```
import numpy as np
import argparse
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import random
import os
import h5py
from dataset_test import HDF5Dataset
from matplotlib import pyplot as plt
import torchvision.transforms as transforms
#from hdf5_io import save_hdf5
from torchvision.utils import save_image
from dcgan_test import Generator, Discriminator
from mpl_toolkits.mplot3d import Axes3D
from datetime import datetime, date, time
#import numpy as np
#np.random.seed(43)

# Set random seed for reproducibility.
seed = 500
random.seed(seed)
torch.manual_seed(seed)
print("Random Seed: ", seed)

parser = argparse.ArgumentParser()
parser.add_argument('--dataroot', default='', help='input dataset file')
parser.add_argument('--out_dir_hdf5', default='', help= 'output file for
generated images')
parser.add_argument('--out_dir_model', default='', help= 'output file for
model')
parser.add_argument('--workers', type=int, default=0, help='number of
workers')
parser.add_argument('--cuda', action='store_true', help='enables cuda')
parser.add_argument('--ngpu', type=int, default=1, help='number of GPUs to
use')
parser.add_argument('--bsize', default=32, help='batch size during training')
parser.add_argument('--imsize', default=64, help='size of training images')
parser.add_argument('--nc', default=2, help='number of channels')
parser.add_argument('--nz', default=100, help='size of z latent vector')
parser.add_argument('--ngf', default=64, help='size of feature maps in
generator')
parser.add_argument('--ndf', default=16, help='size of feature maps in
discriminator')
parser.add_argument('--nepochs', default=1000, help='number of training
epochs')
parser.add_argument('--lr', default=0.00002, help='learning rate for
optimisers')
parser.add_argument('--beta1', default=0.5, help='beta1 hyperparameter for
Adam optimiser')
parser.add_argument('--save_epoch', default=2, help='step for saving paths')
parser.add_argument('--sample_interval', default=50, help='output image step')
```

```python

opt = parser.parse_args()
cudnn.benchmark = True
timestamp = datetime.now()
str_timestamp = timestamp.strftime('%Y-%m-%d-%H-%M-%S')
opt.out_dir_hdf5 = 'reconstruction/results/' + str(opt.imsize) + 'morehdf5pores/img_out_new2'+ str_timestamp
opt.out_dir_model = 'reconstruction/results/' + str(opt.imsize) + 'moremodelpores/mod_out_new'+ str_timestamp
opt.dataroot = 'analyze_pore_samples/results/individual_pore_samples'
ngpu = int(opt.ngpu)
nz = int(opt.nz)
ngf = int(opt.ngf)
ndf = int(opt.ndf)
nc = int(opt.nc)
workers = int(opt.workers)

# Use GPU is available else use CPU.
device = torch.device("cuda:0" if(torch.cuda.is_available() and ngpu > 0) else "cpu")
print(device, " will be used.\n")

# Get the data.
dataset = HDF5Dataset(opt.dataroot,
                          input_transform=transforms.Compose([
                          transforms.ToTensor()
                          ]))

dataloader = torch.utils.data.DataLoader(dataset,
        batch_size=opt.bsize,
        shuffle=True, num_workers=workers)
sample_batch = next(iter(dataloader))

os.makedirs(str(opt.out_dir_hdf5), exist_ok=True)
os.makedirs(str(opt.out_dir_model), exist_ok=True)

###############################################
# Functions to be used:
###############################################
# weights initialisation
def weights_init(w):
    """
    Initializes the weights of the layer, w.
    """
    classname = w.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(w.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(w.weight.data, 1.0, 0.02)
        nn.init.constant_(w.bias.data, 0)

# save tensor into hdf5 format
def save_hdf5(tensor, filename):

    tensor = tensor.cpu()
```

```python
    ndarr = tensor.mul(255).byte().numpy()
    with h5py.File(filename, 'w') as f:
        f.create_dataset('data', data=ndarr, dtype="i8", compression="gzip")

###############################################

# Create the generator
netG = Generator(nz, nc, ngf, ngpu, size = int(opt.imsize)).to(device)

if('cuda' in str(device)) and (ngpu > 1):
    netG = nn.DataParallel(netG, list(range(ngpu)))

netG.apply(weights_init)
print(netG)

# Create the discriminator
netD = Discriminator(nz, nc, ndf, ngpu, size = int(opt.imsize)).to(device)

if('cuda' in str(device)) and (ngpu > 1):
    netD = nn.DataParallel(netD, list(range(ngpu)))

netD.apply(weights_init)
print(netD)

# Binary Cross Entropy loss function.
criterion = nn.BCELoss()

if(device.type == 'cuda'):
    netD.cuda()
    netG.cuda()
    criterion.cuda()

real_label = 0.9 # lable smoothing epsilon = 0.1
fake_label = 0

# Optimizer for the discriminator.
optimizerD = optim.Adam(netD.parameters(), lr=float(opt.lr), betas=(opt.beta1,
0.999))
# Optimizer for the generator.
optimizerG = optim.Adam(netG.parameters(), lr=float(opt.lr), betas=(opt.beta1,
0.999))

# Stores generated images as training progresses.
img_list = []
# Stores generator losses during training.
G_losses = []
# Stores discriminator losses during training.
D_losses = []

iters = 0
W = opt.imsize
H = opt.imsize
L = opt.imsize
print("Starting Training Loop...")
