from collections import OrderedDict

from copy import deepcopy

from batchgenerators.augmentations.utils import resize\_segmentation

from nnformer.configuration import default\_num\_threads, RESAMPLING\_SEPARATE\_Z\_ANISO\_THRESHOLD

from nnformer.preprocessing.cropping import get\_case\_identifier\_from\_npz, ImageCropper

from skimage.transform import resize

from scipy.ndimage.interpolation import map\_coordinates

import numpy as np

from batchgenerators.utilities.file\_and\_folder\_operations import \*

from multiprocessing.pool import Pool

def get\_do\_separate\_z(spacing, anisotropy\_threshold=RESAMPLING\_SEPARATE\_Z\_ANISO\_THRESHOLD):

do\_separate\_z = (np.max(spacing) / np.min(spacing)) > anisotropy\_threshold

return do\_separate\_z

def get\_lowres\_axis(new\_spacing):

axis = np.where(max(new\_spacing) / np.array(new\_spacing) == 1)[0] # find which axis is anisotropic

return axis

def resample\_patient(data, seg, original\_spacing, target\_spacing, order\_data=3, order\_seg=0, force\_separate\_z=False,

cval\_data=0, cval\_seg=-1, order\_z\_data=0, order\_z\_seg=0,

separate\_z\_anisotropy\_threshold=RESAMPLING\_SEPARATE\_Z\_ANISO\_THRESHOLD):

"""

:param cval\_seg:

:param cval\_data:

:param data:

:param seg:

:param original\_spacing:

:param target\_spacing:

:param order\_data:

:param order\_seg:

:param force\_separate\_z: if None then we dynamically decide how to resample along z, if True/False then always

/never resample along z separately

:param order\_z\_seg: only applies if do\_separate\_z is True

:param order\_z\_data: only applies if do\_separate\_z is True

:param separate\_z\_anisotropy\_threshold: if max\_spacing > separate\_z\_anisotropy\_threshold \* min\_spacing (per axis)

then resample along lowres axis with order\_z\_data/order\_z\_seg instead of order\_data/order\_seg

:return:

"""

assert not ((data is None) and (seg is None))

if data is not None:

assert len(data.shape) == 4, "data must be c x y z"

if seg is not None:

assert len(seg.shape) == 4, "seg must be c x y z"

if data is not None:

shape = np.array(data[0].shape)

else:

shape = np.array(seg[0].shape)

new\_shape = np.round(((np.array(original\_spacing) / np.array(target\_spacing)).astype(float) \* shape)).astype(int)

if force\_separate\_z is not None:

do\_separate\_z = force\_separate\_z

if force\_separate\_z:

axis = get\_lowres\_axis(original\_spacing)

else:

axis = None

else:

if get\_do\_separate\_z(original\_spacing, separate\_z\_anisotropy\_threshold):

do\_separate\_z = True

axis = get\_lowres\_axis(original\_spacing)

elif get\_do\_separate\_z(target\_spacing, separate\_z\_anisotropy\_threshold):

do\_separate\_z = True

axis = get\_lowres\_axis(target\_spacing)

else:

do\_separate\_z = False

axis = None

if axis is not None:

if len(axis) == 3:

# every axis has the spacing, this should never happen, why is this code here?

do\_separate\_z = False

elif len(axis) == 2:

# this happens for spacings like (0.24, 1.25, 1.25) for example. In that case we do not want to resample

# separately in the out of plane axis

do\_separate\_z = False

else:

pass

if data is not None:

data\_reshaped = resample\_data\_or\_seg(data, new\_shape, False, axis, order\_data, do\_separate\_z, cval=cval\_data,

order\_z=order\_z\_data)

else:

data\_reshaped = None

if seg is not None:

seg\_reshaped = resample\_data\_or\_seg(seg, new\_shape, True, axis, order\_seg, do\_separate\_z, cval=cval\_seg,

order\_z=order\_z\_seg)

else:

seg\_reshaped = None

return data\_reshaped, seg\_reshaped

def resample\_data\_or\_seg(data, new\_shape, is\_seg, axis=None, order=3, do\_separate\_z=False, cval=0, order\_z=0):

"""

separate\_z=True will resample with order 0 along z

:param data:

:param new\_shape:

:param is\_seg:

:param axis:

:param order:

:param do\_separate\_z:

:param cval:

:param order\_z: only applies if do\_separate\_z is True

:return:

