# GANNoPrints

December 13, 2019

[1]:
```
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]:
```
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]:
```
""" 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]:
```
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]:
```
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=[]
```

```
    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]:
```
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]:
```
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
```

[ ]:
```python
print("hello")
```

[ ]:
```python
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)
```

[ ]:
```
```

# Neural Computation Coursework

Temas Driver, Ehsun Hanif, Mo Wang, Esha Dasgupta, and Jesse Spielman

December 2019

## 1 Introduction

MRI, Magnetic Resonance Imagine, is a medical tool with a very high utility in the diagnostic field. MRI data is complex and in $k - space^3$, but essentially the data is not represented in image space, but rather in its Fourier components. To obtain an usable MRI image, an inverse Fourier transform must be applied upon the k-space data. This process, however, is time consuming and this has drastic impacts when patients are waiting for the results.

A way to speed up MRI reconstruction is to undersample the full k-space data and use machine learning techniques to learn the underlying structure and 'fill' in the gaps. This increases the speed at which MRI images can be rendered and scanned, while retaining the accuracy in the image for the medical professionals to use. Masks are applied to the full k-space data to get the undersampled data; the two masks used are `[kspace_4af]` and `[kspace_8af]` have 4 fold acceleration and 8 fold acceleration respectively, which means that they use a quarter and one eighth of the full data.

This assignment consisted of the task of reconstructing MRI images from undersampled data, so using a high acceleration rate, while preserving the quality of the image from the ground truth. A higher acceleration rate naturally leads to the reconstructed images to be blurry, which is less than ideal for diagnostic purposes.

The model for VGG16 architecture has been included but it has not been used for testing due to time constraints.

Two ways considered to retain a good quality of the image given a high sub-sampling rate, are to use an U-Net and a Generative Adversarial Network (GAN).

The data-set provided were of the HDF5 file format, which can be manipulated using the h5py package. Each file comprises of an MRI scan volume which consists of 'many stacked slices', containing the corresponding k-space data and some meta data associated with the scans. The k-space is the raw data in MRI and is complex-valued. The shape of the data is as follows: number of slices, height, width.

The k-space data and the its equivalent image are associated by an inverse Fourier transform. As the k-space data is complex-valued, the image is also complex-valued, and to make sense of the image, the absolute value of the inverse Fourier transformed image needs to be computed.

## 2 Design

A Generative Adversarial Network, GAN, is an example of an unsupervised learning model where two internal models attempt to learn the structure of the underlying data: the generator to fool the discriminator, and the discriminator to realise whether it is being tricked. In a GAN, the two networks compete with each other. The generative network tries to produce new instances of the data by learning patterns in the ground data set, while the discriminator decides if each data instance belongs to the original training data set or otherwise. The two models feed into each other and help improve each other's performance. This leads to the GAN generating high quality high dimensional data, but has a disadvantage of the network being difficult to optimise because the training is unstable.

MRI data has a high number of dimensions and furthermore, statistical inference between images will not need to be done;a GAN seemed like a sensible choice because the number of parameters it uses to train is less than the number of parameters in the data-set so it is forced to learn the underlying structure quickly.

A U-Net was used for the generating component while a discriminator net was used for the discriminator. Different loss functions were experimented with, for example Binary Cross Entropy Loss (BCELoss), L1, MSE and KLDivLoss. Training the discriminator comprises of four steps, the first one being training on real data, the second one being training on fake data, the third one is to update the wights with gradients using the Adam optimizer and at last but not the least, the calculation of error and predictions for real and fake inputs.

A U-net is a preliminary version of a FCN (Fully Convolutional Networks), which is explained below.

### 2.1 FCN

A FCN implements feature extraction similarly to a CNN. For a CNN, there is a fully connected layer after the convolutional layer which finds a linear mapping from the high-dimensional data space to a low-dimensional space which preserves key features[1]. This causes the feature map to become a feature vector for purposes of classification or regression[2]. Whereas, the FCN uses a fully connected deconvolutional layer after the convolutional layer. Therefore, after upsampling, a figure is obtained which has same size with the input figure. The deconvolution implements the upsampling and the network fully consists of convolutional layers, so it's called a fully convolutional network.

