GANNoPrints

December 13, 2019

[1]:
```python
import torch
from torch import nn, optim
from torch.autograd.variable import Variable
from torchvision import transforms, datasets
import os
import sys
#sys.path.append('/home/students/exd949/fastMRI')
#from Logger import Logger

#import data_loader
import h5py, os
from functions import transforms as T
from functions.subsample import MaskFunc
from scipy.io import loadmat
from torch.utils.data import DataLoader
import numpy as np
from matplotlib import pyplot as plt
import random
import torch.nn.functional as F
```

[2]:
```python
class MRIDataset(DataLoader):
    def __init__(self, data_list, acceleration, center_fraction, use_seed):
        self.data_list = data_list
        self.acceleration = acceleration
        self.center_fraction = center_fraction
        self.use_seed = use_seed

    def __len__(self):
        return len(self.data_list)

    def __getitem__(self, idx):
        subject_id = self.data_list[idx]
        return get_epoch_batch(subject_id, self.acceleration, self.
 ↪center_fraction, self.use_seed)
```

[3]:
```python
""" Parts of the U-Net model """
```

```python
import torch
import torch.nn as nn
import torch.nn.functional as F


class DoubleConv(nn.Module):
    """(convolution => [BN] => ReLU) * 2"""

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.double_conv(x)


class Down(nn.Module):
    """Downscaling with maxpool then double conv"""

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.maxpool_conv = nn.Sequential(
            nn.MaxPool2d(2),
            DoubleConv(in_channels, out_channels)
        )

    def forward(self, x):
        return self.maxpool_conv(x)


class Up(nn.Module):
    """Upscaling then double conv"""

    def __init__(self, in_channels, out_channels, bilinear=True):
        super().__init__()

        # if bilinear, use the normal convolutions to reduce the number of␣
↪channels
        if bilinear:
```

```
            self.up = nn.Upsample(scale_factor=2, mode='bilinear',␣
↪align_corners=True)
        else:
            self.up = nn.ConvTranspose2d(in_channels // 2, in_channels // 2,␣
↪kernel_size=2, stride=2)

        self.conv = DoubleConv(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        # input is CHW
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]

        x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2,
                        diffY // 2, diffY - diffY // 2])
        # if you have padding issues, see
        # https://github.com/HaiyongJiang/U-Net-Pytorch-Unstructured-Buggy/
↪commit/0e854509c2cea854e247a9c615f175f76fbb2e3a
        # https://github.com/xiaopeng-liao/Pytorch-UNet/commit/
↪8ebac70e633bac59fc22bb5195e513d5832fb3bd
        x = torch.cat([x2, x1], dim=1)
        return self.conv(x)


class OutConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(OutConv, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=1)

    def forward(self, x):
        return self.conv(x)
```

[4]:
```
def show_slices(data, slice_nums, cmap=None): # visualisation
    fig = plt.figure(figsize=(15,10))
    for i, num in enumerate(slice_nums):
        plt.subplot(1, len(slice_nums), i + 1)
        plt.imshow(data[num], cmap=cmap)
        plt.axis('off')

def show_single_slice(data, cmap=None): # visualisation
    fig = plt.figure(figsize=(15,10))
    plt.imshow(data, cmap=cmap)
    plt.axis('off')
device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
```

