BME 646/ ECE695DL: Homework 7

Bianjiang Yang

11 April 2022

1 Introduction

The main goal for homework7:
1. To develop and train a Generative Adversarial Network
2. To use the Large-scale CelebFaces Attributes (CelebA) Dataset for this homework; create "deep fakes" of the face images in the CelebA dataset

2 Methodology

Packages: torch, torch.nn, torch.nn.functional, torchvision.transforms, matplotlib.pyplot, copy, os, sys, glob, PIL, tqdm
Language: Python3
Tools: Anaconda3 virtual environment
System: Ubuntu 20.04.2 LTS (GNU/Linux 5.8.0-43-generic x86_64)

Instructions for running the code: The code I submitted has been modified to be runable, so you can simply run it by excuting the following script:
(1)train.py: CUDA_VISIBLE_DEVICES=1 python train.py
(2)validation.py: CUDA_VISIBLE_DEVICES=1 python -u validation.py

Other information: You should modify the savefig directory or just use the plt.show() function to show the plots.

3 Implementation and Results

network.py

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

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        layers = []
        latent_dim = 100
        curr_dim = 512
        layers.append(nn.ConvTranspose2d(latent_dim, curr_dim,
               kernel_size=4, stride=1, padding=0, bias=False)) # B*100*1*1 -> B*512*4*4
        layers.append(nn.InstanceNorm2d(curr_dim, affine=True))
        # layers.append(nn.BatchNorm2d(curr_dim, affine=True))
        layers.append(nn.ReLU(inplace=True))
        for i in range(3):  # B*512*4*4->B*512//8*32*32
            layers.append(nn.ConvTranspose2d(curr_dim, curr_dim // 2,
                    kernel_size=4, stride=2, padding=1, bias=False))
            layers.append(nn.InstanceNorm2d(curr_dim // 2, affine=True))
            # layers.append(nn.BatchNorm2d(curr_dim // 2, affine=True))
            layers.append(nn.ReLU(inplace=True))
            curr_dim = curr_dim // 2
        layers.append(nn.ConvTranspose2d(curr_dim, 3, kernel_size=4,
                stride=2, padding=1, bias=False))  # B*512//8*32*32 -> B*3*64*64
        # layers.append(nn.InstanceNorm2d(3, affine=True))
        # layers.append(nn.BatchNorm2d(3, affine=True))
        layers.append(nn.Tanh())
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        x = self.model(x)
        return x


from torch.nn.utils import spectral_norm
class Discriminator(nn.Module):
    def __init__(self, conv_dim=64, repeat_num=3):
        super(Discriminator, self).__init__()
        layers = []
        # layers.append(SpectralNorm(nn.Conv2d(3, conv_dim, kernel_size=4,
        # stride=2, padding=1))) # B*3*64*64 -> B*64*32*32
        layers.append(nn.Conv2d(3, conv_dim, kernel_size=4, stride=2,
                        padding=1, bias=False))
        # layers.append(nn.BatchNorm2d(conv_dim, affine=True))
        layers.append(nn.LeakyReLU(0.2, inplace=True))

        curr_dim = conv_dim
        for _ in range(repeat_num): #B*64*32*32 -> B*(64*2^3)*4*4
            # layers.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim*2,
            # kernel_size=4, stride=2, padding=1, bias=False)))
            layers.append(nn.Conv2d(curr_dim, curr_dim * 2, kernel_size=4,
                            stride=2, padding=1, bias=False))
            layers.append(nn.InstanceNorm2d(curr_dim * 2, affine=True))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            curr_dim = curr_dim * 2

        # layers.append(SpectralNorm(nn.Conv2d(curr_dim, 1, kernel_size=3,
        # stride=1, padding=1, bias=False))) #B*(64*2^5)*1*1 -> B*1*1*1
        layers.append(nn.Conv2d(curr_dim, 1, kernel_size=4, stride=1,
                            padding=0, bias=False))
        # layers.append(nn.BatchNorm2d(1, affine=True))
        layers.append(nn.Sigmoid())
        layers.append(nn.Flatten())

        self.main = nn.Sequential(*layers)

    def forward(self, x):
        x = self.main(x)
        return x
```

dataloader.py