"""

assert len(data.shape) == 4, "data must be (c, x, y, z)"

if is\_seg:

resize\_fn = resize\_segmentation

kwargs = OrderedDict()

else:

resize\_fn = resize

kwargs = {'mode': 'edge', 'anti\_aliasing': False}

dtype\_data = data.dtype

data = data.astype(float)

shape = np.array(data[0].shape)

new\_shape = np.array(new\_shape)

if np.any(shape != new\_shape):

if do\_separate\_z:

print("separate z, order in z is", order\_z, "order inplane is", order)

assert len(axis) == 1, "only one anisotropic axis supported"

axis = axis[0]

if axis == 0:

new\_shape\_2d = new\_shape[1:]

elif axis == 1:

new\_shape\_2d = new\_shape[[0, 2]]

else:

new\_shape\_2d = new\_shape[:-1]

reshaped\_final\_data = []

for c in range(data.shape[0]):

reshaped\_data = []

for slice\_id in range(shape[axis]):

if axis == 0:

reshaped\_data.append(resize\_fn(data[c, slice\_id], new\_shape\_2d, order, \*\*kwargs))

elif axis == 1:

reshaped\_data.append(resize\_fn(data[c, :, slice\_id], new\_shape\_2d, order, \*\*kwargs))

else:

reshaped\_data.append(resize\_fn(data[c, :, :, slice\_id], new\_shape\_2d, order,

\*\*kwargs))

reshaped\_data = np.stack(reshaped\_data, axis)

if shape[axis] != new\_shape[axis]:

# The following few lines are blatantly copied and modified from sklearn's resize()

rows, cols, dim = new\_shape[0], new\_shape[1], new\_shape[2]

orig\_rows, orig\_cols, orig\_dim = reshaped\_data.shape

row\_scale = float(orig\_rows) / rows

col\_scale = float(orig\_cols) / cols

dim\_scale = float(orig\_dim) / dim

map\_rows, map\_cols, map\_dims = np.mgrid[:rows, :cols, :dim]

map\_rows = row\_scale \* (map\_rows + 0.5) - 0.5

map\_cols = col\_scale \* (map\_cols + 0.5) - 0.5

map\_dims = dim\_scale \* (map\_dims + 0.5) - 0.5

coord\_map = np.array([map\_rows, map\_cols, map\_dims])

if not is\_seg or order\_z == 0:

reshaped\_final\_data.append(map\_coordinates(reshaped\_data, coord\_map, order=order\_z,

mode='nearest')[None])

else:

unique\_labels = np.unique(reshaped\_data)

reshaped = np.zeros(new\_shape, dtype=dtype\_data)

for i, cl in enumerate(unique\_labels):

reshaped\_multihot = np.round(

map\_coordinates((reshaped\_data == cl).astype(float), coord\_map, order=order\_z,

mode='nearest'))

reshaped[reshaped\_multihot > 0.5] = cl

reshaped\_final\_data.append(reshaped[None])

else:

reshaped\_final\_data.append(reshaped\_data[None])

reshaped\_final\_data = np.vstack(reshaped\_final\_data)

else:

print("no separate z, order", order)

reshaped = []

for c in range(data.shape[0]):

reshaped.append(resize\_fn(data[c], new\_shape, order, \*\*kwargs)[None])

reshaped\_final\_data = np.vstack(reshaped)

return reshaped\_final\_data.astype(dtype\_data)

else:

print("no resampling necessary")

return data

class GenericPreprocessor(object):

def \_\_init\_\_(self, normalization\_scheme\_per\_modality, use\_nonzero\_mask, transpose\_forward: (tuple, list), intensityproperties=None):

"""

:param normalization\_scheme\_per\_modality: dict {0:'nonCT'}

:param use\_nonzero\_mask: {0:False}

:param intensityproperties:

"""

self.transpose\_forward = transpose\_forward

self.intensityproperties = intensityproperties

self.normalization\_scheme\_per\_modality = normalization\_scheme\_per\_modality

self.use\_nonzero\_mask = use\_nonzero\_mask

self.resample\_separate\_z\_anisotropy\_threshold = RESAMPLING\_SEPARATE\_Z\_ANISO\_THRESHOLD

@staticmethod

def load\_cropped(cropped\_output\_dir, case\_identifier):

all\_data = np.load(os.path.join(cropped\_output\_dir, "%s.npz" % case\_identifier))['data']

data = all\_data[:-1].astype(np.float32)

seg = all\_data[-1:]

with open(os.path.join(cropped\_output\_dir, "%s.pkl" % case\_identifier), 'rb') as f:

properties = pickle.load(f)

return data, seg, properties

def resample\_and\_normalize(self, data, target\_spacing, properties, seg=None, force\_separate\_z=None):