### 2.2 U-Net

A U-net is a more developed version of a FCN. The u-net architecture achieves very good performance on different biomedical segmentation applications[3]. Compared with a FCN, there are 2 advantages: i) Multiscale, it can extract

lots of different features. ii) It has a better performance on rebuilding larger medical images.

Figure 1[4] shows the structure of the U-Net.It can be divided into two parts, the former implementing the feature extraction, and the latter doing downsampling. The size of the input figure will be reduced and a high dimensional feature map will be obtained. Then, upsampling will rebuild the figure based upon the feature list. Finally, a new figure is generated after repeated deconvolution. Each time the feature extraction section passes through the pooling layer, the size of the image will be altered.

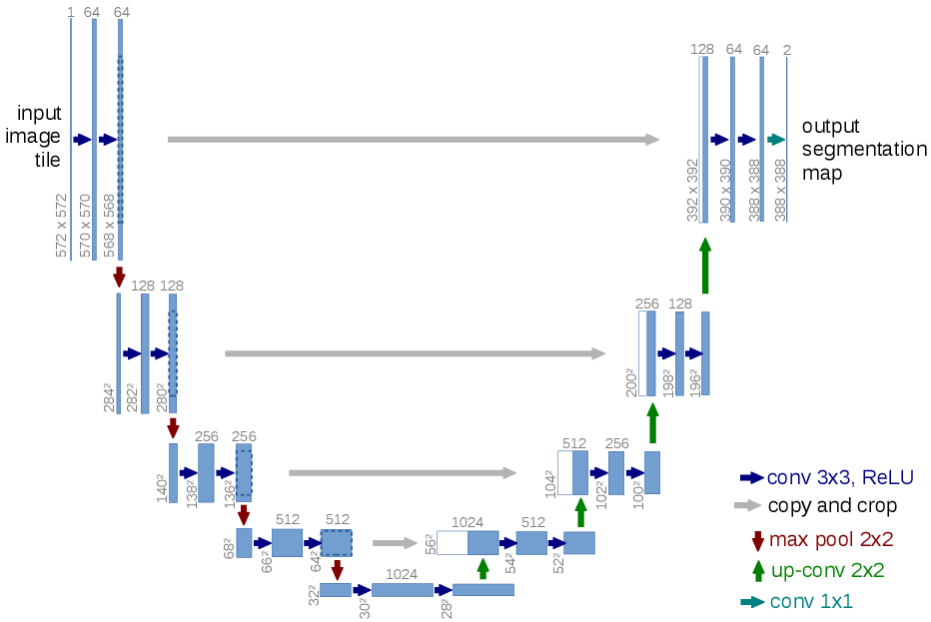


Fig. 1: U-Net Architecture

In the upsampling section, each upsampling is fused using the same number of channels as feature extraction, but they may have a different size, so feature extraction should be cropped to make it has the same size before the fusion.

### 2.3 Bi-linear Upsampling

Bi-linear upsampling is used to develop the model's performance. Generally, Linear interpolation is used to do upsampling and the size will double due to it. Because the previous method of upsampling is very simple, such as scaling, it will cause the image to become blurred. But for an image which is always 2 or 3 dimensional, bi-linear upsampling can be used. The resulting image looks better, with more natural transitions and smoother edges.

## 3 Implementation

### 3.1 Implementation of an U-Net

The U-Net implemented is similar to the typical architecture of a convolutional network. It consists of the repeated application of two 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride of size 2, as depicted in the listing below. But some hyper-parameters are changed, for example an U-net always uses a double convolution, however, for convenience,a two-layer convolution is defined first.

```
Convolution(3x3, padding=1)
Normalization
ReLU(True)
Convolution(3x3, padding=1)
Normalization
ReLU(True)
```

As the image is downsampled, the scale of the feature map changes from 64 to 512, upsampling is implemented to get the correct sized output.There is 1 input layer, 4 downsampling layers, 4 upsampling layers and an output layer. Every upsampling and downsampling function consists of a pooling and a repeated convolution. The listing below shows the size diversification.

```
Input(64)
Down(64,128)
Down(64,128)
Down(256,512)
Down(512,512)
Up(1024,256, bilinear)
Up(512,128, bilinear)
Up(256,64, bilinear)
Up(128,64, bilinear)
Output(64)
```