```
import torch
import glob
# import os
from PIL import Image


class CelebA_Dataset(torch.utils.data.Dataset):
  'Characterizes a dataset for PyTorch'
  def __init__(self, datapath1, datapath2, transform):
        'Initialization'
        self.transform = transform
        self.list_IDs = sorted(glob.glob(datapath1 + '/*'))+\
                        sorted(glob.glob(datapath2 + '/*'))

  def __len__(self):
        'Denotes the total number of samples'
        return len(self.list_IDs)

  def __getitem__(self, index):
        'Generates one sample of data'
        img_path = self.list_IDs[index]
        X = self.transform(Image.open(img_path)).to(
            dtype = torch.float32)

        return X
```

train.py

```
import torch
import torch.nn as nn
import network
import torchvision.transforms as tvt
import matplotlib.pyplot as plt
import dataloader
import copy


def weights_init(m):
    """
    Uses the DCGAN initializations for the weights
    """
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('InstanceNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

adversarial_loss = nn.BCELoss()

def generator_loss(fake_output, label):
    gen_loss = adversarial_loss(fake_output, label)
    return gen_loss

def discriminator_loss(output, label):
    disc_loss = adversarial_loss(output, label)
    return disc_loss


def train(transform, device, lr1 = 1e-3, lr2 = 1e-3, epochs = 50, batch_size = 16,
          data_path1 = "/home/yangbj/695/hw7/data/Train",
          data_path2 = "/home/yangbj/695/hw7/data/Test",
          per = 500, save_path = "/home/yangbj/695/hw7/"):

    CelebA_data = dataloader.CelebA_Dataset(data_path1, data_path2, transform)
    # print(CelebA_data.list_IDs)
    train_data = torch.utils.data.DataLoader(CelebA_data,
                batch_size=batch_size, shuffle=True, num_workers=2)
    generator = copy.deepcopy(network.Generator())
    # copy.deepcopy(net)
    generator = generator.to(device)
    generator.apply(weights_init)
    discriminator = copy.deepcopy(network.Discriminator())
    discriminator = discriminator.to(device)
    discriminator.apply(weights_init)

    G_optimizer = torch.optim.Adam(generator.parameters(), lr=lr1, betas=(0.5, 0.999))
    D_optimizer = torch.optim.Adam(discriminator.parameters(), lr=lr2, betas=(0.5, 0.999))
    D_loss_list1, D_loss_list2, G_loss_list = [], [], []

    print("\nStarting training...")
    for epoch in range(epochs):
        for i, data in enumerate(train_data):
            real_img = data
            real_img = real_img.to(device)

            from torch.autograd import Variable
            noise = Variable(torch.randn(real_img.size(0),
                            100, 1, 1, device=device), requires_grad = True)
            zero_vector = Variable(torch.zeros(real_img.size(0),
                            1, device=device), requires_grad = True)
            one_vector = Variable(torch.ones(real_img.size(0),
                            1, device=device), requires_grad = True)#*0.9
            D_optimizer.zero_grad()
            # discriminator = discriminator.train()
            # generator = generator.eval()
            fake_img = generator(noise)
            # fake_img.requires_grad = True
            # print(fake_img.requires_grad)

            fake_output = discriminator(fake_img.detach())  # not train generator
            D_fake_loss = discriminator_loss(fake_output, zero_vector)
            D_fake_loss.backward()

            real_output = discriminator(real_img)
            D_real_loss = discriminator_loss(real_output, one_vector)
            D_real_loss.backward()
```

```
D_loss = D_real_loss.item() + D_fake_loss.item()
D_loss_list1.append(D_fake_loss.item())
D_loss_list2.append(D_real_loss.item())
D_optimizer.step()

