import numpy as np

import math

from niftynet.layer.base\_layer import Layer

try:

import finufftpy

finufft = True

except ImportError:

finufft = False

def get\_center\_mask(im\_shape, pct=0.07):

mask = np.zeros(im\_shape)

half\_pct = (pct / 2)

center = [int(x / 2) for x in im\_shape]

if len(im\_shape) == 3:

mask[center[0] - math.ceil(im\_shape[0] \* half\_pct):math.ceil(center[0] + im\_shape[0] \* half\_pct),

center[1] - math.ceil(im\_shape[1] \* half\_pct):math.ceil(center[1] + im\_shape[1] \* half\_pct),

center[2] - math.ceil(im\_shape[2] \* half\_pct):math.ceil(center[2] + im\_shape[2] \* half\_pct)] = 1

elif len(im\_shape) == 2:

mask[center[0] - math.ceil(im\_shape[0] \* half\_pct):math.ceil(center[0] + im\_shape[0] \* half\_pct),

center[1] - math.ceil(im\_shape[1] \* half\_pct):math.ceil(center[1] + im\_shape[1] \* half\_pct)] = 1

return mask

def get\_center\_rect(im\_shape, pct=0.07,dim=0):

mask = np.zeros(im\_shape)

half\_pct = (pct / 2)

center = [int(x / 2) for x in im\_shape]

mask = np.swapaxes(mask,0,dim)

mask[:,center[1] - math.ceil(im\_shape[1] \* half\_pct):math.ceil(center[1] + im\_shape[1] \* half\_pct)] = 1

mask = np.swapaxes(mask,0,dim)

return mask

def get\_center\_cross(im\_shape, pct=0.07):

mask = np.zeros(im\_shape)

half\_pct = (pct / 2)

center = [int(x / 2) for x in im\_shape]

if len(im\_shape) == 3:

mask[center[0] - math.ceil(im\_shape[0] \* half\_pct):math.ceil(center[0] + im\_shape[0] \* half\_pct),:,:] = 1

mask[:,center[1] - math.ceil(im\_shape[1] \* half\_pct):math.ceil(center[1] + im\_shape[1] \* half\_pct), :] = 1

mask[:,:,center[2] - math.ceil(im\_shape[2] \* half\_pct):math.ceil(center[2] + im\_shape[2] \* half\_pct)] = 1

elif len(im\_shape) == 2:

mask[:, center[1] - math.ceil(im\_shape[1] \* half\_pct):math.ceil(center[1] + im\_shape[1] \* half\_pct)] = 1

mask[center[0] - math.ceil(im\_shape[0] \* half\_pct):math.ceil(center[0] + im\_shape[0] \* half\_pct), :] = 1

return mask

def segment\_array\_by\_locs(shape, locs):

"""

generate array segmented by locs

:param shape (tuple): shape of array

:param locs (list of ints):

:return (np.array):

"""

mask\_out = np.zeros(np.prod(shape), dtype=int)

for i in range(len(locs) - 1):

l = [locs[i],

locs[i + 1]]

mask\_out[l[0]:l[1]] = i + 1

return mask\_out.reshape(shape)

def assign\_segments\_to\_random\_indices(shape, seg\_lengths):

seg\_mask = np.zeros(shape, dtype='int')

random\_indices = sorted(np.random.choice(shape, replace=False, size=sum(seg\_lengths)))

seg\_new\_indices = (np.cumsum(seg\_lengths)).tolist()

seg\_new\_indices = [0] + seg\_new\_indices

# TODO test this again

for i in range(len(seg\_new\_indices) - 1):

seg\_mask[random\_indices[seg\_new\_indices[i]:seg\_new\_indices[i + 1]]] = i + 1

return seg\_mask

def assign\_segments\_to\_random\_blocks(shape, seg\_lengths):

"""

generate randomly segmented array based on seg\_lengths

:param im\_shape (list of ints):