### 3.2 Implementation of a GAN with a U-Net

Cross validation is a technique used in data science to ensure that models are stable, and to avoid overfitting and underfitting. The validation and training data are drawn from the same distribution, so the data in the data loader (obtained by iterating through the HDF5 files) was split into 3 folds. The GAN would be trained on 2 folds and validated on the third cyclically to help decide the hyper parameters which would generalise best to unseen data. Standard practice is to 5 or 10 folds, but the model took ninety minutes per epoch for 10 folds, so

```python
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=[]
    DL=getLoaders()
    for iteration, sample in enumerate(DL):
        FullData.append(sample)
    folds=3
    ValLower=0
    ValUpper=len(FullData)/folds
    for i in range(0,folds):
        TrainSet=[]
        ValSet=[]
        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

    return Data
```

Fig. 2: Getting the data for cross-validation from the loader

the number of folds was reduced to 3 as a trade-off between time consumption and reducing error in the model. Figure 2 shows the training and validation sets being prepared for each fold. The cross validation data is drawn from the 'train' folder and not the 'test' folder.

As a GAN is suitable for creating realistic images, and an U-net specialises in biomedical image segmentation, the generator in the GAN was replaced with a U-net. Figures 3 and 4 show the individual layers in an U-Net in more detail. Figure 5 shows the U-net built from these components. It takes in an real image and returns a complex valued image as in preliminary testing, this produced better results than keeping the input and output channels consistent.

The U-Net is trained one slice at a time, so to help visualise the training, validation, and generated data, some utility functions were written. The device is set to the GPU if it exists on the machine the code is running on. Figure 6 shows the utility functions.

Crucially, in the GAN, a discriminator is required to distinguish between fake data drawn from the ground distribution and the ground distribution itself. It consists of several convolution layers, as convolutional layers preserve spatial

```python
""" Parts of the U-Net model """

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)
```

Fig. 3: U-Net parts 1

```python
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])
        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)
```

Fig. 4: U-Net parts 2

structure in images so the discriminator has the most important features to try to distinguish between a fake image and a real MRI image. As the discriminator is essentially performing a simple supervised classification, its structure is simple. At the end, a Sigmoid function is used to clamp the predicted value between 0 and 1, to indicate False and True. Figure 7 shows the structure used.

The models are trained on a variety of optimizers and loss functions to determine which has the most reliable and best performance. Pytorch does not have a built in SSIM loss function so the given code was amended to do such that, Figure 8.

The discriminator is first trained on the real data with the prediction being a ones tensor. The error is back-propagated, then it is trained on the fake data with a zeroes tensor. This helps it improve its prediction quickly as the fake data is generated by the generator, and it can learn the difference between the inferred distribution and the true distribution. The generator is trained in a mirror fashion, the discriminator is called upon the image it generates and the loss function is used on the prediction and a ones tensor, since that is what the generator wants to evoke from the discriminator. The error is back-propagated, and in one training loop, both networks feed into each other's improvement. This is depicted in Figures 9 ad 10.

The actual training involves iterating through the Training and Validation sets, defined and collected earlier when cross-validating, for each epoch, shown in Figures 11, 12 and 13.

For the Training set, the ground truth image is reduced to a real image and cropped to act as the truth. The undersampled raw data is passed through the inverse Fourier and made real by taking the absolute value, this gives the generator the low quality image to improve. The data is permuted for entry into

```
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):
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        x = self.up1(x5, x4)
        x = self.up2(x, x3)
        x = self.up3(x, x2)
        x = self.up4(x, x1)
        logits = self.outc(x)
        return logits

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

Fig. 5: U-Net

the neural network as the network expects the form (number, channels, height, width). The network is trained one slice at a time as that is how the data loaders have loaded the data. The generator takes in a real input and gives a complex output, when testing different channel sizes for inputs and outputs, this was seen to outperform the other choices by an average of 0.32 SSIM in the end result. The fake real image is computed and used to train both the discriminator and generator as mentioned earlier.

In the Validation loop, the process is very similar to the training loop. The real image is cropped and set as the truth, and the undersampled lower quality image is used as input for the generator. The difference is that the models are not trained on the generated image, and instead the SSIM values for measuring the similarity is computed. At the end of every fold of cross validation, the average SSIM value is printed, and at the end of every epoch, the average of the folds is printed.

