BME646 and ECE 60146 – Homework #8

Nadine Amin

GAN:

Discriminator Network

Figure 1 shows the implementation of the discriminator network as per the *DiscriminatorDG1* class in Prof. Kak's AdversarialLearning.py (https://engineering.purdue.edu/kak/distDLS/). The network has 4 convolutional layers with different numbers of channels, each a kernel size of 4, a stride of 2, and a padding of 1 (4-2-1). It also has an output convolutional layer with a single node as the discriminator's output for each image. Lastly, it has 3 batch normalization layers with different numbers of channels and a Sigmoid activation function to output a probability. In the *forward* function, the input is first passed through the 1st convolutional layer, followed by a leaky ReLU activation function to avoid the dying ReLU problem. Next, the output is passed through the 2nd convolutional layer, the 1st batch normalization layer, and a leaky ReLU. This is then repeated for the 3rd and 4th convolutional layers. Lastly, the output is passed through the last convolutional layer, followed by the Sigmoid activation function to output the discriminator's probability for each image.

```python
# a class for the discriminator network
# copied from the DiscriminatorDG1 class in Prof. Kak's AdversarialLearning.py
# (https://engineering.purdue.edu/kak/distDLS/) with added comments
class disc(nn.Module):
  """
  This is an implementation of the DCGAN Discriminator. I refer to the DCGAN network topology as
  the 4-2-1 network.  Each layer of the Discriminator network carries out a strided
  convolution with a 4x4 kernel, a 2x2 stride and a 1x1 padding for all but the final
  layer. The output of the final convolutional layer is pushed through a sigmoid to yield
  a scalar value as the final output for each image in a batch.

  Class Path:  AdversarialLearning  ->  DataModeling  ->  DiscriminatorDG1
  """
  def __init__(self):
    super(disc, self).__init__()
    # 4 convolutional layers (different numbers of channels) with a kernel size of 4, stride of 2, and padding of 1
    self.conv_in = nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1)
    self.conv_in2 = nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1)
    self.conv_in3 = nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1)
    self.conv_in4 = nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1)
    # 1 convolutional layer with a single node as the discriminator's output for each image
    self.conv_in5 = nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=0)
    # 3 batch normalization layers with different numbers of channels
    self.bn1  = nn.BatchNorm2d(128)
    self.bn2  = nn.BatchNorm2d(256)
    self.bn3  = nn.BatchNorm2d(512)
    # a Sigmoid activation function to output a probability
    self.sig = nn.Sigmoid()

  def forward(self, x):
    # pass input into 1st convolutional layer + leaky ReLU activation (to avoid dying ReLU problem)
    x = torch.nn.functional.leaky_relu(self.conv_in(x), negative_slope=0.2, inplace=True)
    # pass through 2nd convolutional layer + 1st batch normalization + leaky ReLU
    x = self.bn1(self.conv_in2(x))
    x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
    # pass through 3rd convolutional layer + 2nd batch normalization + leaky ReLU
    x = self.bn2(self.conv_in3(x))
    x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
    # pass through 4th convolutional layer + 3rd batch normalization + leaky ReLU
    x = self.bn3(self.conv_in4(x))
    x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
    # pass through last convolutional layer + Sigmoid activation function to output probability
    x = self.conv_in5(x)
    x = self.sig(x)
    return x
```

