ECE695DL: Homework 7

Qilei Zhang

Due Date: Monday, Apr 11, 2022

1 Introduction

The goal of this homework is to understand the usage of generative adversarial network (GAN). GANs are a framework for teaching a DL model to capture the training data's distribution so model can generate new data from that same distribution. They are made of two distinct models, a generator and a discriminator. Particularly, generator code will use *Transpose Convolutions* to expand noise vectors into images that will look very similar to the images in the training data. The homework will use CelebFaces Attributes Dataset that contains thousands of celebrity images. The expected objective of the homework is to produce images will look very similar to the training images without being exactly the same as any of them.

2 Methodology

2.1 About Preparation

The basic information for the program shall be entered in the `FakeArg` class. `self.img_path` stores the downloaded images for training and the `batch_size` is the batch size settings. Here is the example scripts:

```python
class FakeArg:
    def __init__(self):
        self.img_path = './ECE695/hw7/CelebA/Data/Train/'
        self.batch_size = 24
```

`Dataloader` reads the images in the root folder. It will determine whether the set batch size will produce a remainder for the total number of training images. It will return the transformed normalized image tensor. Validation will also use the same `Dataloader` and `FakeArg`. Only the `FakeArg.img_pth` need to change to test image folder.

2.2 About Net Design

Two networks exists in the model, i.e., discriminator and generator. Discriminator uses the bases from previous homework. Here, discriminator takes a 3x64x64 input image. It contains two convolution layers and two fully-connected layers. It also applied a sigmoid activation function before returning. Discriminator is a binary classification network that takes an image as input and outputs a scalar probability that the input image is real (as opposed to fake). The code of generator is from Professor Kak's `GeneratorDG1` Net in `AdversarialLearning`. Generator takes a vector with the length `nz`. It has five transpose convolutional layers and four batch normalization layers. It will return the output in the range of $[-1, 1]$ after applying `tanh` activation.

2.3 About GAN

The training of GAN nework has two parts, which are discriminator training and generator training. First, the goal of training the discriminator is to maximize the probability of correctly classifying a given input as real or fake. Specifically, it want to maximize

$$\log(D(x)) + \log(1 - D(G(z)))$$

A batch of real samples from the training set calculate the loss $\log(D(x))$. A batch of fake samples computes the second term $\log(1 - D(G(z)))$. On the other hand, the goal of the generator is to minimize $\log(1 - D(G(z)))$ in order to generate better fakes. In summary the loss function is

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

It will use the Binary Cross Entropy loss (BCELoss) function as a criterion.