```
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')

def show_single_tensor_slice(t):
    t=t.detach().cpu()
    show_single_slice(t.squeeze(0), cmap='gray')
    t.cuda()

device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
```

Fig. 6: Different ways of viewing slices and setting the device

## 4 Experiments

Several experiments were conducted by varying the loss function and optimizer to see which parameters had the best performance for the network.

Figure 15 plots the performance of different loss functions against each other for 10 epochs. The optimizer used was Adam. The crosses signify the average SSIM per epoch, while the dashes show performance per fold. The following loss functions were tested.

1. **L1 Loss:** The L1 loss calculates the Least Absolute Deviations, which minimizes the absolute difference between the real and the predicted values. This is a simple loss function to start off with and has a mean of 0.204 SSIM over the epochs and a std-dev of 0.070.
2. **MSE Loss:** The Minimum Square Errors loss is similar to the L1 loss, except it minimizes the square of the absolute difference to penalise larger distances. It had a comparable performance with the L1 loss with a mean of 0.207 and a std-dev of 0.09 suggesting that the data is richly clustered so taking the squares of the distance makes little difference.
3. **BCE Loss:** The Binary Cross Entropy works with a binary classifier, which is what a discriminator is. This, however, has the worst performance out of the loss functions with a mean of 0.145 and a std-dev of 0.029, which is understandable as the generator was using BCE as a loss function too, and that is not a binary classifier at all.
4. **BCE With Logits Loss:** BCE with Logits loss takes advantage of the log-sum-exp trick to be more stable than BCE, and it also combines BCE and a Sigmoid layer into one loss function. It has slightly better performance than regular BCE with a mean of 0.145 and a std-dev of 0.029 but it has a similar limitation to BCE in that it is not a suitable loss function for the generator.

```
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()
        )


    def forward(self, x):
        x = self.l1(x)
        x = self.l2(x)
        x = self.l3(x)
        x = self.l4(x)
        x = self.l5(x)
        x = self.l6(x)
        x = self.l7(x)
        x = self.l8(x)
        return x
discriminator = DiscriminatorNet()
discriminator.cuda()
```

Fig. 7: Discriminator Structure

```
from skimage.measure import compare_ssim
def ssim(gt, pred):
    """ Compute Structural Similarity Index Metric (SSIM). """
    gtn=gt.detach().numpy()
    predn=pred.detach().numpy()

    ssim_val= compare_ssim(gtn.transpose(1, 2, 0), predn.transpose(1, 2, 0), multichannel=True, data_range=gtn.max())
    ssim_val_numpy=np.asarray([ssim_val])
    ssim_val_tensor=torch.tensor(ssim_val_numpy,requires_grad=True)
    return ssim_val_tensor
loss = nn.BCELoss()
```

Fig. 8: The amended SSIM function and the Loss function

5. **SSIM:** The SSIM computes the structural similarity between two images. This was used because the SSIM is used as a measure of accuracy of reconstruction. The SSIM loss function had a mean SSIM of 0.17 over the epochs

```
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))
    error_real = loss(prediction_real.squeeze().squeeze().squeeze(), 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, 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
```

Fig. 9: Training the Discriminator

```
def train_generator(optimizer, fake_data, real_data):
    # Reset gradients
    optimizer.zero_grad()
    prediction = discriminator(fake_data.unsqueeze(0).unsqueeze(0))
    error = loss(prediction, ones_target(1)) #changed from prediction
    error.backward(retain_graph=True)
    fake_data.cuda()
    real_data.cuda()
    optimizer.step()
    return error
```

Fig. 10: Training the Generator