:param seg\_lengths (list of ints):

:return:

"""

seg\_mask = np.zeros(shape, dtype='int')

seg\_lengths\_sorted = sorted(seg\_lengths, reverse=True)

for i, seg\_len in enumerate(seg\_lengths\_sorted):

loc = np.random.randint(0, seg\_mask.size)

while (sum(seg\_mask[loc:loc + seg\_len]) != 0) or (loc + seg\_len > seg\_mask.size): # ensure that the segment

loc = np.random.randint(0, seg\_mask.size)

seg\_mask[loc:loc + seg\_len] = i + 1

return seg\_mask

def create\_rand\_partition(im\_length, n\_seg):

"""

:param im\_length (int): length of 1D array to partition

:param n\_seg (int): num segs to partition into

:return: partition locations (list of indices)

"""

rand\_segment\_locs = sorted(np.random.randint(im\_length, size=n\_seg + 1).astype(list))

rand\_segment\_locs[0] = 0

rand\_segment\_locs[-1] = None

return rand\_segment\_locs

def create\_rotation\_matrix\_3d(angles) -> np.array:

"""

given a list of 3 angles, create a 3x3 rotation matrix that describes rotation about the origin

:param angles (list or np.array) : rotation angles in 3 dimensions

:return (np.array) : rotation matrix 3x3

"""

mat1 = np.array([[1., 0., 0.],

[0., math.cos(angles[0]), math.sin(angles[0])],

[0., -math.sin(angles[0]), math.cos(angles[0])]],

dtype='float')

mat2 = np.array([[math.cos(angles[1]), 0., -math.sin(angles[1])],

[0., 1., 0.],

[math.sin(angles[1]), 0., math.cos(angles[1])]],

dtype='float')

mat3 = np.array([[math.cos(angles[2]), math.sin(angles[2]), 0.],

[-math.sin(angles[2]), math.cos(angles[2]), 0.],

[0., 0., 1.]],

dtype='float')

mat = (mat1 @ mat2) @ mat3

return mat

class MotionSimLayer(Layer):

"""

given a real valued 3D MRI image, simulate random translations and rotations

"""

def \_\_init\_\_(self, image\_name, std\_rotation\_angle=0, std\_translation=10,

corrupt\_pct=(15, 20), freq\_encoding\_dim=(0, 1, 2), preserve\_center\_pct=0.07,

apply\_mask=True, nufft=False, corruption\_scheme='piecewise\_constant',

n\_seg=8, fixed\_n\_seg=False):

"""

:param image\_name (str): key in data dictionary

:param std\_rotation\_angle (float) : std of rotations

:param std\_translation (float) : std of translations

:param corrupt\_pct (list of ints): range of percents

:param freq\_encoding\_dim (list of ints): randomly choose freq encoding dim

:param preserve\_center\_pct (float): percentage of k-space center to preserve

:param name (str) : name of layer

:param apply\_mask (bool): apply mask to output or not

:param nufft (bool): whether to use nufft for introducing rotations

:param num\_pieces (int): number of pieces for piecewise constant simulation

:param corruption\_scheme 'piecewise\_transient', 'piecewise\_constant', 'gaussian'

raises ImportError if nufft is true but finufft cannot be imported

"""

self.image\_name = image\_name

self.trajectory = None

self.preserve\_center\_frequency\_pct = preserve\_center\_pct

self.freq\_encoding\_choice = freq\_encoding\_dim

self.frequency\_encoding\_dim = np.random.choice(self.freq\_encoding\_choice)

self.std\_rotation\_angle, self.std\_translation = std\_rotation\_angle, std\_translation

self.corrupt\_pct\_range = corrupt\_pct

self.apply\_mask = apply\_mask

self.corruption\_scheme = corruption\_scheme

self.n\_seg = n\_seg

self.fixed\_n\_seg = fixed\_n\_seg

self.nufft = nufft

if (not finufft) and nufft:

raise ImportError('finufftpy cannot be imported')

def \_calc\_dimensions(self, im\_shape):