[5]:

```python
def get_epoch_batch(subject_id, acc, center_fract, use_seed=True):
    ''' random select a few slices (batch_size) from each volume'''

    fname, rawdata_name, slice = subject_id

    with h5py.File(rawdata_name, 'r') as data:
        rawdata = data['kspace'][slice]

    slice_kspace = T.to_tensor(rawdata).unsqueeze(0)
    S, Ny, Nx, ps = slice_kspace.shape

    # apply random mask
    shape = np.array(slice_kspace.shape)
    mask_func = MaskFunc(center_fractions=[center_fract], accelerations=[acc])
    seed = None if not use_seed else tuple(map(ord, fname))
    mask = mask_func(shape, seed)

    # undersample
    masked_kspace = torch.where(mask == 0, torch.Tensor([0]), slice_kspace)
    masks = mask.repeat(S, Ny, 1, ps)

    img_gt, img_und = T.ifft2(slice_kspace), T.ifft2(masked_kspace)

    # perform data normalization which is important for network to learn useful␣
↪features
    # during inference there is no ground truth image so use the zero-filled␣
↪recon to normalize
    norm = T.complex_abs(img_und).max()
    if norm < 1e-6: norm = 1e-6

    # normalized data
    img_gt, img_und, rawdata_und = img_gt/norm, img_und/norm, masked_kspace/norm

    return img_gt.squeeze(0), img_und.squeeze(0), rawdata_und.squeeze(0), masks.
↪squeeze(0), norm
```

[6]:

```python
def load_data_path(train_data_path, val_data_path):
    """ Go through each subset (training, validation) and list all
    the file names, the file paths and the slices of subjects in the training␣
↪and validation sets
    """

    data_list = {}
    train_and_val = ['train', 'val']
    data_path = [train_data_path, val_data_path]

    for i in range(len(data_path)):
```

```
        data_list[train_and_val[i]] = []

        which_data_path = data_path[i]

        for fname in sorted(os.listdir(which_data_path)):

            subject_data_path = os.path.join(which_data_path, fname)

            if not os.path.isfile(subject_data_path): continue

            with h5py.File(subject_data_path, 'r') as data:
                if i==0:
                    k='kspace'
                if i==1:
                    k='kspace_4af'
                num_slice = data[k].shape[0]

            # the first 5 slices are mostly noise so it is better to exlude them
            data_list[train_and_val[i]] += [(fname, subject_data_path, slice)␣
↪for slice in range(5, num_slice)]

    return data_list
```

[7]:
```
def getTrainData(number):
    if (number==0):
        return []
    data_path_train = '/data/local/NC2019MRI/train'
    data_path_val = '/data/local/NC2019MRI/test'
    data_list = load_data_path(data_path_train, data_path_val) # first load all␣
↪file names, paths and slices.

    acc = 8
    cen_fract = 0.04
    seed = False # random masks for each slice
    num_workers = 0 #12 # data loading is faster using a bigger number for␣
↪num_workers. 0 means using one cpu to load data

    # create data loader for training set. It applies same to validation set as␣
↪well
    train_dataset = MRIDataset(data_list['train'], acceleration=acc,␣
↪center_fraction=cen_fract, use_seed=seed)
    train_loader = DataLoader(train_dataset, shuffle=True, batch_size=1,␣
↪num_workers=num_workers)
    data=[]
    for iteration, sample in enumerate(train_loader):
```

```
        img_gt, img_und, rawdata_und, masks, norm = sample

        # stack different slices into a volume for visualisation
        A = masks[...,0].squeeze()
        B = torch.log(T.complex_abs(rawdata_und) + 1e-9).squeeze()
        C = T.complex_abs(img_und).squeeze()
        D = T.complex_abs(img_gt).squeeze()
        all_imgs = torch.stack([A,B,C,D], dim=0)
        data.append(all_imgs)

        # from left to right: mask, masked kspace, undersampled image, ground␣
↪truth
        #show_slices(all_imgs, [0, 1, 2, 3], cmap='gray')
        #plt.pause(1)
        #print("Iteration",iteration, "Data len",len(data))
        if iteration >= (number-1):
            #print("Breaking")
            break
    return data
```