"""

data and seg must already have been transposed by transpose\_forward. properties are the un-transposed values

(spacing etc)

:param data:

:param target\_spacing:

:param properties:

:param seg:

:param force\_separate\_z:

:return:

"""

# target\_spacing is already transposed, properties["original\_spacing"] is not so we need to transpose it!

# data, seg are already transposed. Double check this using the properties

original\_spacing\_transposed = np.array(properties["original\_spacing"])[self.transpose\_forward]

before = {

'spacing': properties["original\_spacing"],

'spacing\_transposed': original\_spacing\_transposed,

'data.shape (data is transposed)': data.shape

}

# remove nans

data[np.isnan(data)] = 0

data, seg = resample\_patient(data, seg, np.array(original\_spacing\_transposed), target\_spacing, 3, 1,

force\_separate\_z=force\_separate\_z, order\_z\_data=0, order\_z\_seg=0,

separate\_z\_anisotropy\_threshold=self.resample\_separate\_z\_anisotropy\_threshold)

after = {

'spacing': target\_spacing,

'data.shape (data is resampled)': data.shape

}

print("before:", before, "\nafter: ", after, "\n")

if seg is not None: # hippocampus 243 has one voxel with -2 as label. wtf?

seg[seg < -1] = 0

properties["size\_after\_resampling"] = data[0].shape

properties["spacing\_after\_resampling"] = target\_spacing

use\_nonzero\_mask = self.use\_nonzero\_mask

assert len(self.normalization\_scheme\_per\_modality) == len(data), "self.normalization\_scheme\_per\_modality " \

"must have as many entries as data has " \

"modalities"

assert len(self.use\_nonzero\_mask) == len(data), "self.use\_nonzero\_mask must have as many entries as data" \

" has modalities"

for c in range(len(data)):

scheme = self.normalization\_scheme\_per\_modality[c]

if scheme == "CT":

# clip to lb and ub from train data foreground and use foreground mn and sd from training data

assert self.intensityproperties is not None, "ERROR: if there is a CT then we need intensity properties"

mean\_intensity = self.intensityproperties[c]['mean']

std\_intensity = self.intensityproperties[c]['sd']

lower\_bound = self.intensityproperties[c]['percentile\_00\_5']

upper\_bound = self.intensityproperties[c]['percentile\_99\_5']

data[c] = np.clip(data[c], lower\_bound, upper\_bound)

data[c] = (data[c] - mean\_intensity) / std\_intensity

if use\_nonzero\_mask[c]:

data[c][seg[-1] < 0] = 0

elif scheme == "CT2":

# clip to lb and ub from train data foreground, use mn and sd form each case for normalization

assert self.intensityproperties is not None, "ERROR: if there is a CT then we need intensity properties"

lower\_bound = self.intensityproperties[c]['percentile\_00\_5']

upper\_bound = self.intensityproperties[c]['percentile\_99\_5']

mask = (data[c] > lower\_bound) & (data[c] < upper\_bound)

data[c] = np.clip(data[c], lower\_bound, upper\_bound)

mn = data[c][mask].mean()

sd = data[c][mask].std()

data[c] = (data[c] - mn) / sd

if use\_nonzero\_mask[c]:

data[c][seg[-1] < 0] = 0

else:

if use\_nonzero\_mask[c]:

mask = seg[-1] >= 0

else:

mask = np.ones(seg.shape[1:], dtype=bool)

data[c][mask] = (data[c][mask] - data[c][mask].mean()) / (data[c][mask].std() + 1e-8)

data[c][mask == 0] = 0

return data, seg, properties

def preprocess\_test\_case(self, data\_files, target\_spacing, seg\_file=None, force\_separate\_z=None):

data, seg, properties = ImageCropper.crop\_from\_list\_of\_files(data\_files, seg\_file)

data = data.transpose((0, \*[i + 1 for i in self.transpose\_forward]))

seg = seg.transpose((0, \*[i + 1 for i in self.transpose\_forward]))

data, seg, properties = self.resample\_and\_normalize(data, target\_spacing, properties, seg,

force\_separate\_z=force\_separate\_z)

return data.astype(np.float32), seg, properties

def \_run\_internal(self, target\_spacing, case\_identifier, output\_folder\_stage, cropped\_output\_dir, force\_separate\_z,

all\_classes):

data, seg, properties = self.load\_cropped(cropped\_output\_dir, case\_identifier)

data = data.transpose((0, \*[i + 1 for i in self.transpose\_forward]))

seg = seg.transpose((0, \*[i + 1 for i in self.transpose\_forward]))

data, seg, properties = self.resample\_and\_normalize(data, target\_spacing,

properties, seg, force\_separate\_z)

all\_data = np.vstack((data, seg)).astype(np.float32)