"""

calculate dimensions based on im\_shape

:param im\_shape (list/tuple) : image shape

- sets self.phase\_encoding\_dims, self.phase\_encoding\_shape, self.num\_phase\_encoding\_steps, self.frequency\_encoding\_dim

- initializes self.translations and self.rotations

"""

pe\_dims = [0, 1, 2]

pe\_dims.pop(self.frequency\_encoding\_dim)

self.phase\_encoding\_dims = pe\_dims

im\_shape = list(im\_shape)

self.im\_shape = im\_shape.copy()

im\_shape.pop(self.frequency\_encoding\_dim)

self.phase\_encoding\_shape = im\_shape

self.num\_phase\_encoding\_steps = self.phase\_encoding\_shape[0] \* self.phase\_encoding\_shape[1]

self.translations = np.zeros(shape=(3, self.num\_phase\_encoding\_steps))

self.rotations = np.zeros(shape=(3, self.num\_phase\_encoding\_steps))

self.frequency\_encoding\_dim = len(

self.im\_shape) - 1 if self.frequency\_encoding\_dim == -1 else self.frequency\_encoding\_dim

def \_simulate\_random\_trajectory(self):

"""

corruption\_scheme is either {'piecewise\_transient','piecewise\_constant','gaussian'}

simulates transient blocked random trajectory using a random number of lines generated from corrupt\_pct\_range

modifies self.translations and self.rotations

"""

pct\_corrupt = np.random.uniform(\*[x / 100 for x in self.corrupt\_pct\_range])

corrupt\_matrix\_shape = [int(x \* math.sqrt(pct\_corrupt)) for x in self.phase\_encoding\_shape]

# TODO keep this as a vector

if np.prod(corrupt\_matrix\_shape) == 0:

corrupt\_matrix\_shape = [1, 1]

if self.corruption\_scheme in {'gaussian'}:

n\_seg = np.prod(corrupt\_matrix\_shape)

else:

if self.fixed\_n\_seg:

n\_seg = self.n\_seg

else:

n\_seg = np.random.choice(np.arange(1, self.n\_seg))

# segment a smaller vector occupying pct\_corrupt percent of the space

if self.corruption\_scheme in {'piecewise\_transient', 'piecewise\_constant'}:

seg\_locs = create\_rand\_partition(np.prod(corrupt\_matrix\_shape),

n\_seg=n\_seg)

else:

seg\_locs = list(range(n\_seg))

rand\_segmentation = segment\_array\_by\_locs(

shape=np.prod(corrupt\_matrix\_shape), locs=seg\_locs)

seg\_lengths = [(rand\_segmentation == seg\_num).sum() for seg\_num in np.unique(rand\_segmentation)]

# assign segments to a vector with same number of elements as pe-steps

if self.corruption\_scheme in {'piecewise\_transient', 'gaussian'}:

seg\_vector = assign\_segments\_to\_random\_indices(np.prod(self.phase\_encoding\_shape), seg\_lengths)

else:

seg\_vector = assign\_segments\_to\_random\_blocks(np.prod(self.phase\_encoding\_shape), seg\_lengths)

# reshape to phase encoding shape with a random order

# if np.random.random() > 0.5:

reshape\_order = np.random.choice(['F','C'])

seg\_array = seg\_vector.reshape(self.phase\_encoding\_shape, order=reshape\_order)

self.order = reshape\_order

# mask center k-space

if reshape\_order == 'C':

mask\_not\_including\_center = get\_center\_rect(self.phase\_encoding\_shape, dim=1) == 0

else:

mask\_not\_including\_center = get\_center\_rect(self.phase\_encoding\_shape, dim=0) == 0

seg\_array = seg\_array \* mask\_not\_including\_center

# generate random translations and rotations

rand\_translations = np.random.normal(scale=self.std\_translation, size=(n\_seg + 1, 3))

rand\_rotations = np.random.normal(scale=self.std\_rotation\_angle, size=(n\_seg + 1, 3))