[8]:
```
def getLoaders():
    data_path_train = '/data/local/NC2019MRI/train'
    data_path_val = '/data/local/NC2019MRI/test'
    data_list = load_data_path(data_path_train, data_path_val) # first load all␣
↪file names, paths and slices.

    acc = 8
    cen_fract = 0.04
    seed = False # random masks for each slice
    num_workers = 0 #12 # data loading is faster using a bigger number for␣
↪num_workers. 0 means using one cpu to load data

    # create data loader for training set. It applies same to validation set as␣
↪well
    train_dataset = MRIDataset(data_list['train'], acceleration=acc,␣
↪center_fraction=cen_fract, use_seed=seed)
    train_loader = DataLoader(train_dataset, shuffle=True, batch_size=1,␣
↪num_workers=num_workers)

    return train_loader


def getCrossValidation():
    Data=[]
    FullData=[]
```

```python
    DL=getLoaders()
    #print("Here")
    for iteration, sample in enumerate(DL):
        FullData.append(sample)
    #print(len(FullData))
    folds=3
    ValLower=0
    ValUpper=len(FullData)/folds
    for i in range(0,folds):
        TrainSet=[]
        ValSet=[]
        #print(ValLower,ValUpper)
        for j in range(0,len(FullData)):
            if ((j>=ValLower) and (j<ValUpper)):
                #print(j,"for fold",i)
                ValSet.append(FullData[j])
            else:
                TrainSet.append(FullData[j])
        Data.append([TrainSet,ValSet])
        ValLower=ValUpper
        ValUpper=ValUpper+len(FullData)/folds

    #for i in range (0,folds):
        #print(len(Data[i][0]),len(Data[i][1]))
    return Data

#d=getCrossValidation()
```

[9]:
```python
from functions.subsample import MaskFunc

def apply4Mask(vk):
    volume_kspace = T.to_tensor(vk)
    mask_func0 = MaskFunc(center_fractions=[0.08], accelerations=[4])  # Create␣
↪the mask function object
    shape = np.array(volume_kspace.shape)
    mask0 = mask_func0(shape, seed=0) # use seed here to exclude randomness
    masked_kspace0 = torch.where(mask0 == 0, torch.Tensor([0]), volume_kspace)
    S_Num, Ny, Nx, _ = volume_kspace.shape
    masks0 = mask0.repeat(S_Num, Ny, 1, 1).squeeze() # masks when AF=4
    return masked_kspace0
```

[10]:
```python
import random
def getRandomValData():
    file_path = '/data/local/NC2019MRI/test/'
    valData=np.asarray([])
    num=random.randint(0,len(sorted(os.listdir(file_path)))-1)
    fname=sorted(os.listdir(file_path))[num]
```

```
    subject_path = os.path.join(file_path, fname)
    with h5py.File(subject_path,  "r") as hf:
        volume_kspace_4af = hf['kspace_4af'][()]
        return volume_kspace_4af


def getRandomTrainData():
    file_path = '/data/local/NC2019MRI/train/'
    valData=np.asarray([])
    num=random.randint(0,len(sorted(os.listdir(file_path)))-1)
    fname=sorted(os.listdir(file_path))[num]
    subject_path = os.path.join(file_path, fname)
    with h5py.File(subject_path,  "r") as hf:
        volume_kspace = hf['kspace'][()]
        volume_kspace_4af=apply4Mask(volume_kspace)
        volume_kspace_4af.requires_grad=True
        volume_kspace2 = T.to_tensor(volume_kspace)
        volume_kspace2.requires_grad=True
        volume_image = T.ifft2(volume_kspace2)
        #volume_image.requires_grad=True
        volume_image_abs = T.complex_abs(volume_image)
        return volume_kspace_4af ,volume_image

vd,ri=getRandomTrainData()
num=random.randint(0,vd.shape[0]-1)
#show_slices(vd, [num], cmap='gray')
#show_slices(vd, [num], cmap='gray') # Original images without undersampling
```