3 Implementation and Results

3.1 Main Program Code

3.1.1 hw07_training.py

```python
import glob
import torch
import torchvision
import torch.utils.data

import imageio  # For gif generation
from PIL import Image
from torch.utils.data import DataLoader, Dataset

import numpy as np
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as functional
import torchvision.transforms as tvt
import torchvision.transforms.functional as tvtF
```

```python
import matplotlib.pyplot as plt


class FakeArg:
    """
    Put the fundamental information
    """

    def __init__(self):
        self.img_path = '/Users/xxx/CelebA/Data/Train/'
        self.batch_size = 24


class DatasetClass(Dataset):
    def __init__(self, root, batch, trans=None):
        self.root = root
        self.img_info = []
        self.batch = batch
        self.transform = trans

        search = self.root + "*"
        for img in glob.glob(search):
            self.img_info.append(img)

        if len(self.img_info) % self.batch != 0:
            remain = len(self.img_info) % self.batch
            self.img_info = self.img_info[:-remain]

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

    def __getitem__(self, idx):
        img_path = self.img_info[idx]
        image = Image.open(img_path)
        im_ts = self.transform(image)
        return im_ts


class DisNet(nn.Module):
    def __init__(self):
        super(DisNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 128, 3, padding=1)
        self.conv2 = nn.Conv2d(128, 128, 3)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(128 * 15 * 15, 1000)
        self.fc2 = nn.Linear(1000, 1)
```

```python
        self.sig = nn.Sigmoid()

    def forward(self, x):
        x = self.pool(functional.relu(self.conv1(x)))
        x = self.pool(functional.relu(self.conv2(x)))
        x = x.view(-1, 128 * 15 * 15)
        x = functional.relu(self.fc1(x))
        x = self.fc2(x)
        x = self.sig(x)
        return x


# This network structure is referred from Prof. Kak's GeneratorDG1 Net.
class GenNet(nn.Module):
    def __init__(self):
        super(GenNet, self).__init__()
        self.latent_to_image = nn.ConvTranspose2d(200, 512, kernel_size=4,
        ↪  stride=1, padding=0, bias=False)
        self.upsampler2 = nn.ConvTranspose2d(512, 256, kernel_size=4,
        ↪  stride=2, padding=1, bias=False)
        self.upsampler3 = nn.ConvTranspose2d(256, 128, kernel_size=4,
        ↪  stride=2, padding=1, bias=False)
        self.upsampler4 = nn.ConvTranspose2d(128, 64, kernel_size=4,
        ↪  stride=2, padding=1, bias=False)
        self.upsampler5 = nn.ConvTranspose2d(64, 3, kernel_size=4,
        ↪  stride=2, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(512)
        self.bn2 = nn.BatchNorm2d(256)
        self.bn3 = nn.BatchNorm2d(128)
        self.bn4 = nn.BatchNorm2d(64)
        self.tanh = nn.Tanh()

    def forward(self, x):
        x = self.latent_to_image(x)
        x = torch.nn.functional.relu(self.bn1(x))
        x = self.upsampler2(x)
        x = torch.nn.functional.relu(self.bn2(x))
        x = self.upsampler3(x)
        x = torch.nn.functional.relu(self.bn3(x))
        x = self.upsampler4(x)
        x = torch.nn.functional.relu(self.bn4(x))
        x = self.upsampler5(x)
        x = self.tanh(x)
        return x
```

```python
#  The structure of this code is taken from Professor Kak's AdversarialLearning.
def run_code_for_training(dis_net, gen_net, trainLoader, bs, learning_rate=1e-5,
                          momentum_set=0.9, epochs=20, device='cuda:0'):
    beta1 = 0.5  # ?
    nz = 200  # Change
    iteration_count = 0
    dnet = dis_net.to(device)
    gnet = gen_net.to(device)
    fixed_noise = torch.randn(bs, nz, 1, 1, device=device)
    real_label = 1
    fake_label = 0
    criterion = nn.BCELoss()
    dis_optimizer = optim.Adam(dnet.parameters(), lr=learning_rate, betas=(beta1, 0.999))
    gen_optimizer = optim.Adam(gnet.parameters(), lr=learning_rate, betas=(beta1, 0.999))

    dloss_record = []
    gloss_record = []
    fake_img_list = []

    for epoch in range(epochs):
        g_losses_per_print_cycle = []
        d_losses_per_print_cycle = []
        for i, data in enumerate(trainLoader):
            if i % 10 == 0:
                print(i)
            dnet.zero_grad()  # ?