# if segment==0, then no motion

rand\_translations[0, :] = 0

rand\_rotations[0, :] = 0

# lookup values for each segment

translations\_pe = [rand\_translations[:, i][seg\_array] for i in range(3)]

rotations\_pe = [rand\_rotations[:, i][seg\_array] for i in range(3)]

# reshape and convert to radians

translations = np.array(

[np.broadcast\_to(np.expand\_dims(x, self.frequency\_encoding\_dim), self.im\_shape) for x in translations\_pe])

rotations = np.array(

[np.broadcast\_to(np.expand\_dims(x, self.frequency\_encoding\_dim), self.im\_shape) for x in rotations\_pe])

rotations = rotations \* (math.pi / 180.) # convert to radians

self.translations = translations.reshape(3, -1)

self.rotations = rotations.reshape(3, -1).reshape(3, -1)

def \_fft\_im(self, image):

output = (np.fft.fftshift(np.fft.fftn(np.fft.ifftshift(image)))).astype(np.complex128)

return output

def \_ifft\_im(self, freq\_domain):

output = np.fft.ifftshift(np.fft.ifftn(freq\_domain))

return output

def \_translate\_freq\_domain(self, freq\_domain):

"""

image domain translation by adding phase shifts in frequency domain

:param freq\_domain - frequency domain data 3d numpy array:

:return frequency domain array with phase shifts added according to self.translations:

"""

lin\_spaces = [np.linspace(-0.5, 0.5, x) for x in freq\_domain.shape]

meshgrids = np.meshgrid(\*lin\_spaces, indexing='ij')

grid\_coords = np.array([mg.flatten() for mg in meshgrids])

phase\_shift = np.multiply(grid\_coords, self.translations).sum(axis=0) # phase shift is added

exp\_phase\_shift = np.exp(-2j \* math.pi \* phase\_shift)

freq\_domain\_translated = np.multiply(exp\_phase\_shift, freq\_domain.flatten(order='C')).reshape(freq\_domain.shape)

return freq\_domain\_translated

def \_rotate\_coordinates(self):

"""

:return: grid\_coordinates after applying self.rotations

"""

center = [math.ceil((x - 1) / 2) for x in self.im\_shape]

[i1, i2, i3] = np.meshgrid(np.arange(self.im\_shape[0]) - center[0],

np.arange(self.im\_shape[1]) - center[1],

np.arange(self.im\_shape[2]) - center[2], indexing='ij')

grid\_coordinates = np.array([i1.T.flatten(), i2.T.flatten(), i3.T.flatten()])

rotations = self.rotations.reshape([3] + self.im\_shape)

ix = (len(self.im\_shape) + 1) \* [slice(None)]

ix[self.frequency\_encoding\_dim + 1] = 0 # dont need to rotate along freq encoding

rotations = rotations[ix].reshape(3, -1)

rotation\_matrices = np.apply\_along\_axis(create\_rotation\_matrix\_3d, axis=0, arr=rotations).transpose([-1, 0, 1])

rotation\_matrices = rotation\_matrices.reshape(self.phase\_encoding\_shape + [3, 3])

rotation\_matrices = np.expand\_dims(rotation\_matrices, self.frequency\_encoding\_dim)

rotation\_matrices = np.tile(rotation\_matrices,

reps=([self.im\_shape[

self.frequency\_encoding\_dim] if i == self.frequency\_encoding\_dim else 1

for i in range(5)])) # tile in freq encoding dimension

rotation\_matrices = rotation\_matrices.reshape([-1, 3, 3])