[11]:
```
class DiscriminatorNet(torch.nn.Module):

    def __init__(self):
        super(DiscriminatorNet, self).__init__()
        n_features = 1
        n_out = 1

        self.l1 = nn.Sequential(
            nn.Conv2d(in_channels=n_features, out_channels=16, kernel_size=4,␣
 ↪stride=2, padding=1, bias = False)
        )
        self.l2 = nn.Sequential(
            nn.LeakyReLU(0.2, inplace = True)
        )
        self.l3 = nn.Sequential(
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=4, stride=2,␣
 ↪padding=1, bias = False)
        )
        self.l4 = nn.Sequential(
```

```
            nn.LeakyReLU(0.2, inplace = True)
        )
        self.l5 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2,␣
↪padding=1, bias = False)
        )
        self.l6 = nn.Sequential(
            nn.Conv2d(64, 1, 4, 1, 0, bias = False)
        )
        self.l7 = nn.Sequential(
            nn.AdaptiveMaxPool2d((1,1))
        )
        self.l8 = nn.Sequential(
            nn.Sigmoid()
        )

        """self.main = nn.Sequential(

            nn.LeakyReLU(0.2, inplace = True),
            nn.Conv2d(128, 256, 4, 2, 1, bias = False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace = True),
            nn.Conv2d(256, 512, 4, 2, 1, bias = False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace = True),
            nn.Conv2d(512, 1, 4, 1, 0, bias = False),
            nn.Sigmoid()
        )"""

    def forward(self, x):
        #print("Before l1",x.shape)
        x = self.l1(x)
        #print("Before l2",x.shape)
        x = self.l2(x)
        #print("Before l3",x.shape)
        x = self.l3(x)
        #print("Before l4",x.shape)
        x = self.l4(x)
        #print("Before l5",x.shape)
        x = self.l5(x)
        #print("Before l6",x.shape)
        x = self.l6(x)
        #print("Before l7",x.shape)
        x = self.l7(x)
        #print("Before l8",x.shape)
        x = self.l8(x)
        #print("Final gen_out",x.shape)
```

```
        return x
discriminator = DiscriminatorNet()
discriminator.cuda()
```

[11]:
```
DiscriminatorNet(
  (l1): Sequential(
    (0): Conv2d(1, 16, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
  )
  (l2): Sequential(
    (0): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (l3): Sequential(
    (0): Conv2d(16, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
  )
  (l4): Sequential(
    (0): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (l5): Sequential(
    (0): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
  )
  (l6): Sequential(
    (0): Conv2d(64, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
  )
  (l7): Sequential(
    (0): AdaptiveMaxPool2d(output_size=(1, 1))
  )
  (l8): Sequential(
    (0): Sigmoid()
  )
)
```

[12]:
```
class UNet(nn.Module):
    def __init__(self, n_channels, n_classes, bilinear=True):
        super(UNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.bilinear = bilinear

        self.inc = DoubleConv(n_channels, 64)
        self.down1 = Down(64, 128)
        self.down2 = Down(128, 256)
        self.down3 = Down(256, 512)
        self.down4 = Down(512, 512)
        self.up1 = Up(1024, 256, bilinear)
```

```
        self.up2 = Up(512, 128, bilinear)
        self.up3 = Up(256, 64, bilinear)
        self.up4 = Up(128, 64, bilinear)
        self.outc = OutConv(64, n_classes)

    def forward(self, x):
        #print("Start",x.grad_fn)
        x1 = self.inc(x)
        #print(x1.grad_fn)
        x2 = self.down1(x1)
        #print(x2.grad_fn)
        x3 = self.down2(x2)
        #print(x3.grad_fn)
        x4 = self.down3(x3)
        #print(x4.grad_fn)
        x5 = self.down4(x4)
        #print(x5.grad_fn)
        x = self.up1(x5, x4)
        #print(x.grad_fn)
        x = self.up2(x, x3)
        #print(x.grad_fn)
        x = self.up3(x, x2)
        #print(x.grad_fn)
        x = self.up4(x, x1)
        #print(x.grad_fn)
        logits = self.outc(x)
        #print("End",logits.grad_fn)
        return logits

generator = UNet(n_channels=1, n_classes=2)
generator.cuda()
```