Figure 1

Generator Network

Figure 2 shows the implementation of the generator network as per the *GeneratorDG1* class in Prof. Kak's AdversarialLearning.py (https://engineering.purdue.edu/kak/distDLS/). The network has a transpose convolutional layer that takes in the 1x1 noise input of 100 channels. Next, it has 3 transpose convolutional layers with different numbers of channels, each a kernel size of 4, stride of 2, and padding of 1 (4-2-1). It also has an output convolutional layer with 3 output channels resembling the generator's generated fake image. Lastly, it has 4 batch normalization layers with different numbers of channels and a Tanh activation function. In the *forward* function, the input is first passed through the 1st transpose convolutional layer, followed by the 1st batch normalization and a ReLU activation function. Next, the output is passed through the 3 transpose convolutional layers, each followed by a batch normalization layer and a ReLU activation function. Lastly, the output is passed through the final transpose convolutional layer and the Tanh activation function outputting the generated fake image (with 3 channels).

```python
# a class for the generator network
# copied from the GeneratorDG1 class in Prof. Kak's AdversarialLearning.py
# (https://engineering.purdue.edu/kak/distDLS/) with added comments
class gen(nn.Module):
  """
  This is an implementation of the DCGAN Generator. As was the case with the Discriminator network,
  you again see the 4-2-1 topology here.  A Generator's job is to transform a random noise
  vector into an image that is supposed to look like it came from the training dataset. (We refer
  to the images constructed from noise vectors in this manner as fakes.)  As you will see later
  in the "run_gan_code()" method, the starting noise vector is a 1x1 image with 100 channels.  In
  order to output 64x64 output images, the network shown below use the Transpose Convolution
  operator nn.ConvTranspose2d with a stride of 2.  If (H_in, W_in) are the height and the width
  of the image at the input to a nn.ConvTranspose2d layer and (H_out, W_out) the same at the
  output, the size pairs are related by
              H_out   =   (H_in - 1) * s   +   k   -   2 * p
              W_out   =   (W_in - 1) * s   +   k   -   2 * p
  were s is the stride and k the size of the kernel.  (I am assuming square strides, kernels, and
  padding). Therefore, each nn.ConvTranspose2d layer shown below doubles the size of the input.
  Class Path:  AdversarialLearning  ->  DataModeling  ->  GeneratorDG1
  """
  def __init__(self):
    super(gen, self).__init__()
    # a transpose convolutional layer taking in a 1x1 input of 100 channels
    self.latent_to_image = nn.ConvTranspose2d(100, 512, kernel_size=4, stride=1, padding=0, bias=False)
    # 3 transpose convolutional layers (different numbers of channels) with a kernel size of 4, stride of 2, and padding of 1
    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)
    # 1 transpose convolutional layer with 3 output channels resembling the generator's fake image
    self.upsampler5 = nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1, bias=False)
    # 4 batch normalization layers with different numbers of channels
    self.bn1 = nn.BatchNorm2d(512)
    self.bn2 = nn.BatchNorm2d(256)
    self.bn3 = nn.BatchNorm2d(128)
    self.bn4 = nn.BatchNorm2d(64)
    # a tanh activation function
    self.tanh  = nn.Tanh()
  def forward(self, x):
    # pass input into first transpose convolutional layer + batch normalization + ReLU activation
    x = self.latent_to_image(x)
    x = torch.nn.functional.relu(self.bn1(x))
    # pass through each of 3 transpose convolutional layers + batch normalization + ReLU activation
    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))
    # pass through last transpose convolutional layer outputting the generated fake image
    x = self.upsampler5(x)
    x = self.tanh(x)
    return x
```

Figure 2

Training

Figure 3 shows the function implementing the adversarial training logic for the GAN network as per the *run_gan_code* function in Prop. Kak's AdversarialLearning.py (https://engineering.purdue.edu/kak/distDLS/). The function starts by setting the number of channels of the generator's 1x1 noise input vector to 100. It then moves both the discriminator and generator networks to the device and initializes their parameters in an attempt to mitigate training instability. A fixed noise vector is then defined to check the generator's progress during training. Values of 1 and 0 are then set as the real and fake labels respectively. Adam optimizers are then initialized for each of the discriminator and the generator networks, as well as a binary cross entropy loss criterion. Lists for storing accumulated results during training are also initialized.

Next, for each training epoch, lists for storing running losses for each of the discriminator and generator networks are initialized. For the discriminator network, the loss is calculated (and the parameters are updated) in two steps. In the first step, the parameters are updated such that the discriminator's output probability is maximized for the real images. To do this, the gradients of discriminator's learnable parameters are first set to zero, and the real images are moved to the device. A *label* tensor is populated with 1s, which refers to the label of the real images. Next, the real images are passed through the discriminator network, and the discriminator's loss for the real images is calculated using the BCE criterion with labels being 1s. This loss is then backpropagated through the discriminator network. For the second step, the parameters of the discriminator are updated such that the discriminator's output probability is minimized for the fake images. To do this, a noise tensor is first initialized with the previously specified number of channels (100). It is then passed through the generator network which correspondingly generates fake images. Next, the label tensor is repopulated with the value of the fake label (0). Next, the generator's output is detached from the computational graph and then passed through the discriminator. This is done so that the generator's parameters are not updated or affected by the following calculations. The discriminator's loss for the fake images is then calculated using the BCE criterion with labels being 0s this time. This loss is then backpropagated through the discriminator network. The combined loss (from both steps) is then saved for displaying purposes, and the discriminator's optimizer step is updated.