# generator = generator.train()
# discriminator = discriminator.eval()
G_optimizer.zero_grad()
gan_output = discriminator(fake_img)
G_loss = generator_loss(gan_output, one_vector)
G_loss_list.append(G_loss.item())
G_loss.backward()
G_optimizer.step()
# print(G_loss)
# print(D_loss)

if (i + 1) % per == 0:
    print("[epoch:%d, batch:%5d, lr: %.9f] discriminator_loss: %.13f" %
          (epoch + 1, i + 1, lr1, D_fake_loss))  #sum(D_loss_list[epoch*(i-per+1) : epoch
    print("[epoch:%d, batch:%5d, lr: %.9f] generator_loss: %.13f" %
          (epoch + 1, i + 1, lr2, G_loss.item()))#sum(G_loss_list[epoch*(i-per+1) : epoch
    print("[epoch:%d, batch:%5d, lr: %.9f] discriminator_loss_total: %.13f" %
          (epoch + 1, i + 1, lr1, D_loss))

    plt.figure()
    plt.title('Discriminator Loss')
    plt.xlabel('Per '+ str(per)+ ' Iterations')
    plt.ylabel('Loss')
    plt.plot(D_loss_list1, label='Discriminator Loss')
    plt.legend(loc='upper right')
    plt.show()
    plt.savefig(save_path + "Discriminator_loss9_1" + ".jpg")

    plt.figure()
    plt.title('Discriminator Loss')
    plt.xlabel('Per ' + str(per) + ' Iterations')
    plt.ylabel('Loss')
    plt.plot(D_loss_list2, label='Discriminator Loss')
    plt.legend(loc='upper right')
    plt.show()
    plt.savefig(save_path + "Discriminator_loss9_2" + ".jpg")

    plt.figure()
    plt.title('Generator Loss')
    plt.xlabel('Per '+ str(per)+ ' Iterations')
```

```
                plt.ylabel('Loss')
                plt.plot(G_loss_list, label='Generator Loss')
                plt.legend(loc='upper right')
                plt.show()
                plt.savefig(save_path + "Generator_loss9" + ".jpg")
        torch.save(generator.state_dict(), save_path + 'generator9_'+str(epoch)+'.pth')
        torch.save(discriminator.state_dict(), save_path + 'discriminator9_'+str(epoch)+'.pth')
    return D_loss_list1, D_loss_list2, G_loss_list

if __name__ == '__main__':
    device  = torch.device('cuda:1')
    transform = tvt.Compose([tvt.ToTensor(), tvt.Normalize(
        (0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
    trainset_path1 = "/home/yangbj/695/hw7/Data/Train"
    trainset_path2 = "/home/yangbj/695/hw7/Data/Test"
    save_path = "/home/yangbj/695/hw7/"
    running_loss1, running_loss2, running_loss3 = train(
        transform = transform, device = device, lr1 = 2e-4, lr2 = 2e-4,
          epochs = 10, batch_size = 32, data_path1 = trainset_path1,
        data_path2 = trainset_path2, per = 50,
          save_path = save_path)
```

validation.py

```
import torch
import torch.nn as nn
import network
import torchvision.transforms as tvt
import matplotlib.pyplot as plt
import dataloader
import copy
import sklearn.metrics
import os
import sys
import glob
import seaborn
import cv2
from torchvision.utils import save_image

def test(transform, device, data_path, data_path2, save_path,
         generator_path, discriminator_path):
    CelebA_data = dataloader.CelebA_Dataset(data_path, data_path2, transform)
    test_data = torch.utils.data.DataLoader(CelebA_data, batch_size=1,
                                        shuffle=False, num_workers=2)
generator = network.Generator()
generator.load_state_dict(torch.load(generator_path))
generator = generator.to(device)
discriminator = network.Discriminator()
discriminator.load_state_dict(torch.load(discriminator_path))
discriminator = discriminator.to(device)
generator.eval()
discriminator.eval()

print("\n\nStarting testing...")
fake_pred = []
real_pred = []