# tile grid coordinates

grid\_coordinates\_tiled = np.tile(grid\_coordinates, [3, 1])

grid\_coordinates\_tiled = grid\_coordinates\_tiled.reshape([3, -1], order='F').T

rotation\_matrices = rotation\_matrices.reshape([-1, 3])

new\_grid\_coords = (rotation\_matrices \* grid\_coordinates\_tiled).sum(axis=1)

# reshape new grid coords back to 3 x nvoxels

new\_grid\_coords = new\_grid\_coords.reshape([3, -1], order='F')

# scale data between -pi and pi

max\_vals = [abs(x) for x in grid\_coordinates[:, 0]]

new\_grid\_coordinates\_scaled = [(new\_grid\_coords[i, :] / max\_vals[i]) \* math.pi for i in

range(new\_grid\_coords.shape[0])]

new\_grid\_coordinates\_scaled = [np.asfortranarray(i) for i in new\_grid\_coordinates\_scaled]

return new\_grid\_coordinates\_scaled, [grid\_coordinates, new\_grid\_coords]

def \_nufft(self, freq\_domain\_data, iflag=1, eps=1E-7):

"""

rotate coordinates and perform nufft

:param freq\_domain\_data:

:param iflag/eps: see finufftpy doc

:param eps: precision of nufft

:return: nufft of freq\_domain\_data after applying self.rotations

"""

if not finufft:

raise ImportError('finufftpy not available')

new\_grid\_coords = self.\_rotate\_coordinates()[0]

# initialize array for nufft output

f = np.zeros([len(new\_grid\_coords[0])], dtype=np.complex128, order='F')

freq\_domain\_data\_flat = np.asfortranarray(freq\_domain\_data.flatten(order='F'))

finufftpy.nufft3d1(new\_grid\_coords[0], new\_grid\_coords[1], new\_grid\_coords[2], freq\_domain\_data\_flat,

iflag, eps, self.im\_shape[0], self.im\_shape[1],

self.im\_shape[2], f, debug=0, spread\_debug=0, spread\_sort=2, fftw=0, modeord=0,

chkbnds=0, upsampfac=1.25) # upsampling at 1.25 saves time at low precisions

im\_out = f.reshape(self.im\_shape, order='F')

return im\_out

def layer\_op(self, input\_data, mask=None,

test\_trajectory=False, freq\_encoding\_dim=None,

translations\_rotations=None):

if freq\_encoding\_dim is None:

self.frequency\_encoding\_dim = np.random.choice(self.freq\_encoding\_choice)

else:

self.frequency\_encoding\_dim = freq\_encoding\_dim

image\_data = input\_data[self.image\_name]

original\_image = np.squeeze(image\_data[:, :, :, 0, 0])

self.\_calc\_dimensions(original\_image.shape)

if translations\_rotations is None:

self.\_simulate\_random\_trajectory()

else:

self.translations, self.rotations = translations\_rotations

# fft

im\_freq\_domain = self.\_fft\_im(original\_image)

translated\_im\_freq\_domain = self.\_translate\_freq\_domain(freq\_domain=im\_freq\_domain)

# iNufft for rotations

if self.nufft:

corrupted\_im = self.\_nufft(translated\_im\_freq\_domain)

corrupted\_im = corrupted\_im / corrupted\_im.size # normalize

else:

corrupted\_im = self.\_ifft\_im(translated\_im\_freq\_domain)

# magnitude

corrupted\_im = abs(corrupted\_im)

if self.apply\_mask:

# todo: use input arg mask

mask\_im = input\_data['mask'][:, :, :, 0, 0] > 0

corrupted\_im = np.multiply(corrupted\_im, mask\_im)

masked\_original = np.multiply(original\_image, mask\_im)

image\_data[:, :, :, 0, 0] = masked\_original

image\_data[:, :, :, 0, 1] = corrupted\_im

output\_data = input\_data

output\_data[self.image\_name] = image\_data

return output\_data, mask

def \_\_call\_\_(self, \*args, \*\*kwargs):

return self.layer\_op(\*args, \*\*kwargs)