```
# Total number of epochs to train
num_epochs = 25
for epoch in range(num_epochs):
    ssim_error=[]
    loss_error=[]
    done=False
    fold=0
    for TrainSet, ValSet  in Data:
```

Fig. 11: The outer most loops for the epoch

and a std-dev of 0.0066. While this was very stable, the performance was not high enough to warrant its use.

6. **KLDivLoss:** This loss function was chosen because the Kullback-Leibler divergence Loss is good for measuring distance for continuous distributions

```
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)

    # 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()
    fs=T.center_crop(fs, [320, 320])
    gen_input=Variable(fs.unsqueeze(0).unsqueeze(0),requires_grad=True).cuda()
    fake_data_tensor = generator(gen_input)
    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())
```

Fig. 12: Training the Neural Network

```
    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()
        fs=T.center_crop(fs, [320, 320])
        gen_input=Variable(fs.unsqueeze(0).unsqueeze(0),requires_grad=True).cuda()
        fake_data_tensor = generator(gen_input)
        fake_data_tensor=fake_data_tensor.squeeze().permute(1,2,0)
        fake_data_img=T.complex_abs(fake_data_tensor)

        ssim_val=ssim(fake_data_img.unsqueeze(0).cpu(),real_data.unsqueeze(0).cpu())
        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)))
```

Fig. 13: Validating the Neural Network

Fig. 14: The loops within each epoch

and when regression is being done over a discretely sampled continuous distribution. The MRI image can be considered a continuous space where the undersampled data is the discretely sampled version of the continuous space. Unsurprisingly, the KLDivLoss had the best performance out of the loss functions, with a mean performance of 0.62 across the epochs and a std-dev of 0.0088. This loss function is very stable, which is surprising, given that the model is a GAN.

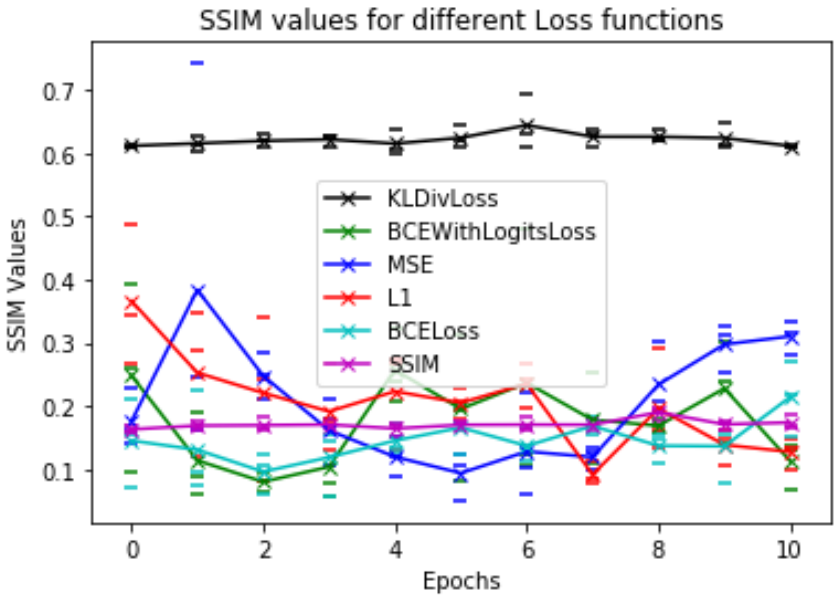


Fig. 15: A graph showing the SSIM Similarity between the real and generated image per epoch for 10 epochs, for different loss functions. The optimizer used is Adam for all.

The KLDivLoss had the best performance so tests were conducted by keeping it as a steady loss function, while varying the optimizer. Figure 16 shows the SSIM results of three optimizers.

1. **Adam:** Adam is an optimization which has an adaptive learning rate. It was initialised with a value of 0.0002; Adam will use squared gradients to scale the hyper-parameter while using the moving average of the gradient to find the minimum. This was used because Adam's update is not dependent on the gradient magnitude, and as the loss graph is expected to be unstable, Adam's update will help catch the saddle points and exit quicker. Adam had a mean SSIM of 0.75 and a std-dev of 0.0028.