# we need to find out where the classes are and sample some random locations

# let's do 10.000 samples per class

# seed this for reproducibility!

num\_samples = 10000

min\_percent\_coverage = 0.01 # at least 1% of the class voxels need to be selected, otherwise it may be too sparse

rndst = np.random.RandomState(1234)

class\_locs = {}

for c in all\_classes:

all\_locs = np.argwhere(all\_data[-1] == c)

if len(all\_locs) == 0:

class\_locs[c] = []

continue

target\_num\_samples = min(num\_samples, len(all\_locs))

target\_num\_samples = max(target\_num\_samples, int(np.ceil(len(all\_locs) \* min\_percent\_coverage)))

selected = all\_locs[rndst.choice(len(all\_locs), target\_num\_samples, replace=False)]

class\_locs[c] = selected

print(c, target\_num\_samples)

properties['class\_locations'] = class\_locs

print("saving: ", os.path.join(output\_folder\_stage, "%s.npz" % case\_identifier))

np.savez\_compressed(os.path.join(output\_folder\_stage, "%s.npz" % case\_identifier),

data=all\_data.astype(np.float32))

with open(os.path.join(output\_folder\_stage, "%s.pkl" % case\_identifier), 'wb') as f:

pickle.dump(properties, f)

def run(self, target\_spacings, input\_folder\_with\_cropped\_npz, output\_folder, data\_identifier,

num\_threads=default\_num\_threads, force\_separate\_z=None):

"""

:param target\_spacings: list of lists [[1.25, 1.25, 5]]

:param input\_folder\_with\_cropped\_npz: dim: c, x, y, z | npz\_file['data'] np.savez\_compressed(fname.npz, data=arr)

:param output\_folder:

:param num\_threads:

:param force\_separate\_z: None

:return:

"""

print("Initializing to run preprocessing")

print("npz folder:", input\_folder\_with\_cropped\_npz)

print("output\_folder:", output\_folder)

list\_of\_cropped\_npz\_files = subfiles(input\_folder\_with\_cropped\_npz, True, None, ".npz", True)

maybe\_mkdir\_p(output\_folder)

num\_stages = len(target\_spacings)

if not isinstance(num\_threads, (list, tuple, np.ndarray)):

num\_threads = [num\_threads] \* num\_stages

assert len(num\_threads) == num\_stages

# we need to know which classes are present in this dataset so that we can precompute where these classes are

# located. This is needed for oversampling foreground

all\_classes = load\_pickle(join(input\_folder\_with\_cropped\_npz, 'dataset\_properties.pkl'))['all\_classes']

for i in range(num\_stages):

all\_args = []

output\_folder\_stage = os.path.join(output\_folder, data\_identifier + "\_stage%d" % i)

maybe\_mkdir\_p(output\_folder\_stage)

spacing = target\_spacings[i]

for j, case in enumerate(list\_of\_cropped\_npz\_files):

case\_identifier = get\_case\_identifier\_from\_npz(case)

args = spacing, case\_identifier, output\_folder\_stage, input\_folder\_with\_cropped\_npz, force\_separate\_z, all\_classes

all\_args.append(args)

p = Pool(num\_threads[i])

p.starmap(self.\_run\_internal, all\_args)

p.close()

p.join()

class Preprocessor3DDifferentResampling(GenericPreprocessor):

def resample\_and\_normalize(self, data, target\_spacing, properties, seg=None, force\_separate\_z=None):

"""

data and seg must already have been transposed by transpose\_forward. properties are the un-transposed values

(spacing etc)

:param data:

:param target\_spacing:

:param properties:

:param seg:

:param force\_separate\_z:

:return:

"""