            real_img_inputs = data
            real_img_inputs = real_img_inputs.to(device)

            # Loss for real
            label = torch.full((bs,), real_label, dtype=torch.float, device=device)
            output = dnet(real_img_inputs).view(-1)
            dis_loss_for_reals = criterion(output, label)
            dis_loss_for_reals.backward()

            # max operation
            noise = torch.randn(bs, nz, 1, 1, device=device)
            fakes = gnet(noise)
            label.fill_(fake_label)
            output = dnet(fakes.detach()).view(-1)
            dis_loss_for_fakes = criterion(output, label)
```

```python
            dis_loss_for_fakes.backward()
            dis_loss = dis_loss_for_reals + dis_loss_for_fakes
            d_losses_per_print_cycle.append(dis_loss)
            dis_optimizer.step()

            # max-min optimization
            gnet.zero_grad()
            label.fill_(real_label)
            output = dnet(fakes).view(-1)
            gen_loss = criterion(output, label)
            g_losses_per_print_cycle.append(gen_loss)
            gen_loss.backward()
            gen_optimizer.step()

            dloss_record.append(dis_loss.item())
            gloss_record.append(gen_loss.item())

            if (i + 1) % 100 == 0:
                mean_d_loss = torch.mean(torch.FloatTensor(d_losses_per_print_cycle))
                mean_g_loss = torch.mean(torch.FloatTensor(g_losses_per_print_cycle))
                print("\n[epoch:%d, batch:%5d] mean_D_loss=%7.4f mean_G_loss=%7.4f" %
                      (epoch + 1, i + 1, mean_d_loss.item(), mean_g_loss.item()))
                d_losses_per_print_cycle = []
                g_losses_per_print_cycle = []

            if (iteration_count % 200 == 0) or ((epoch == epochs - 1) and (i == len(trainLoader) - 1)):
                with torch.no_grad():
                    fake = gnet(fixed_noise).detach().cpu()
                fake_img_list.append(torchvision.utils.make_grid(fake, padding=1, pad_value=1, normalize=True))
            iteration_count += 1

    dis_net_path = './dis_net.pth'
    gen_net_path = './gen_net.pth'
    torch.save(dnet.state_dict(), dis_net_path)
    torch.save(gnet.state_dict(), gen_net_path)
    print('Finished Training')

    return dloss_record, gloss_record, fake_img_list
```

```python
def plot_trained_fake_img(img_list, train_dataloader, device):
    images_list = []
    for fake_img in img_list:
        image = tvtF.to_pil_image(fake_img)
        image_array = np.array(image)
        images_list.append(image_array)
    imageio.mimsave("generation_animation.gif", images_list, fps=5)

    plt.subplot(1, 1, 1)
    plt.axis("off")
    plt.title("Fake Images")
    plt.imshow(np.transpose(img_list[-1], (1, 2, 0)))
    plt.savefig("real_vs_fake_images.png")
    plt.show()


if __name__ == '__main__':
    # Training on GPU or CPU
    if torch.cuda.is_available():
        device = 'cuda:0'
    else:
        device = 'cpu'

    # Basic Information
    args = FakeArg()

    # Load and normalize data
    transform = tvt.Compose([tvt.ToTensor(), tvt.Normalize((0.5, 0.5, 0.5),
     ↪  (0.5, 0.5, 0.5))])

    trainSet = DatasetClass(args.img_path, args.batch_size, transform)
    trainLoader = DataLoader(dataset=trainSet, batch_size=args.batch_size,
     ↪  shuffle=True, num_workers=0)
    print('Loader Created')
    print(len(trainLoader), 'images')

    discriminator = DisNet()
    generator = GenNet()
    print('Net Created')

    DNetLoss, GNetLOSS, FakeImg = run_code_for_training(discriminator,
     ↪  generator, trainLoader, args.batch_size,
                                                        learning_rate=1e-4,
                                                         ↪  momentum_set=0.9,
                                                         ↪  epochs=50,
                                                         ↪  device=device)
```

```python

    plt.figure(figsize=(10, 5))
    plt.title("Generator and Discriminator Loss During Training")
    plt.plot(DNetLoss, label="D")
    plt.plot(GNetLOSS, label="G")
    plt.xlabel("iterations")
    plt.ylabel("Loss")
    plt.legend()
    plt.savefig("loss_training.png")
    plt.show()

    plot_trained_fake_img(FakeImg, trainLoader, device)

    print('Done')
```