[12]:
```
UNet(
  (inc): DoubleConv(
    (double_conv): Sequential(
      (0): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
  (down1): Down(
    (maxpool_conv): Sequential(
```

```
      (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (1): DoubleConv(
        (double_conv): Sequential(
          (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
          (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
          (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (5): ReLU(inplace=True)
        )
      )
    )
  )
  (down2): Down(
    (maxpool_conv): Sequential(
      (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (1): DoubleConv(
        (double_conv): Sequential(
          (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
          (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (5): ReLU(inplace=True)
        )
      )
    )
  )
  (down3): Down(
    (maxpool_conv): Sequential(
      (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (1): DoubleConv(
        (double_conv): Sequential(
          (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
          (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (5): ReLU(inplace=True)
        )
      )
    )
  )
  (down4): Down(
    (maxpool_conv): Sequential(
      (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (1): DoubleConv(
        (double_conv): Sequential(
          (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
          (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (5): ReLU(inplace=True)
        )
      )
    )
  )
  (up1): Up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): DoubleConv(
      (double_conv): Sequential(
        (0): Conv2d(1024, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (5): ReLU(inplace=True)
      )
    )
  )
```

```
  (up2): Up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): DoubleConv(
      (double_conv): Sequential(
        (0): Conv2d(512, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (5): ReLU(inplace=True)
      )
    )
  )
  (up3): Up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): DoubleConv(
      (double_conv): Sequential(
        (0): Conv2d(256, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (5): ReLU(inplace=True)
      )
    )
  )
  (up4): Up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): DoubleConv(
      (double_conv): Sequential(
        (0): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (5): ReLU(inplace=True)
      )
    )
  )
  (outc): OutConv(
    (conv): Conv2d(64, 2, kernel_size=(1, 1), stride=(1, 1))
```

```
  )
)
```

[13]:
```
d_optimizer = optim.AdamW(discriminator.parameters(), lr=0.0002)
g_optimizer = optim.AdamW(generator.parameters(), lr=0.0002)
```

[14]:
```
def ones_target(size):
    '''
    Tensor containing ones, with shape = size
    '''
    data = Variable(torch.ones(size, 1).cuda())
    return data

def zeros_target(size):
    '''
    Tensor containing zeros, with shape = size
    '''
    data = Variable(torch.zeros(size, 1).cuda())
    return data
```

[15]:
```
from skimage.measure import compare_ssim
def ssim(gt, pred):
    """ Compute Structural Similarity Index Metric (SSIM). """
    #print("Input",gt.shape, pred.shape)
    #print("Input val gt",gt)
    #print("Input val pred",pred)
    gtn=gt.detach().cpu().numpy()
    predn=pred.detach().cpu().numpy()

    ssim_val= compare_ssim(gtn.transpose(1, 2, 0), predn.transpose(1, 2, 0),␣
↪multichannel=True, data_range=gtn.max())
    #print("Input -> SSIM",ssim_val,ssim_val.shape,ssim_val.dtype)

    ssim_val_numpy=np.asarray([ssim_val])
    #print("To Numpy", ssim_val_numpy,ssim_val_numpy.shape,ssim_val_numpy.dtype)

    ssim_val_tensor=torch.tensor(ssim_val_numpy,requires_grad=True)
    #print("Numpy to Tensor",ssim_val_tensor,ssim_val_tensor.
↪shape,ssim_val_tensor.dtype)

    gt.cuda()
    pred.cuda()

    return ssim_val_tensor

def ssim_2d(gt, pred):
    """ Compute Structural Similarity Index Metric (SSIM). """
```

```
    print("Input",gt.shape, pred.shape)
    #print("Input val gt",gt)
    #print("Input val pred",pred)
    gtn=gt.detach().cpu().numpy()
    predn=pred.detach().cpu().numpy()

    ssim_val= compare_ssim(gtn, predn, multichannel=False, data_range=gtn.max())
    ssim_val_numpy=np.asarray([ssim_val])
    ssim_val_tensor=torch.tensor(ssim_val_numpy,requires_grad=True)
    gt.cuda()
    pred.cuda()

    return ssim_val_tensor

loss = nn.KLDivLoss()
```