# target\_spacing is already transposed, properties["original\_spacing"] is not so we need to transpose it!

# data, seg are already transposed. Double check this using the properties

original\_spacing\_transposed = np.array(properties["original\_spacing"])[self.transpose\_forward]

before = {

'spacing': properties["original\_spacing"],

'spacing\_transposed': original\_spacing\_transposed,

'data.shape (data is transposed)': data.shape

}

# remove nans

data[np.isnan(data)] = 0

data, seg = resample\_patient(data, seg, np.array(original\_spacing\_transposed), target\_spacing, 3, 1,

force\_separate\_z=force\_separate\_z, order\_z\_data=3, order\_z\_seg=1,

separate\_z\_anisotropy\_threshold=self.resample\_separate\_z\_anisotropy\_threshold)

after = {

'spacing': target\_spacing,

'data.shape (data is resampled)': data.shape

}

print("before:", before, "\nafter: ", after, "\n")

if seg is not None: # hippocampus 243 has one voxel with -2 as label. wtf?

seg[seg < -1] = 0

properties["size\_after\_resampling"] = data[0].shape

properties["spacing\_after\_resampling"] = target\_spacing

use\_nonzero\_mask = self.use\_nonzero\_mask

assert len(self.normalization\_scheme\_per\_modality) == len(data), "self.normalization\_scheme\_per\_modality " \

"must have as many entries as data has " \

"modalities"

assert len(self.use\_nonzero\_mask) == len(data), "self.use\_nonzero\_mask must have as many entries as data" \

" has modalities"

for c in range(len(data)):

scheme = self.normalization\_scheme\_per\_modality[c]

if scheme == "CT":

# clip to lb and ub from train data foreground and use foreground mn and sd from training data

assert self.intensityproperties is not None, "ERROR: if there is a CT then we need intensity properties"

mean\_intensity = self.intensityproperties[c]['mean']

std\_intensity = self.intensityproperties[c]['sd']

lower\_bound = self.intensityproperties[c]['percentile\_00\_5']

upper\_bound = self.intensityproperties[c]['percentile\_99\_5']

data[c] = np.clip(data[c], lower\_bound, upper\_bound)

data[c] = (data[c] - mean\_intensity) / std\_intensity

if use\_nonzero\_mask[c]:

data[c][seg[-1] < 0] = 0

elif scheme == "CT2":

# clip to lb and ub from train data foreground, use mn and sd form each case for normalization

assert self.intensityproperties is not None, "ERROR: if there is a CT then we need intensity properties"

lower\_bound = self.intensityproperties[c]['percentile\_00\_5']

upper\_bound = self.intensityproperties[c]['percentile\_99\_5']

mask = (data[c] > lower\_bound) & (data[c] < upper\_bound)

data[c] = np.clip(data[c], lower\_bound, upper\_bound)

mn = data[c][mask].mean()

sd = data[c][mask].std()

data[c] = (data[c] - mn) / sd

if use\_nonzero\_mask[c]:

data[c][seg[-1] < 0] = 0

else:

if use\_nonzero\_mask[c]:

mask = seg[-1] >= 0

else:

mask = np.ones(seg.shape[1:], dtype=bool)

data[c][mask] = (data[c][mask] - data[c][mask].mean()) / (data[c][mask].std() + 1e-8)

data[c][mask == 0] = 0

return data, seg, properties

class Preprocessor3DBetterResampling(GenericPreprocessor):

"""

This preprocessor always uses force\_separate\_z=False. It does resampling to the target spacing with third

order spline for data (just like GenericPreprocessor) and seg (unlike GenericPreprocessor). It never does separate

resampling in z.

"""

def resample\_and\_normalize(self, data, target\_spacing, properties, seg=None, force\_separate\_z=False):

"""

data and seg must already have been transposed by transpose\_forward. properties are the un-transposed values

(spacing etc)

:param data:

:param target\_spacing:

:param properties:

:param seg:

:param force\_separate\_z:

:return:

"""

if force\_separate\_z is not False:

print("WARNING: Preprocessor3DBetterResampling always uses force\_separate\_z=False. "

"You specified %s. Your choice is overwritten" % str(force\_separate\_z))

force\_separate\_z = False

# be safe

assert force\_separate\_z is False

# target\_spacing is already transposed, properties["original\_spacing"] is not so we need to transpose it!

# data, seg are already transposed. Double check this using the properties

original\_spacing\_transposed = np.array(properties["original\_spacing"])[self.transpose\_forward]

before = {

'spacing': properties["original\_spacing"],

'spacing\_transposed': original\_spacing\_transposed,

'data.shape (data is transposed)': data.shape

}

# remove nans

data[np.isnan(data)] = 0

data, seg = resample\_patient(data, seg, np.array(original\_spacing\_transposed), target\_spacing, 3, 3,

force\_separate\_z=force\_separate\_z, order\_z\_data=99999, order\_z\_seg=99999,

separate\_z\_anisotropy\_threshold=self.resample\_separate\_z\_anisotropy\_threshold)

after = {

'spacing': target\_spacing,

'data.shape (data is resampled)': data.shape

}

print("before:", before, "\nafter: ", after, "\n")

if seg is not None: # hippocampus 243 has one voxel with -2 as label. wtf?