For the generator network, the parameters are updated such that the discriminator's output probability is maximized for the fake images. To do so, the gradients of the generator's learnable parameters are first set to zero and the label tensor is repopulated with the value of the real label (1). The generator's output is then through the discriminator one more time, this time keeping it in the computational graph (since we want to update the parameters of the generator). The generator's loss is calculated using the BCE criterion with labels being 1s. The generator's loss is then stored for displaying purposes. Lastly, the calculated loss is backpropagated through the generator network and the generator's optimizer step is updated.

Every 100 iterations, the number of epochs, iterations, the elapsed time, and the average losses of the discriminator and generator are printed. The running losses are then stored and reinitialized. At the end of training, the discriminator and generator losses are plotted, and real and fake images are displayed alongside each other. Lastly, model parameters are saved for future reference.

```python
# a function that trains the discriminator and the generator networks (adversarial training logic)
# copied from Prop. Kak's AdversarialLearning.py (https://engineering.purdue.edu/kak/distDLS/) with
# some edits and added comments
def run_gan_code(self, dlstudio, adversarial, discriminator, generator, results_dir):
            """
            This function is meant for training a Discriminator-Generator based Adversarial Network.
            The implementation shown uses several programming constructs from the "official" DCGAN
            implementations at the PyTorch website and at GitHub.

            Regarding how to set the parameters of this method, see the following script

                         dcgan_DG1.py

            in the "ExamplesAdversarialLearning" directory of the distribution.
            """

            # prepare directory to hold results
            dir_name_for_results = results_dir
            if os.path.exists(dir_name_for_results):
                files = glob.glob(dir_name_for_results + "/*")
                for file in files:
                    if os.path.isfile(file):
                        os.remove(file)
                    else:
                        files = glob.glob(file + "/*")
                        list(map(lambda x: os.remove(x), files))
            else:
                os.mkdir(dir_name_for_results)

            # the number of channels of the generator's 1x1 noise input vector
            nz = 100
            # move both the discriminator and generator networks to the device
            netD = discriminator.to(self.device)
            netG = generator.to(self.device)

            # initialize the parameters of the discriminator and generator networks in an attempt to
            # mitigate training instability
            netD.apply(self.weights_init)
            netG.apply(self.weights_init)

            # define a fixed noise vector to check the generator's training progress
            fixed_noise = torch.randn(self.dlstudio.batch_size, nz, 1, 1, device=self.device)

            # set real and fake labels to 1 and 0 respectively
            real_label = 1
            fake_label = 0

            # Adam optimizers for each of the discriminator and the generator
            optimizerD = optim.Adam(netD.parameters(), lr=dlstudio.learning_rate, betas=(adversarial.beta1, 0.999))
            optimizerG = optim.Adam(netG.parameters(), lr=dlstudio.learning_rate, betas=(adversarial.beta1, 0.999))

            # binary cross entropy loss criterion
            criterion = nn.BCELoss()

            # lists for storing accumulated results during training
            img_list = []
            D_losses = []
            G_losses = []

            iters = 0
            print("\n\nStarting Training Loop...\n\n")
            start_time = time.perf_counter()
            # for every training epoch
            for epoch in range(dlstudio.epochs):
                # lists for storing running losses for each of the discriminator and generator
                d_losses_per_print_cycle = []
                g_losses_per_print_cycle = []

                # For each batch of images
                for i, data in enumerate(self.train_dataloader, 0):
                    ##  Maximization Part of the Min-Max Objective of Eq. (3):
                    ##
                    ##  As indicated by Eq. (3) in the DCGAN part of the doc section at the beginning of this
                    ##  file, the GAN training boils down to carrying out a min-max optimization. Each iterative
                    ##  step of the max part results in updating the Discriminator parameters and each iterative
                    ##  step of the min part results in the updating of the Generator parameters.  For each
                    ##  batch of the training data, we first do max and then do min.  Since the max operation
                    ##  affects both terms of the criterion shown in the doc section, it has two parts: In the
                    ##  first part we apply the Discriminator to the training images using 1.0 as the target;
                    ##  and, in the second part, we supply to the Discriminator the output of the Generator
                    ##  and use 0 as the target. In what follows, the Discriminator is being applied to
                    ##  the training images:
```

```python
# set gradients of discriminator's learnable parameters to zero
netD.zero_grad()
# move dataset images to device
real_images_in_batch = data[0].to(self.device)
# get number of images in batch
b_size = real_images_in_batch.size(0)
# populate a tensor with values of the real label (1) with that size
label = torch.full((b_size,), real_label, dtype=torch.float, device=self.device)