3.1.2 hw07_validation.py

```python
import copy
import torch
import torchvision
import torch.nn as nn
import matplotlib.pyplot as plt
import torchvision.transforms as tvt
from torch.utils.data import DataLoader
from hw07_training import FakeArg, DatasetClass, DisNet, GenNet,
↪ plot_trained_fake_img


def validation(dis_net, gen_net, val_loader, bs, device="cpu"):
    dis_net = copy.deepcopy(dis_net)
    gen_net = copy.deepcopy(gen_net)
    dnet = dis_net.to(device)
    gnet = gen_net.to(device)
    nz = 200
    fixed_noise = torch.randn(bs, nz, 1, 1, device=device)
    real_label = 1
    fake_label = 0
    criterion = nn.BCELoss()

    fake_img_list = []
    dloss_record = []
    gloss_record = []
    for i, data in enumerate(val_loader):
        print(i)
        real_img_inputs = data
        real_img_inputs = real_img_inputs.to(device)
```

```python

        # Loss for real
        label = torch.full((bs,), real_label, dtype=torch.float, device=device)
        output = dnet(real_img_inputs).view(-1)
        dis_loss_for_reals = criterion(output, label)

        # max operation
        noise = torch.randn(bs, nz, 1, 1, device=device)
        fakes = gnet(noise)
        label.fill_(fake_label)
        output = dnet(fakes.detach()).view(-1)
        dis_loss_for_fakes = criterion(output, label)
        dis_loss = dis_loss_for_reals + dis_loss_for_fakes

        # max-min optimization
        label.fill_(real_label)
        output = dnet(fakes).view(-1)
        gen_loss = criterion(output, label)

        dloss_record.append(dis_loss.item())
        gloss_record.append(gen_loss.item())

        if (i % 200 == 0) or (i == len(val_loader) - 1):
            with torch.no_grad():
                fake = gnet(fixed_noise).detach().cpu()
            fake_img_list.append(torchvision.utils.make_grid(fake, padding=1, pad_value=1, normalize=True))

    return dloss_record, gloss_record, fake_img_list


if __name__ == '__main__':
    print('start')
    if torch.cuda.is_available():
        device = 'cuda:0'
    else:
        device = 'cpu'

    # Basic Information
    args = FakeArg()
    args.img_path = '/Users/XXXX/CelebA/Data/Test/'

    transform = tvt.Compose([tvt.ToTensor(), tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