seg[seg < -1] = 0

properties["size\_after\_resampling"] = data[0].shape

properties["spacing\_after\_resampling"] = target\_spacing

use\_nonzero\_mask = self.use\_nonzero\_mask

assert len(self.normalization\_scheme\_per\_modality) == len(data), "self.normalization\_scheme\_per\_modality " \

"must have as many entries as data has " \

"modalities"

assert len(self.use\_nonzero\_mask) == len(data), "self.use\_nonzero\_mask must have as many entries as data" \

" has modalities"

for c in range(len(data)):

scheme = self.normalization\_scheme\_per\_modality[c]

if scheme == "CT":

# clip to lb and ub from train data foreground and use foreground mn and sd from training data

assert self.intensityproperties is not None, "ERROR: if there is a CT then we need intensity properties"

mean\_intensity = self.intensityproperties[c]['mean']

std\_intensity = self.intensityproperties[c]['sd']

lower\_bound = self.intensityproperties[c]['percentile\_00\_5']

upper\_bound = self.intensityproperties[c]['percentile\_99\_5']

data[c] = np.clip(data[c], lower\_bound, upper\_bound)

data[c] = (data[c] - mean\_intensity) / std\_intensity

if use\_nonzero\_mask[c]:

data[c][seg[-1] < 0] = 0

elif scheme == "CT2":

# clip to lb and ub from train data foreground, use mn and sd form each case for normalization

assert self.intensityproperties is not None, "ERROR: if there is a CT then we need intensity properties"

lower\_bound = self.intensityproperties[c]['percentile\_00\_5']

upper\_bound = self.intensityproperties[c]['percentile\_99\_5']

mask = (data[c] > lower\_bound) & (data[c] < upper\_bound)

data[c] = np.clip(data[c], lower\_bound, upper\_bound)

mn = data[c][mask].mean()

sd = data[c][mask].std()

data[c] = (data[c] - mn) / sd

if use\_nonzero\_mask[c]:

data[c][seg[-1] < 0] = 0

else:

if use\_nonzero\_mask[c]:

mask = seg[-1] >= 0

else:

mask = np.ones(seg.shape[1:], dtype=bool)

data[c][mask] = (data[c][mask] - data[c][mask].mean()) / (data[c][mask].std() + 1e-8)

data[c][mask == 0] = 0

return data, seg, properties

class PreprocessorFor2D(GenericPreprocessor):

def \_\_init\_\_(self, normalization\_scheme\_per\_modality, use\_nonzero\_mask, transpose\_forward: (tuple, list), intensityproperties=None):

super(PreprocessorFor2D, self).\_\_init\_\_(normalization\_scheme\_per\_modality, use\_nonzero\_mask,

transpose\_forward, intensityproperties)

def run(self, target\_spacings, input\_folder\_with\_cropped\_npz, output\_folder, data\_identifier,

num\_threads=default\_num\_threads, force\_separate\_z=None):

print("Initializing to run preprocessing")

print("npz folder:", input\_folder\_with\_cropped\_npz)

print("output\_folder:", output\_folder)

list\_of\_cropped\_npz\_files = subfiles(input\_folder\_with\_cropped\_npz, True, None, ".npz", True)

assert len(list\_of\_cropped\_npz\_files) != 0, "set list of files first"

maybe\_mkdir\_p(output\_folder)

all\_args = []

num\_stages = len(target\_spacings)

# we need to know which classes are present in this dataset so that we can precompute where these classes are

# located. This is needed for oversampling foreground

all\_classes = load\_pickle(join(input\_folder\_with\_cropped\_npz, 'dataset\_properties.pkl'))['all\_classes']

for i in range(num\_stages):

output\_folder\_stage = os.path.join(output\_folder, data\_identifier + "\_stage%d" % i)

maybe\_mkdir\_p(output\_folder\_stage)

spacing = target\_spacings[i]

for j, case in enumerate(list\_of\_cropped\_npz\_files):

case\_identifier = get\_case\_identifier\_from\_npz(case)