[16]:
```
def train_discriminator(optimizer, real_data, fake_data):
    optimizer.zero_grad()

    # 1.1 Train on Real Data
    prediction_real = discriminator(real_data.unsqueeze(0).unsqueeze(0))
    #print(prediction_real.squeeze(0).shape, ones_target(1).shape)
    error_real = loss(prediction_real.squeeze(0).squeeze(0), ones_target(1) )
    error_real.backward(retain_graph=True)

    # 1.2 Train on Fake Data
    prediction_fake = discriminator(fake_data.unsqueeze(0).unsqueeze(0))
    error_fake = loss(prediction_fake.squeeze(0).squeeze(0), zeros_target(1))
    error_fake.backward(retain_graph=True)

    # 1.3 Update weights with gradients
    optimizer.step()

    # Return error and predictions for real and fake inputs
    return error_real + error_fake, prediction_real, prediction_fake
```

[17]:
```
def train_generator(optimizer, fake_data, real_data):
    # Reset gradients
    optimizer.zero_grad()
    prediction = discriminator(fake_data.unsqueeze(0).unsqueeze(0))
    error = loss(prediction.squeeze(0).squeeze(0), ones_target(1)) #changed from␣
 ↪prediction #ssim(fake_data_img.unsqueeze(0).cpu(),real_data.unsqueeze(0).
 ↪cpu()) #
    error.backward(retain_graph=True)
    fake_data.cuda()
    real_data.cuda()
```

```
    optimizer.step()
    return error
```

[18]:
```
def show_single_tensor_slice(t):
    #v_abs = T.complex_abs(t)   # Compute absolute value to get a real image
    t=t.detach().cpu()
    #print("volume_image_abs",volume_image_abs_4af.shape,volume_image_abs_4af.
 ↪dtype)
    show_single_slice(t.squeeze(0), cmap='gray')
    t.cuda()
```