print("-"*25)
```

```python

for epoch in range(opt.nepochs):
    for i, data_labels in enumerate(dataloader, 0):
        ############################
        # (1) Update D network: maximise log(D(x)) + log(1 - D(G(z)))
        ###########################
        netD.zero_grad()

        data = data_labels[:, None, :, :, :]
        data = torch.cat((data, 1-data), dim = 1)

        real_data = data.to(device)
        #print('real', real_data.shape)

        b_size = real_data.size(0)
        #print(b_size)

        label = torch.full((b_size,), real_label, device=device)

        output = netD(real_data).view(-1)
        #output from D will be of size (b_size, 1, 1, 1, 1), with view(-1) we
        #reshape the output to have size (b_size)
        errD_real = criterion(output, label) # log(D(x))
        errD_real.backward()
        D_x = output.mean().item()


        noise = torch.randn(b_size, nz, 1, 1, 1, device=device)
        fake_data = netG(noise)
        label.data.fill_(fake_label)
        output = netD(fake_data.detach()).view(-1) # detach() no need for gradients
        #print(output.shape)
        errD_fake = criterion(output, label) # log(1 - D(G(z)))
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        errD = errD_real + errD_fake
        optimizerD.step()

        ############################
        # (2) Update G network: maximise log(D(G(z)))
        ###########################

        gen_it = 1
        while gen_it != 0:
            netG.zero_grad()
            label.data.fill_(real_label)
            noise = torch.randn(b_size, nz, 1, 1, 1, device=device)
            fake_data = netG(noise)
            #print(fake_data.shape)
            output = netD(fake_data).view(-1)
            errG = criterion(output,label) # log(D(G(z)))
            errG.backward()
            D_G_z2 = output.data.mean().item()
            optimizerG.step()
```

```python
            gen_it -= 1

        iters += 1

        # Check progress of training.
        if i%50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G(z)): %.4f / %.4f'
                  % (epoch, opt.nepochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

        # Save the losses for plotting.
        G_losses.append(errG.item())
        D_losses.append(errD.item())
        batches_done = epoch * len(dataloader) + i
        if i%50 == 0:
            plt.plot( G_losses, label = 'G loss')
            plt.plot( D_losses, label = 'D loss')
            plt.legend()
            plt.ylabel('loss')
            plt.xlabel(' Iterations')
            plt.savefig(str(opt.out_dir_hdf5)+'/dgloss.png')
            plt.clf()

        if batches_done % opt.sample_interval == 0:
            #fake = netG(noise)
            save_hdf5(fake_data.data, str(opt.out_dir_hdf5)+'/fake_{0}.hdf5'.format(batches_done))

###############################################################################
# This section can be included for saving the images produced at each timestep
# It increases the processing time more than three times, but is useful to view
# if the algorithm is producing reasonable images
        print(batches_done)
        if batches_done % opt.sample_interval == 0:
            print("SAVING")
            #fig = plt.figure()
          #  ax = fig.gca(projection='3d')

            output_data = fake_data.argmax(dim=1)
            plt.imshow(np.sum(output_data[0].cpu().numpy(), axis = 2))
            plt.savefig(str(opt.out_dir_hdf5)+'/2d_%d.png'%batches_done)
            plt.clf()
            plt.close('all')
            print("DONE SAVING")
###############################################################################

    if epoch % opt.save_epoch == 0:
        # Save checkpoints
        torch.save(netG.state_dict(), str(opt.out_dir_model)+'/netG_epoch_{}.pth'.format(epoch))
        torch.save(netD.state_dict(),
```

```
str(opt.out_dir_model)+'/netD_epoch_{}.pth'.format(epoch))
        torch.save(optimizerG.state_dict(), str(opt.out_dir_model)+'/optimG_epoch_{}.pth'.format(epoch))
        torch.save(optimizerD.state_dict(), str(opt.out_dir_model)+'/optimD_epoch_{}.pth'.format(epoch))

# Save the final trained model
torch.save(netG.state_dict(), str(opt.out_dir_model)+'/netG_final.pth'.format(epoch))
torch.save(netD.state_dict(), str(opt.out_dir_model)+'/netD_final.pth'.format(epoch))
torch.save(optimizerG.state_dict(), str(opt.out_dir_model)+'/optimG_final.pth'.format(epoch))
torch.save(optimizerD.state_dict(), str(opt.out_dir_model)+'/optimD_final.pth'.format(epoch))

```