```

```python
    valSet = DatasetClass(args.img_path, args.batch_size, transform)
    valLoader = DataLoader(dataset=valSet, batch_size=args.batch_size, shuffle=False, num_workers=0)
    print('Loader Created')

    discriminator = DisNet()
    generator = GenNet()
    print('Net Created')

    discriminator.load_state_dict(torch.load("dis_net.pth", map_location=torch.device(device)))
    discriminator.eval()
    generator.load_state_dict(torch.load("gen_net.pth", map_location=torch.device(device)))
    generator.eval()
    DNetLoss, GNetLOSS, FakeImg = validation(discriminator, generator, valLoader, args.batch_size, device=device)

    print('Validation Finished, Starting Plots')

    plt.figure(figsize=(10, 5))
    plt.title("Generator and Discriminator Loss During Testing")
    plt.plot(DNetLoss, label="D")
    plt.plot(GNetLOSS, label="G")
    plt.xlabel("iterations")
    plt.ylabel("Loss")
    plt.legend()
    plt.savefig("loss_testing.png")
    plt.show()

    plot_trained_fake_img(FakeImg, valLoader, device)

    print('Done')
```

3.2 Results

The output are shown in following pages.

3.2.1 Training loss

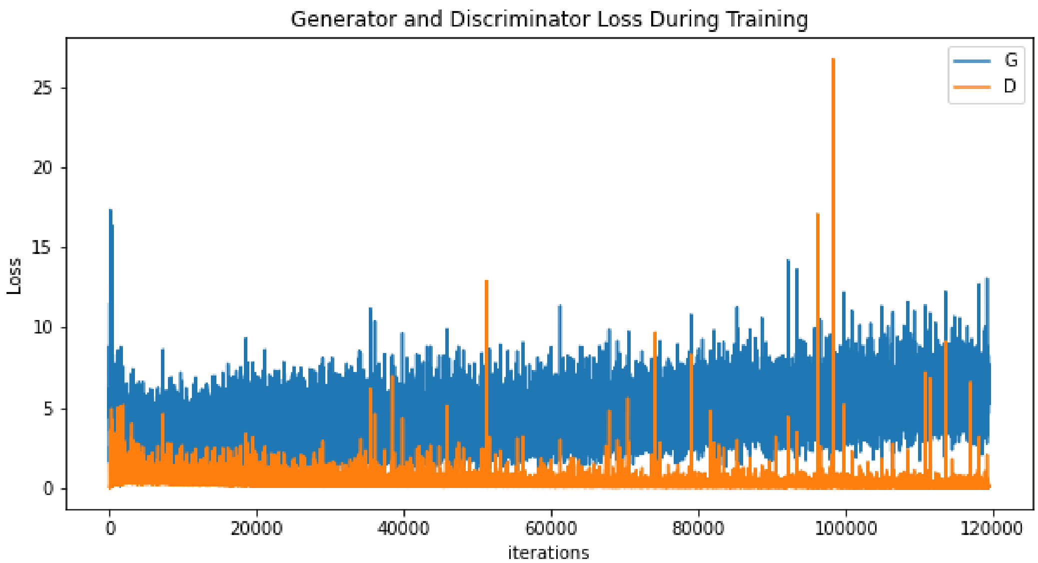

Figure 1: Training loss with learning rate 1e-4 and 50 epochs.

3.2.2 Validation loss

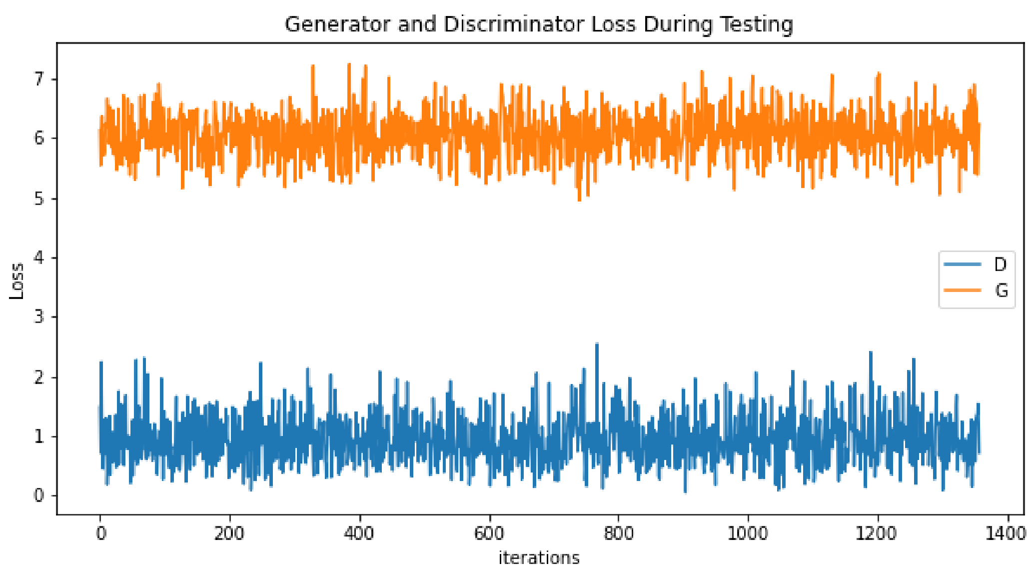

Figure 2: Validation Loss.

3.2.3 Real Images

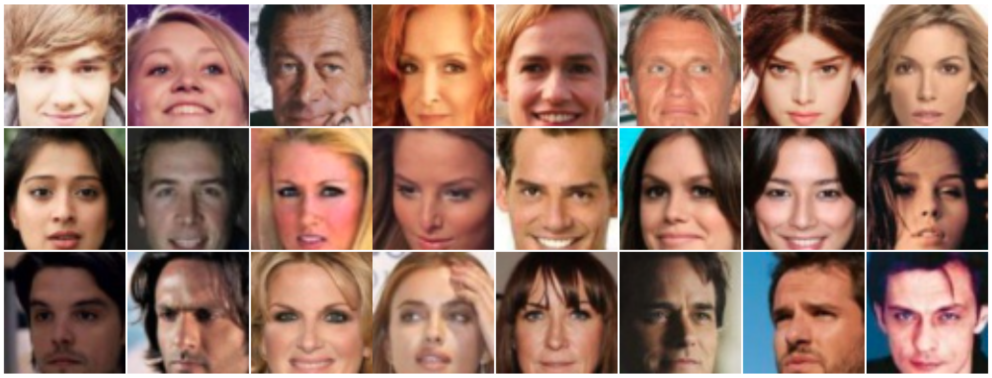

Figure 3: Real Images.

