py
37 |─ SparrowLink_metric.py
38 |─ SparrowLink_Post_Process.py
39 |─ SparrowLinkv3.sh
40 |─ SparrowLinkv3_nnUnet.sh
41 |─ SparrowLinkv3_nnUnet_ASOCA.sh
42 |─ transform/
43 |   |─ Anatomical_augumentation_for_CCTA_images.md
44 |   |─ AnatomyTransformD.py
45 |   |─ cardiac_transformation.py
46 |   |─ fast_crop.py
47 |   |─ IntensityTransformD.py
48 |   |─ NoiseTransformD.py
49 |   |─ SpatialTransformationD.py
50 |   |─ test_contrast_reduce.ipynb
51 |   |─ test_motion_artifact.ipynb
52 |   |─ try_HeartTransformD.ipynb
53 |   |─ utils.py
54 |─ utils/
55 |   |─ Config.py
56 |   |─ get_module.py
57 |   |─ get_network_from_plans_nnunet.py
58 |   |─ inferer.py
59 |   |─ test.py
60 |   |─ trainer.py
```