# pass dataset images through the discriminator network
output = netD(real_images_in_batch).view(-1)
# calculate the discriminator's loss for the real images using the BCE criterion
lossD_for_reals = criterion(output, label)
# perform back propagation for the discriminator network (1st time)
lossD_for_reals.backward()

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

##  That brings us the second part of what it takes to carry out the max operation on the
##  min-max criterion shown in Eq. (3) in the doc section at the beginning of this file.
##  part calls for applying the Discriminator to the images produced by the Generator from noise:

# initialize a noise tensor with the previously specified number of channels
noise = torch.randn(b_size, nz, 1, 1, device=self.device)
# pass the noise tensor through the generator network
fakes = netG(noise)
# populate the label tensor with the value of the fake label (0)
label.fill_(fake_label)

##  The call to fakes.detach() in the next statement returns a copy of the 'fakes' tensor
##  that does not exist in the computational graph. That is, the call shown below first
##  makes a copy of the 'fakes' tensor and then removes it from the computational graph.
##  The original 'fakes' tensor continues to remain in the computational graph.  This ploy
##  ensures that a subsequent call to backward() in the 3rd statement below would only
##  update the netD weights.

# detach the generator output from the computational graph then pass it through the discriminator
output = netD(fakes.detach()).view(-1)
# calculate the discriminator's loss for the fake images using the BCE criterion
lossD_for_fakes = criterion(output, label)
# perform back propagation for the discriminator network (2nd time)
lossD_for_fakes.backward()
# store combined loss for the discriminator for displaying purposes
lossD = lossD_for_reals + lossD_for_fakes
d_losses_per_print_cycle.append(lossD)
# update the discriminator's optimizer step
optimizerD.step()

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

##  Minimization Part of the Min-Max Objective of Eq. (3):
##
##  That brings to the min part of the max-min optimization described in Eq. (3) the doc
##  section at the beginning of this file.  The min part requires that we minimize
##  "1 - D(G(z))" which, since D is constrained to lie in the interval (0,1), requires that
##  we maximize D(G(z)).  We accomplish that by applying the Discriminator to the output
##  of the Generator and use 1 as the target for each image:

# set gradients of generator's learnable parameters to zero
netG.zero_grad()
# populate the label tensor with the value of the real label (1)
label.fill_(real_label)
# pass the generator output through the discriminator one more time, this time keeping it
# in the computational graph
output = netD(fakes).view(-1)
# calculate the generator's loss using the BCE criterion
lossG = criterion(output, label)
# store the generator's loss for displaying purposes
g_losses_per_print_cycle.append(lossG)
# perform back propagation for the generator network
lossG.backward()
# update the generator's optimizer step
optimizerG.step()

# for displaying purposes
if i % 100 == 99:
    current_time = time.perf_counter()
    elapsed_time = current_time - start_time
    mean_D_loss = torch.mean(torch.FloatTensor(d_losses_per_print_cycle))
    mean_G_loss = torch.mean(torch.FloatTensor(g_losses_per_print_cycle))
```

```python
                    # print number of epochs, iterations, the elapsed time, and the average losses of both netwotks
                    print("[epoch=%d/%d   iter=%4d   elapsed_time=%5d secs]     mean_D_loss=%7.4f    mean_G_loss=%7.4f" %
                                  ((epoch+1),dlstudio.epochs,(i+1),elapsed_time,mean_D_loss,mean_G_loss))
                    # reinitialize the lists of running losses
                    d_losses_per_print_cycle = []
                    g_losses_per_print_cycle = []
                # add calculated losses to lists for plotting
                G_losses.append(lossG.item())
                D_losses.append(lossD.item())
                # check generator's progress on the fixed noise vector every 500 iterations
                if (iters % 500 == 0) or ((epoch == dlstudio.epochs-1) and (i == len(self.train_dataloader)-1)):
                    with torch.no_grad():
                        fake = netG(fixed_noise).detach().cpu()
                    img_list.append(torchvision.utils.make_grid(fake, padding=1, pad_value=1, normalize=True))
                iters += 1

        # plot discriminator and generator training losses
        plt.figure(figsize=(10,5))
        plt.title("Generator and Discriminator Loss During Training")
        plt.plot(G_losses,label="G")
        plt.plot(D_losses,label="D")
        plt.xlabel("iterations")
        plt.ylabel("Loss")
        plt.legend()
        plt.savefig(dir_name_for_results + "/gen_and_disc_loss_training.png")
        plt.show()
        #  display a batch-size sample of real images and another of the generator's outputs at the end of training
        real_batch = next(iter(self.train_dataloader))
        real_batch = real_batch[0]
        plt.figure(figsize=(15,15))
        plt.subplot(1,2,1)
        plt.axis("off")
        plt.title("Real Images")
        plt.imshow(np.transpose(torchvision.utils.make_grid(real_batch.to(self.device),
                                               padding=1, pad_value=1, normalize=True).cpu(),(1,2,0)))
        plt.subplot(1,2,2)
        plt.axis("off")
        plt.title("Fake Images")
        plt.imshow(np.transpose(img_list[-1],(1,2,0)))
        plt.savefig(dir_name_for_results + "/real_vs_fake_images.png")
        plt.show()

        # save discriminator and generator parameters
        torch.save(netD.state_dict(), my_dataroot + 'netD.pth')
        torch.save(netG.state_dict(), my_dataroot + 'netG.pth')
```