2. **Adagrad:** Adagrad is another gradient based optimization method which uses an adaptive learning rate. it is well suited for sparse data, so its performance may not be competitive on MRI data but it did quite well with a mean of 0.748 and a std-dev of 0.009. Adam shares its advantages with Adagrad, so it not surprising that their performances are similar.
3. **AdamW:** AdamW is an improved version of Adam involving weight decay to keep track of regularization. This was an experiment to see if an improved version of Adam would give better performance and it didn't, with a mean of 0.67 and a std-dev of 0.0008.

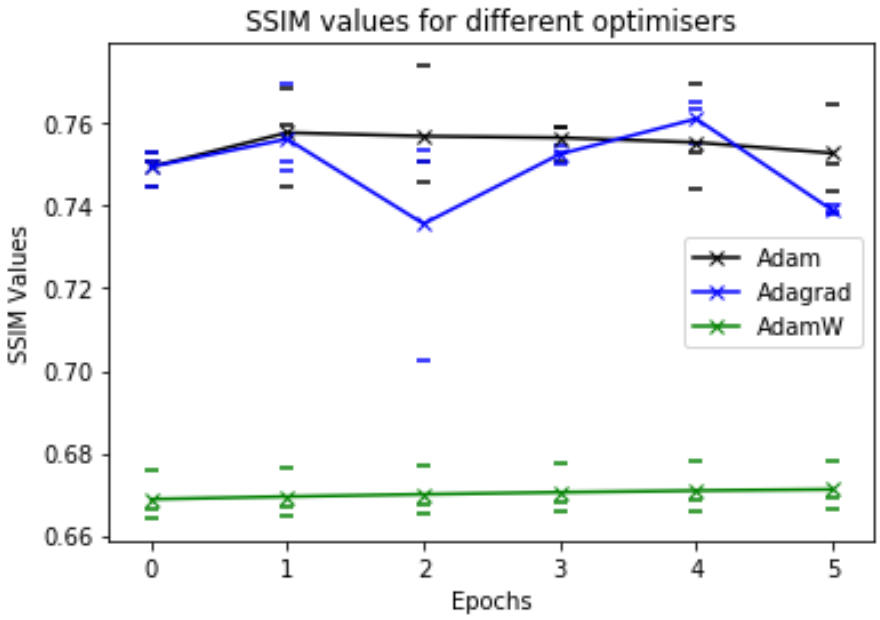


Fig. 16: A graph showing the SSIM Similarity between the real and generated image per epoch for 10 epochs, for different optimizers functions. The loss function used is KLDivLoss for all.

The performance for all three is quite high, with Adam and Adagrad performing better than AdamW. As Adam is more stable over the epochs than Adagrad, it was concluded that with our limited testing, that a combination of the Adam optimizer and the KLDivLoss Loss function would give the best results.

## 5 Conclusion

The key findings for the experiment is that the loss function impacts performance dramatically. Experimentation showed that the KLDivLoss function gave the best results, as expected from its theoretical description, along with the Adam optimizer. It can be argued however that the choice of optimizer did not influence performance too much as the experiments were ran for 10 epochs, and not for longer. The best performance achieved was an SSIM of 0.774.

## 6 Contribution

### 6.1 Temas Driver

Worked on writing different section of the report and also worked on the experiments section to create graphs. Was also responsible for conducting meetings.

### 6.2 Esha Dasgupta

Wrote the pipeline for generating the images, wrote and trained the GAN with the U-net network, did the experiments, wrote half of the implementation , wrote the experiments, and wrote the code for reconstructing the MRI images from the test set.

### 6.3 Mo Wang

Described and implemented about U-net.

### 6.4 Ehsun

Worked on different sections of writing the report. Built the VGG16 model but wasn't tested on due to time constraints so we went ahead with the U-Net and GAN architectures.

## References

1. Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.
2. John P Cunningham and Zoubin Ghahramani. Linear dimensionality reduction: Survey, insights, and generalizations. *The Journal of Machine Learning Research*, 16(1):2859–2900, 2015.
3. Deepa Kundur and Dimitrios Hatzinakos. Blind image deconvolution. *IEEE signal processing magazine*, 13(3):43–64, 1996.
4. Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.