for i, data in enumerate(test_data):
    real_img = data
    real_img = real_img.to(device)
    noise = torch.randn(real_img.size(0), 100, 1, 1, device=device)
    fake_img = generator(noise)
    fake_output = discriminator(fake_img).squeeze(0).squeeze(-1).squeeze(-1)
    real_output = discriminator(real_img).squeeze(0).squeeze(-1).squeeze(-1)
    if fake_output<0.5:
        fake_pred.append(1)
    else:
        fake_pred.append(0)
    if real_output>=0.5:
        real_pred.append(1)
    else:
        real_pred.append(0)
    save_image((fake_img[0]+1)/2, save_path + 'img' + str(i) + '.png')
    # for i in range(inputs.shape[0]):
    #     img = cv2.imread(img_path[i])
    #     cv2.rectangle(img, (bbox[i][0], bbox[i][1]),
    #                   (bbox[i][0]+bbox[i][2], bbox[i][1]+bbox[i][3]),
    #                   (0, 0, 255), 1)
    #     cv2.rectangle(img, (boundingbox[i][0], boundingbox[i][1]),
    #                   (boundingbox[i][0]+boundingbox[i][2],
    #                    boundingbox[i][1]+boundingbox[i][3]), (0, 255, 0), 1)
    #     if not os.path.exists(save_path+'tested_imgs/'+
    #                           str(labels[i]) +'/'):
    #         os.makedirs(save_path+'tested_imgs/'+
    #                    str(labels[i]) +'/')
    #     cv2.imwrite(save_path+'tested_imgs/'+
    #                 str(labels[i]) +'/'+img_path[i][-16:], img)
    #     print("save an img to:" + save_path+
    #           'tested_imgs/'+str(labels[i]) +'/'+img_path[i][-16:])
    # calculate confusion matrix with sklearn module
    # confus_matrix = sklearn.metrics.confusion_matrix(
    #     label_total, output_total, labels=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    # print(confus_matrix)

    # classification accuracy calculation
    # acc = 0
    # for i in range(confus_matrix.shape[0]):
    #     acc += confus_matrix[i][i]
    # Accuracy = acc / confus_matrix.sum() * 100
    fake_acc = sum(fake_pred)/len(fake_pred)*100
    real_acc = sum(real_pred) / len(real_pred) * 100

    print('Fake Image Classification Accuracy: '+str(fake_acc)+'%')
    print('Real Image Classification Accuracy: ' + str(real_acc) + '%')

    # plot confusion matrix
    # plt_labels = []
    # path_list = sorted(glob.glob(data_path + '/*'))
    # for p in path_list:
    #     plt_labels.append(os.path.splitext(os.path.basename(p))[0])
    # plt.figure(figsize = (10,7))
    # seaborn.heatmap(confus_matrix, annot=True, fmt= 'd', linewidths = .5,
    #                 xticklabels= plt_labels, yticklabels= plt_labels)
    # plt.title("Net" + name + " " + 'Accuracy:'+str(Accuracy)+'%')
    # plt.savefig(save_path + "net_"+name+"confusion_matrix.jpg")



if __name__ == '__main__':

    # settings
    device  = torch.device('cuda:1')
    transform = tvt.Compose([tvt.ToTensor(),
                             tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
    dataset_path = "/home/yangbj/695/hw7/Data/Test"
    save_path = "/home/yangbj/695/hw7/Results7/"
    generator_path = "/home/yangbj/695/hw7/generator7_9.pth"
    discriminator_path = "/home/yangbj/695/hw7/discriminator7_9.pth"
    # batch_size = 1
    # name = '1'
    # resize = [128, 128]
    # net = network.Single_LOAD_Net()