Figure 3

The code script for training the GAN model is copied from DLStudio's dcgan_DG1.py (https://engineering.purdue.edu/kak/distDLS/) and is shown in Figure 4. An instance of DLStudio is first created specifying the image size (64x64), learning rate, number of epochs, and batch size. The dataroot is also specified such that it is where the celebrity dataset is saved. Next, an instance of AdversarialLearning is created specifying size of the latent vector (1x1 noise image) and beta value for Adam optimizer. Instances of AdversarialLearning.DataModeling, the discriminator network, and the generator network are then created. The number of learnable parameters and number of layers for each of the discriminator and generator networks are displayed, the dataloader is set, and one batch from the dataset is displayed (Figure 5). Lastly, the run_gan_code function is called to run the training logic for the GAN networks.

```python
# training code is copied from DLStudio's dcgan_DG1.py
# (https://engineering.purdue.edu/kak/distDLS/) with added comments

# create an instance of DLStudio specifying image size, learning rate, number of epochs, and batch size
# specify the dataroot where the celebrity dataset is saved
dls = DLStudio(dataroot = my_dataroot,
               image_size = [64,64],
               path_saved_model = "./saved_model",
               learning_rate = 1e-4,
               epochs = 30,
               batch_size = 32,
               use_gpu = True,)
# create an instance of AdversarialLearning specifying size of the latent vector (1x1 noise image) and beta value f
adversarial = AdversarialLearning(
    dlstudio = dls,
    ngpu = 1,
    latent_vector_size = 100,
    beta1 = 0.5,)
# create an instance of AdversarialLearning.DataModeling
dcgan =   AdversarialLearning.DataModeling(dlstudio = dls, adversarial = adversarial)

# create instances of the discriminator and generator networks
discriminator =  disc()
generator =  gen()
# print number of learnable parameters and number of layers for each of the discriminator and generator networks
num_learnable_params_disc = sum(p.numel() for p in discriminator.parameters() if p.requires_grad)
print("\n\nThe number of learnable parameters in the Discriminator: %d\n" % num_learnable_params_disc)
num_learnable_params_gen = sum(p.numel() for p in generator.parameters() if p.requires_grad)
print("\nThe number of learnable parameters in the Generator: %d\n" % num_learnable_params_gen)
num_layers_disc = len(list(discriminator.parameters()))
print("\nThe number of layers in the discriminator: %d\n" % num_layers_disc)
num_layers_gen = len(list(generator.parameters()))
print("\nThe number of layers in the generator: %d\n\n" % num_layers_gen)

# set the dataloader and show one batch from the dataset
dcgan.set_dataloader()
print("\n\nHere is one batch of images from the training dataset:")
dcgan.show_sample_images_from_dataset(dls)

# run the training loop for the GAN networks
run_gan_code(dcgan, dls, adversarial, discriminator=discriminator, generator=generator, results_dir="gan_results")
```