[19]:
```
Data=getCrossValidation()
```

[ ]:
```
# Create logger instance
#logger = Logger(model_name='VGAN', data_name='MRI')
# Total number of epochs to train

num_epochs = 5
for epoch in range(num_epochs):
    ssim_error=[]
    loss_error=[]
    done=False
    fold=0
    for TrainSet, ValSet  in Data:
        print("Starting training for Epoch",epoch,"Fold",fold)
        for sample in TrainSet:
            img_gt, img_und, rawdata_und, masks, norm = sample

            img_gt.requires_grad=True
            img_und.requires_grad=True
            rawdata_und.requires_grad=True
            # stack different slices into a volume for visualisation
            A = masks[...,0].squeeze()
            B = torch.log(T.complex_abs(rawdata_und) + 1e-9).squeeze()
            C = T.complex_abs(img_und).squeeze()
            D = T.complex_abs(img_gt).squeeze()
            all_imgs = torch.stack([A,B,C,D], dim=0)

            #show_slices(all_imgs, [0, 1, 2, 3], cmap='gray')

            # 1. Train Discriminator
            comp_img_gt = T.complex_abs(img_gt)
            fs=T.center_crop(comp_img_gt, [320, 320])
            real_data = Variable(fs.squeeze(0),requires_grad=True).cuda()
```

```
            # Generate fake data and detach
            fs = T.complex_abs(T.ifft2(rawdata_und)).squeeze() #640x372
            fs=T.center_crop(fs, [320, 320])   #320x320
            gen_input=Variable(fs.unsqueeze(0).unsqueeze(0),requires_grad=True).cuda()
            fake_data_tensor = generator(gen_input)  #Output from gen,
            fake_data_tensor=fake_data_tensor.squeeze().permute(1,2,0)
            fake_data_img=T.complex_abs(fake_data_tensor)

            # Train D
            d_error, d_pred_real, d_pred_fake = train_discriminator(d_optimizer, real_data, fake_data_img)


            # 2. Train Generator
            # Generate fake data


            # Train G
            g_error = train_generator(g_optimizer, fake_data_img, real_data)
            loss_error.append(g_error.item())

            #ssim_val=ssim(fake_data_img.unsqueeze(0).cpu(),real_data.unsqueeze(0).cpu())
            #ssim_error.append(ssim_val.detach().numpy()[0])
            #print(ssim_val.detach().numpy()[0],g_error.item())

        # Display Progress every few batches
        print("Starting validation for Epoch",epoch,"Fold",fold)
        fold_error=[]
        for sample in ValSet:
            img_gt, img_und, rawdata_und, masks, norm = sample

            img_gt.requires_grad=True
            img_und.requires_grad=True
            rawdata_und.requires_grad=True

            comp_img_gt = T.complex_abs(img_gt)
            fs=T.center_crop(comp_img_gt, [320, 320])
            real_data = Variable(fs.squeeze(0),requires_grad=True).cuda()

            fs = T.complex_abs(T.ifft2(rawdata_und)).squeeze() #640x372
            fs=T.center_crop(fs, [320, 320])   #320x320
            gen_input=Variable(fs.unsqueeze(0).unsqueeze(0),requires_grad=True).cuda()
            fake_data_tensor = generator(gen_input)  #Output from gen,
```

```
            fake_data_tensor=fake_data_tensor.squeeze().permute(1,2,0)
            fake_data_img=T.complex_abs(fake_data_tensor)

            #ssim_real=T.center_crop(T.complex_abs(real_image[num]), [320, 320]).
↪detach().cuda()
            ssim_val=ssim(fake_data_img.unsqueeze(0).cpu(),real_data.
↪unsqueeze(0).cpu())
            #print("SSIM",ssim_val.detach().numpy()[0])
            ssim_error.append(ssim_val.detach().numpy()[0])
            fold_error.append(ssim_val.detach().numpy()[0])
            done=True
        print("Fold",fold,"SSIM Ave",np.mean(np.asarray(fold_error)))
        fold+=1
    print("Epoch",epoch,"SSIM Ave",np.mean(np.asarray(ssim_error)))
    torch.save(generator.state_dict(),'/home/students/exd949/fastMRI/models/
↪KLDivLoss_Adagrad_generator_1.pt')
    torch.save(discriminator.state_dict(),'/home/students/exd949/fastMRI/models/
↪KLDivLoss_Adagrad_discriminator_1.pt')
```

```
Starting training for Epoch 0 Fold 0

/bham/modules/roots/neural-comp/2019-20/lib64/python3.6/site-
packages/torch/nn/functional.py:1932: UserWarning: reduction: 'mean' divides the
total loss by both the batch size and the support size.'batchmean' divides only
by the batch size, and aligns with the KL div math definition.'mean' will be
changed to behave the same as 'batchmean' in the next major release.
  warnings.warn("reduction: 'mean' divides the total loss by both the batch size
and the support size."

Starting validation for Epoch 0 Fold 0

/bham/modules/roots/neural-comp/2019-20/lib/python3.6/site-
packages/ipykernel_launcher.py:10: UserWarning: DEPRECATED:
skimage.measure.compare_ssim has been moved to
skimage.metrics.structural_similarity. It will be removed from skimage.measure
in version 0.18.
  # Remove the CWD from sys.path while we load stuff.

Fold 0 SSIM Ave 0.5687091311130936
Starting training for Epoch 0 Fold 1
Starting validation for Epoch 0 Fold 1
Fold 1 SSIM Ave 0.5695295676362866
Starting training for Epoch 0 Fold 2
Starting validation for Epoch 0 Fold 2
Fold 2 SSIM Ave 0.5726399855497858
Epoch 0 SSIM Ave 0.5702921526091004
Starting training for Epoch 1 Fold 0
```

```
Starting validation for Epoch 1 Fold 0
Fold 0 SSIM Ave 0.5723798343157489
Starting training for Epoch 1 Fold 1
Starting validation for Epoch 1 Fold 1
Fold 1 SSIM Ave 0.573149670003155
Starting training for Epoch 1 Fold 2
Starting validation for Epoch 1 Fold 2
Fold 2 SSIM Ave 0.5762441987703808
Epoch 1 SSIM Ave 0.5739238438288646
Starting training for Epoch 2 Fold 0
Starting validation for Epoch 2 Fold 0
Fold 0 SSIM Ave 0.5759231443444185
Starting training for Epoch 2 Fold 1
Starting validation for Epoch 2 Fold 1
Fold 1 SSIM Ave 0.5766472212241699
Starting training for Epoch 2 Fold 2
Starting validation for Epoch 2 Fold 2
Fold 2 SSIM Ave 0.5797705703072938
Epoch 2 SSIM Ave 0.5774462645511231
Starting training for Epoch 3 Fold 0
Starting validation for Epoch 3 Fold 0
Fold 0 SSIM Ave 0.5794185294212847
Starting training for Epoch 3 Fold 1
Starting validation for Epoch 3 Fold 1
Fold 1 SSIM Ave 0.5801078093168229
Starting training for Epoch 3 Fold 2
Starting validation for Epoch 3 Fold 2
Fold 2 SSIM Ave 0.5832506828995667
Epoch 3 SSIM Ave 0.5809249676259642
Starting training for Epoch 4 Fold 0
Starting validation for Epoch 4 Fold 0
Fold 0 SSIM Ave 0.582866099460932
Starting training for Epoch 4 Fold 1
```