    # Run test
```

```
test(transform = transform, device = device,
     data_path = dataset_path,
     data_path2="/home/yangbj/695/hw7/Data/Train",
     save_path = save_path, generator_path = generator_path,
     discriminator_path=discriminator_path)
```

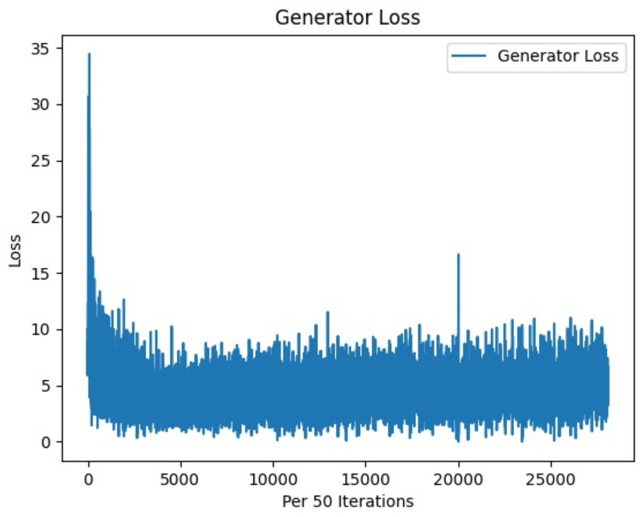

Figure 1: Generator Loss

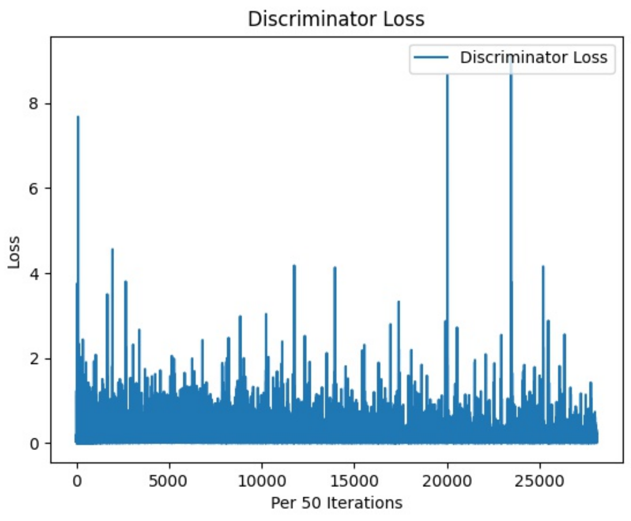

Figure 2: Discriminator Fake Loss

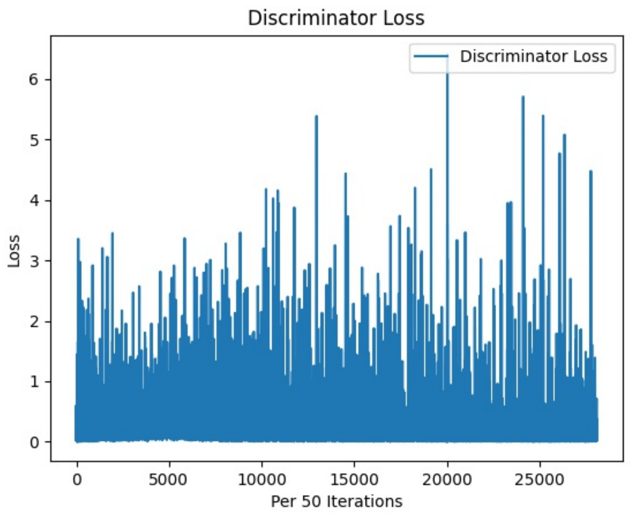

Figure 3: Discriminator Real Loss

Train Loss

The results in Fig. 1, Fig. 2 and Fig. 3 above show that:
(1) The generator loss is always decreasing, indicating that the generator is converging.
(2) The discriminator loss 1 and 2 are both unstable but the discriminator is surely helping train the generator.

Framework

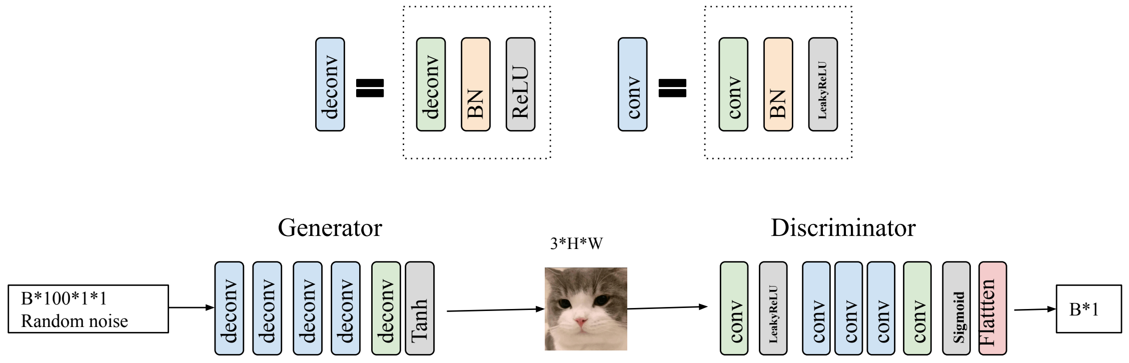

Figure 4: The framework: The generator and discriminator. Generator contains a deep feature extraction stream with a 5 transposed convolutional layers, a ReLU is utilized after each normalization layer, expect that a Tanh is utilized for output to constrain it in the range (-1,1); The discriminator contains 5 convolution layers with a LeakyReLU with negative slop 0.2 is utilized after each normalization layer expecet that a sigmoid is utilized at the end of the discriminator for BCE Loss. I utilized instancenorm for the generator and the discriminator although spectral norm should be better for the discriminator.

Deep Fake Face samples: As shown in Fig. 5, the first 2 rows are deep fakes; the last 2 rows are the true face images.