Figure 4

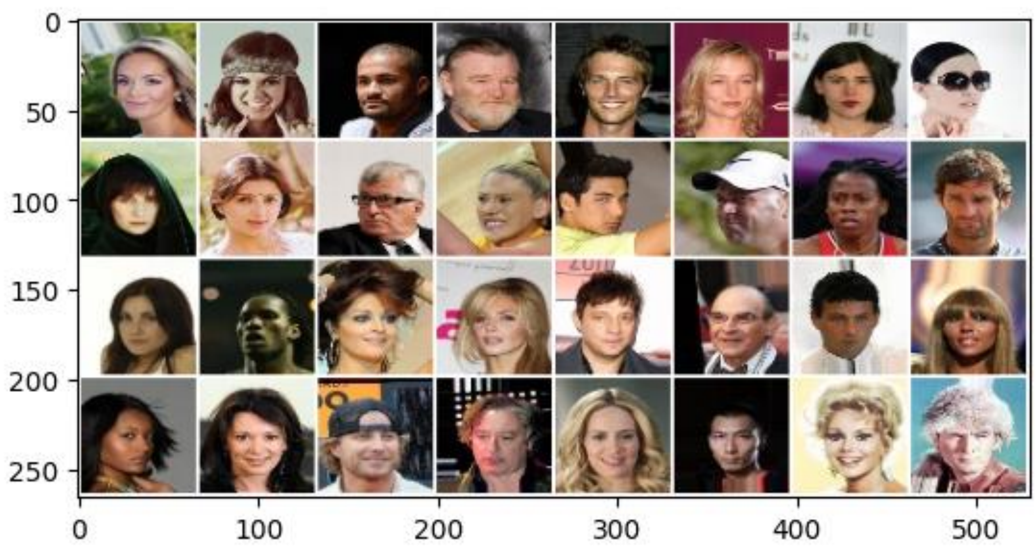

Figure 5 – One batch of real images.