args = spacing, case\_identifier, output\_folder\_stage, input\_folder\_with\_cropped\_npz, force\_separate\_z, all\_classes

all\_args.append(args)

p = Pool(num\_threads)

p.starmap(self.\_run\_internal, all\_args)

p.close()

p.join()

def resample\_and\_normalize(self, data, target\_spacing, properties, seg=None, force\_separate\_z=None):

original\_spacing\_transposed = np.array(properties["original\_spacing"])[self.transpose\_forward]

before = {

'spacing': properties["original\_spacing"],

'spacing\_transposed': original\_spacing\_transposed,

'data.shape (data is transposed)': data.shape

}

target\_spacing[0] = original\_spacing\_transposed[0]

data, seg = resample\_patient(data, seg, np.array(original\_spacing\_transposed), target\_spacing, 3, 1,

force\_separate\_z=force\_separate\_z, order\_z\_data=0, order\_z\_seg=0,

separate\_z\_anisotropy\_threshold=self.resample\_separate\_z\_anisotropy\_threshold)

after = {

'spacing': target\_spacing,

'data.shape (data is resampled)': data.shape

}

print("before:", before, "\nafter: ", after, "\n")

if seg is not None: # hippocampus 243 has one voxel with -2 as label. wtf?

seg[seg < -1] = 0

properties["size\_after\_resampling"] = data[0].shape

properties["spacing\_after\_resampling"] = target\_spacing

use\_nonzero\_mask = self.use\_nonzero\_mask

assert len(self.normalization\_scheme\_per\_modality) == len(data), "self.normalization\_scheme\_per\_modality " \

"must have as many entries as data has " \

"modalities"

assert len(self.use\_nonzero\_mask) == len(data), "self.use\_nonzero\_mask must have as many entries as data" \

" has modalities"

print("normalization...")

for c in range(len(data)):

scheme = self.normalization\_scheme\_per\_modality[c]

if scheme == "CT":

# clip to lb and ub from train data foreground and use foreground mn and sd from training data

assert self.intensityproperties is not None, "ERROR: if there is a CT then we need intensity properties"

mean\_intensity = self.intensityproperties[c]['mean']

std\_intensity = self.intensityproperties[c]['sd']

lower\_bound = self.intensityproperties[c]['percentile\_00\_5']

upper\_bound = self.intensityproperties[c]['percentile\_99\_5']

data[c] = np.clip(data[c], lower\_bound, upper\_bound)

data[c] = (data[c] - mean\_intensity) / std\_intensity

if use\_nonzero\_mask[c]:

data[c][seg[-1] < 0] = 0

elif scheme == "CT2":

# clip to lb and ub from train data foreground, use mn and sd form each case for normalization

assert self.intensityproperties is not None, "ERROR: if there is a CT then we need intensity properties"

lower\_bound = self.intensityproperties[c]['percentile\_00\_5']

upper\_bound = self.intensityproperties[c]['percentile\_99\_5']

mask = (data[c] > lower\_bound) & (data[c] < upper\_bound)

data[c] = np.clip(data[c], lower\_bound, upper\_bound)

mn = data[c][mask].mean()

sd = data[c][mask].std()

data[c] = (data[c] - mn) / sd

if use\_nonzero\_mask[c]:

data[c][seg[-1] < 0] = 0

else:

if use\_nonzero\_mask[c]:

mask = seg[-1] >= 0

else:

mask = np.ones(seg.shape[1:], dtype=bool)

data[c][mask] = (data[c][mask] - data[c][mask].mean()) / (data[c][mask].std() + 1e-8)

data[c][mask == 0] = 0

print("normalization done")

return data, seg, properties

class PreprocessorFor3D\_NoResampling(GenericPreprocessor):

def resample\_and\_normalize(self, data, target\_spacing, properties, seg=None, force\_separate\_z=None):

"""

if target\_spacing[0] is None or nan we use original\_spacing\_transposed[0] (no resampling along z)

:param data:

:param target\_spacing:

:param properties:

:param seg:

:param force\_separate\_z:

:return:

"""

original\_spacing\_transposed = np.array(properties["original\_spacing"])[self.transpose\_forward]

before = {

'spacing': properties["original\_spacing"],

'spacing\_transposed': original\_spacing\_transposed,

'data.shape (data is transposed)': data.shape

}

# remove nans

data[np.isnan(data)] = 0

target\_spacing = deepcopy(original\_spacing\_transposed)