Implementation Details. Batch size=32; Generator: contains a deep feature extraction stream with a 5 transposed convolutional layers, a ReLU is utilized after each normalization layer, expect that a Tanh is utilized for output to constrain it in the range (-1,1); The discriminator contains 5 convolution layers with a LeakyReLU with negative slop 0.2 is utilized after each normalization layer expecet that a sigmoid is utilized at the end of the discriminator for BCE Loss. I utilized instancenorm for the generator and the discriminator although spectral norm should be better for the discriminator. I choose Adam optimizer with learning rate 2×10^{-4}, beta1=0.5, beta2=0.999. I initialize the convolution/tranposed conv/ normalization layers with Normal initialization.

4 Lessons Learned

The hurdles faced and the techniques employed to overcome them:

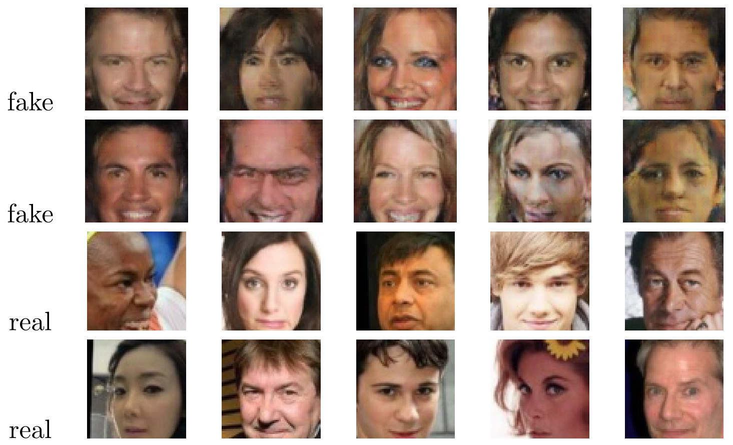

Figure 5: Testing samples

(1) For dataloader,
Nothing special.

(2) For training,
When training the discriminator, the fake images inputed should be detached so that the gradient of the generator will not be updated.

(3) For network,
BatchNorm and InstanceNorm are both acceptable for the networks. However, it seems spectral_norm is not okay for the discriminator.

(4) For validation,
When saving images, one should transform the value range from (-1,1) to (0,1).

5 Suggested Enhancements

This project helps me focus on the train part and network design part. I think it's more meaningful and I can learn a lot from implementing training GANs but not struggling in the dataloader.