Figure 6 shows the training losses of both the discriminator and the generator every 100 iterations, and Figure 7 shows the corresponding plot. As can be seen, both losses decrease with iterations, indicating that both models are learning. We can see that the discriminator loss is generally lower than that of the generator. However, for both models, we can see spikes where the loss increases and then falls back. This might be attributed to having both models being trained at the same time, with the improvement in each of them meaning that the job is becoming harder for the other one. Figure 8 shows the real and fake images displayed alongside each other (will comment on the generated fake images later in this homework).

```
Starting Training Loop...


[epoch=1/30   iter= 100   elapsed_time=  106 secs]     mean_D_loss= 0.0838    mean_G_loss= 6.9664
[epoch=1/30   iter= 200   elapsed_time=  117 secs]     mean_D_loss= 0.0163    mean_G_loss= 9.2349
[epoch=1/30   iter= 300   elapsed_time=  127 secs]     mean_D_loss= 0.3242    mean_G_loss=14.7539
[epoch=2/30   iter= 100   elapsed_time=  139 secs]     mean_D_loss= 0.4582    mean_G_loss= 6.8935
[epoch=2/30   iter= 200   elapsed_time=  150 secs]     mean_D_loss= 0.5476    mean_G_loss= 5.3684
[epoch=2/30   iter= 300   elapsed_time=  162 secs]     mean_D_loss= 0.5226    mean_G_loss= 4.5083
[epoch=3/30   iter= 100   elapsed_time=  174 secs]     mean_D_loss= 0.5309    mean_G_loss= 5.0721
[epoch=3/30   iter= 200   elapsed_time=  185 secs]     mean_D_loss= 0.5202    mean_G_loss= 4.7056
[epoch=3/30   iter= 300   elapsed_time=  196 secs]     mean_D_loss= 0.5266    mean_G_loss= 4.7250
[epoch=4/30   iter= 100   elapsed_time=  208 secs]     mean_D_loss= 0.5303    mean_G_loss= 4.6911
[epoch=4/30   iter= 200   elapsed_time=  218 secs]     mean_D_loss= 0.4674    mean_G_loss= 4.4783
[epoch=4/30   iter= 300   elapsed_time=  229 secs]     mean_D_loss= 0.5067    mean_G_loss= 4.4919
[epoch=5/30   iter= 100   elapsed_time=  241 secs]     mean_D_loss= 0.5343    mean_G_loss= 4.7875
[epoch=5/30   iter= 200   elapsed_time=  253 secs]     mean_D_loss= 0.5090    mean_G_loss= 4.4662
[epoch=5/30   iter= 300   elapsed_time=  264 secs]     mean_D_loss= 0.4499    mean_G_loss= 4.4037
[epoch=6/30   iter= 100   elapsed_time=  276 secs]     mean_D_loss= 0.4463    mean_G_loss= 4.5950
[epoch=6/30   iter= 200   elapsed_time=  287 secs]     mean_D_loss= 0.5057    mean_G_loss= 4.4660
[epoch=6/30   iter= 300   elapsed_time=  298 secs]     mean_D_loss= 0.3717    mean_G_loss= 4.2916
[epoch=7/30   iter= 100   elapsed_time=  310 secs]     mean_D_loss= 0.3929    mean_G_loss= 4.5235
[epoch=7/30   iter= 200   elapsed_time=  321 secs]     mean_D_loss= 0.3858    mean_G_loss= 4.4694
[epoch=7/30   iter= 300   elapsed_time=  332 secs]     mean_D_loss= 0.3225    mean_G_loss= 4.2470
[epoch=8/30   iter= 100   elapsed_time=  346 secs]     mean_D_loss= 0.3737    mean_G_loss= 4.5276
[epoch=8/30   iter= 200   elapsed_time=  356 secs]     mean_D_loss= 0.3273    mean_G_loss= 4.4095
[epoch=8/30   iter= 300   elapsed_time=  367 secs]     mean_D_loss= 0.4350    mean_G_loss= 4.4106
[epoch=9/30   iter= 100   elapsed_time=  379 secs]     mean_D_loss= 0.3382    mean_G_loss= 4.2311
[epoch=9/30   iter= 200   elapsed_time=  390 secs]     mean_D_loss= 0.3207    mean_G_loss= 4.1880
[epoch=9/30   iter= 300   elapsed_time=  401 secs]     mean_D_loss= 0.2723    mean_G_loss= 4.0774
[epoch=10/30   iter= 100   elapsed_time=  414 secs]     mean_D_loss= 0.3870    mean_G_loss= 4.4338
[epoch=10/30   iter= 200   elapsed_time=  425 secs]     mean_D_loss= 0.2986    mean_G_loss= 4.2992
[epoch=10/30   iter= 300   elapsed_time=  436 secs]     mean_D_loss= 0.2922    mean_G_loss= 4.3903
[epoch=11/30   iter= 100   elapsed_time=  448 secs]     mean_D_loss= 0.4247    mean_G_loss= 4.7235
[epoch=11/30   iter= 200   elapsed_time=  458 secs]     mean_D_loss= 0.3479    mean_G_loss= 4.1239
[epoch=11/30   iter= 300   elapsed_time=  469 secs]     mean_D_loss= 0.2698    mean_G_loss= 4.2596
[epoch=12/30   iter= 100   elapsed_time=  481 secs]     mean_D_loss= 0.4034    mean_G_loss= 4.3758
[epoch=12/30   iter= 200   elapsed_time=  493 secs]     mean_D_loss= 0.3263    mean_G_loss= 4.1643
[epoch=12/30   iter= 300   elapsed_time=  504 secs]     mean_D_loss= 0.4370    mean_G_loss= 4.4322
[epoch=13/30   iter= 100   elapsed_time=  516 secs]     mean_D_loss= 0.3827    mean_G_loss= 4.3467
[epoch=13/30   iter= 200   elapsed_time=  528 secs]     mean_D_loss= 0.2962    mean_G_loss= 4.1623
[epoch=13/30   iter= 300   elapsed_time=  538 secs]     mean_D_loss= 0.3746    mean_G_loss= 4.2631
[epoch=14/30   iter= 100   elapsed_time=  550 secs]     mean_D_loss= 0.3330    mean_G_loss= 4.1321
[epoch=14/30   iter= 200   elapsed_time=  561 secs]     mean_D_loss= 0.4608    mean_G_loss= 4.1477
[epoch=14/30   iter= 300   elapsed_time=  572 secs]     mean_D_loss= 0.2969    mean_G_loss= 4.1526
[epoch=15/30   iter= 100   elapsed_time=  585 secs]     mean_D_loss= 0.3679    mean_G_loss= 4.0017
[epoch=15/30   iter= 200   elapsed_time=  596 secs]     mean_D_loss= 0.3569    mean_G_loss= 4.1798
[epoch=15/30   iter= 300   elapsed_time=  607 secs]     mean_D_loss= 0.3941    mean_G_loss= 3.9580
```