[ ]:

```
print("hello")
```

[ ]:

```
def save_reconstructions(reconstructions, out_dir):
    """
    Saves the reconstructions from a model into h5 files that is appropriate for␣
↪submission
    to the leaderboard.
    Args:
        reconstructions (dict[str, np.array]): A dictionary mapping input␣
↪filenames to
            corresponding reconstructions (of shape num_slices x height x width).
        out_dir (pathlib.Path): Path to the output directory where the␣
↪reconstructions
```

```
            should be saved.
    """
    for fname, recons in reconstructions.items():
        subject_path = os.path.join(out_dir, fname)
        print(subject_path)
        with h5py.File(subject_path, 'w') as f:
            f.create_dataset('reconstruction', data=recons)
```

[ ]:
```
file_path = '/data/local/NC2019MRI/test'

generator = UNet(n_channels=1, n_classes=2)
generator.cuda()
generator.load_state_dict(torch.load('/home/students/exd949/fastMRI/models/
 ↪KLDivLoss_Adam_generator_3.pt'))
generator.eval()

for fname in sorted(os.listdir(file_path)):
    subject_path = os.path.join(file_path, fname)
    with h5py.File(subject_path,  "r") as hf:
        volume_kspace_4af = hf['kspace_4af'][()]
        volume_kspace_8af = hf['kspace_8af'][()]
        before_recon_4 =Variable(T.center_crop(T.complex_abs(T.ifft2(T.
 ↪to_tensor(volume_kspace_4af))), [320, 320])).cuda()
        print("Before",before_recon_4.unsqueeze(0).shape)
        after_recon_4=generator(before_recon_4.unsqueeze(1))
        print("After",after_recon_4.shape)
        fake_4=T.complex_abs(after_recon_4.permute(0,2,3, 1))
        show_slices(before_recon_4, [5, 10, 20], cmap='gray')
        show_slices(fake_4, [5, 10, 20], cmap='gray')

        before_recon_8 = T.center_crop(T.complex_abs(T.ifft2(T.
 ↪to_tensor(volume_kspace_8af))), [320, 320])
        #show_slices(before_recon_8, [5, 10, 20], cmap='gray')

        #fname0 = 'file1000000.h5'
        #reconstructions = {fname0: cropped_gt.numpy(), fname1: cropped_4af.
 ↪numpy(), fname2: cropped_8af.numpy()}
        #out_dir = '/home/students/exd949/fastMRI/saved/'
        #if not (os.path.exists(out_dir)): os.makedirs(out_dir)
            #save_reconstructions(reconstructions, out_dir)
```

[ ]:
```
```