#print(target\_spacing, original\_spacing\_transposed)

data, seg = resample\_patient(data, seg, np.array(original\_spacing\_transposed), target\_spacing, 3, 1,

force\_separate\_z=force\_separate\_z, order\_z\_data=0, order\_z\_seg=0,

separate\_z\_anisotropy\_threshold=self.resample\_separate\_z\_anisotropy\_threshold)

after = {

'spacing': target\_spacing,

'data.shape (data is resampled)': data.shape

}

st = "before:" + str(before) + '\nafter' + str(after) + "\n"

print(st)

if seg is not None: # hippocampus 243 has one voxel with -2 as label. wtf?

seg[seg < -1] = 0

properties["size\_after\_resampling"] = data[0].shape

properties["spacing\_after\_resampling"] = target\_spacing

use\_nonzero\_mask = self.use\_nonzero\_mask

assert len(self.normalization\_scheme\_per\_modality) == len(data), "self.normalization\_scheme\_per\_modality " \

"must have as many entries as data has " \

"modalities"

assert len(self.use\_nonzero\_mask) == len(data), "self.use\_nonzero\_mask must have as many entries as data" \

" has modalities"

for c in range(len(data)):

scheme = self.normalization\_scheme\_per\_modality[c]

if scheme == "CT":

# clip to lb and ub from train data foreground and use foreground mn and sd from training data

assert self.intensityproperties is not None, "ERROR: if there is a CT then we need intensity properties"

mean\_intensity = self.intensityproperties[c]['mean']

std\_intensity = self.intensityproperties[c]['sd']

lower\_bound = self.intensityproperties[c]['percentile\_00\_5']

upper\_bound = self.intensityproperties[c]['percentile\_99\_5']

data[c] = np.clip(data[c], lower\_bound, upper\_bound)

data[c] = (data[c] - mean\_intensity) / std\_intensity

if use\_nonzero\_mask[c]:

data[c][seg[-1] < 0] = 0

elif scheme == "CT2":

# clip to lb and ub from train data foreground, use mn and sd form each case for normalization

assert self.intensityproperties is not None, "ERROR: if there is a CT then we need intensity properties"

lower\_bound = self.intensityproperties[c]['percentile\_00\_5']

upper\_bound = self.intensityproperties[c]['percentile\_99\_5']

mask = (data[c] > lower\_bound) & (data[c] < upper\_bound)

data[c] = np.clip(data[c], lower\_bound, upper\_bound)

mn = data[c][mask].mean()

sd = data[c][mask].std()

data[c] = (data[c] - mn) / sd

if use\_nonzero\_mask[c]:

data[c][seg[-1] < 0] = 0

else:

if use\_nonzero\_mask[c]:

mask = seg[-1] >= 0

else:

mask = np.ones(seg.shape[1:], dtype=bool)

data[c][mask] = (data[c][mask] - data[c][mask].mean()) / (data[c][mask].std() + 1e-8)

data[c][mask == 0] = 0

return data, seg, properties

class PreprocessorFor2D\_noNormalization(GenericPreprocessor):

def resample\_and\_normalize(self, data, target\_spacing, properties, seg=None, force\_separate\_z=None):

original\_spacing\_transposed = np.array(properties["original\_spacing"])[self.transpose\_forward]

before = {

'spacing': properties["original\_spacing"],

'spacing\_transposed': original\_spacing\_transposed,

'data.shape (data is transposed)': data.shape

}

target\_spacing[0] = original\_spacing\_transposed[0]

data, seg = resample\_patient(data, seg, np.array(original\_spacing\_transposed), target\_spacing, 3, 1,

force\_separate\_z=force\_separate\_z, order\_z\_data=0, order\_z\_seg=0,

separate\_z\_anisotropy\_threshold=self.resample\_separate\_z\_anisotropy\_threshold)

after = {

'spacing': target\_spacing,

'data.shape (data is resampled)': data.shape

}

print("before:", before, "\nafter: ", after, "\n")

if seg is not None: # hippocampus 243 has one voxel with -2 as label. wtf?

seg[seg < -1] = 0

properties["size\_after\_resampling"] = data[0].shape

properties["spacing\_after\_resampling"] = target\_spacing

use\_nonzero\_mask = self.use\_nonzero\_mask

assert len(self.normalization\_scheme\_per\_modality) == len(data), "self.normalization\_scheme\_per\_modality " \

"must have as many entries as data has " \

"modalities"

assert len(self.use\_nonzero\_mask) == len(data), "self.use\_nonzero\_mask must have as many entries as data" \

" has modalities"

return data, seg, properties