3.2.4 Fake Training Images

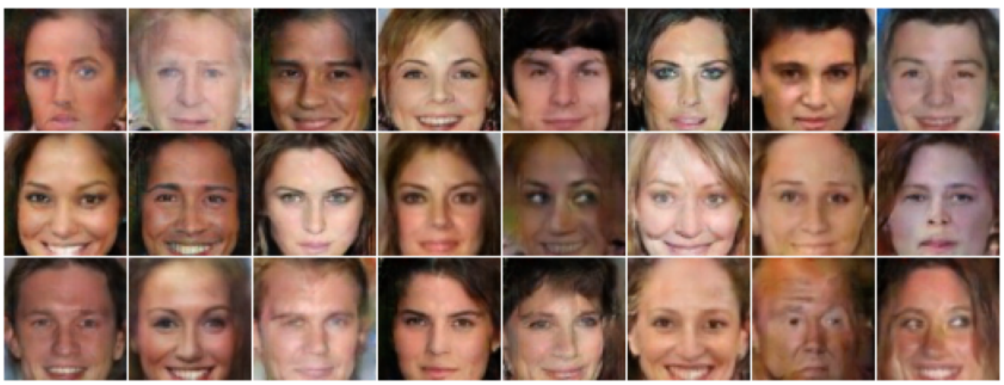

Figure 4: Fake Training Images Output.

3.2.5 Fake Test Images

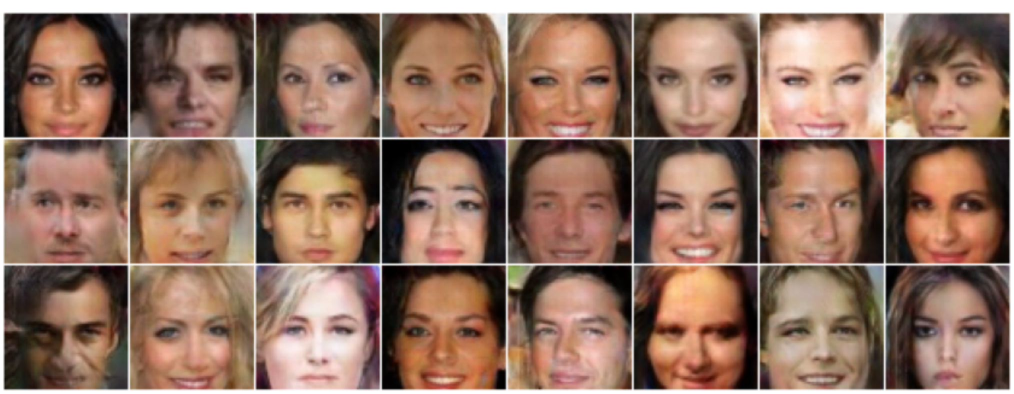

Figure 5: Fake Test Images Output.

4 Lessons Learned

1. It is expected that at the end of training (very large epochs) generator can always fool discriminator. However, when I tried a very large epochs, sometimes it not works as the small epochs, i.e., after some moment generator starts to diverge. Probably, the problem is that the discriminator overfits.

2. Google Drive operations can time out when the number of files or subfolders in a folder grows too large. This may bring additional process time to let dataloader read the images. The solution is to move files contained in one folder into sub-folders.

3. Two neural networks contest with each other in a game (in the form of a zero-sum game, where one agent's gain is another agent's loss).

5 Suggested Enhancements

1. Add checkpoints to ensure the unexpected situation such as network interruption. In that way, training could continue from the break point. Otherwise, the training needs to start again.

2. Try to train a network fed with specific labels, and generate a image with the appointed label.

3. Try training on higher pixel photos.