Figure 6

```
[epoch=16/30   iter= 100   elapsed_time=  619 secs]     mean_D_loss= 0.3048     mean_G_loss= 3.8824
[epoch=16/30   iter= 200   elapsed_time=  630 secs]     mean_D_loss= 0.5236     mean_G_loss= 3.9540
[epoch=16/30   iter= 300   elapsed_time=  642 secs]     mean_D_loss= 0.3024     mean_G_loss= 3.9344
[epoch=17/30   iter= 100   elapsed_time=  654 secs]     mean_D_loss= 0.4402     mean_G_loss= 4.0197
[epoch=17/30   iter= 200   elapsed_time=  665 secs]     mean_D_loss= 0.4386     mean_G_loss= 3.9877
[epoch=17/30   iter= 300   elapsed_time=  676 secs]     mean_D_loss= 0.3191     mean_G_loss= 3.9175
[epoch=18/30   iter= 100   elapsed_time=  689 secs]     mean_D_loss= 0.3854     mean_G_loss= 4.1342
[epoch=18/30   iter= 200   elapsed_time=  699 secs]     mean_D_loss= 0.4600     mean_G_loss= 3.9106
[epoch=18/30   iter= 300   elapsed_time=  711 secs]     mean_D_loss= 0.3948     mean_G_loss= 3.9229
[epoch=19/30   iter= 100   elapsed_time=  724 secs]     mean_D_loss= 0.2472     mean_G_loss= 3.7651
[epoch=19/30   iter= 200   elapsed_time=  735 secs]     mean_D_loss= 0.4780     mean_G_loss= 4.2479
[epoch=19/30   iter= 300   elapsed_time=  746 secs]     mean_D_loss= 0.3616     mean_G_loss= 4.0488
[epoch=20/30   iter= 100   elapsed_time=  759 secs]     mean_D_loss= 0.3367     mean_G_loss= 3.9201
[epoch=20/30   iter= 200   elapsed_time=  770 secs]     mean_D_loss= 0.4964     mean_G_loss= 4.0999
[epoch=20/30   iter= 300   elapsed_time=  780 secs]     mean_D_loss= 0.2444     mean_G_loss= 3.8674
[epoch=21/30   iter= 100   elapsed_time=  793 secs]     mean_D_loss= 0.4423     mean_G_loss= 4.1697
[epoch=21/30   iter= 200   elapsed_time=  804 secs]     mean_D_loss= 0.5070     mean_G_loss= 4.0178
[epoch=21/30   iter= 300   elapsed_time=  815 secs]     mean_D_loss= 0.3308     mean_G_loss= 3.8530
[epoch=22/30   iter= 100   elapsed_time=  827 secs]     mean_D_loss= 0.2422     mean_G_loss= 3.8152
[epoch=22/30   iter= 200   elapsed_time=  838 secs]     mean_D_loss= 0.3060     mean_G_loss= 4.1834
[epoch=22/30   iter= 300   elapsed_time=  849 secs]     mean_D_loss= 0.2733     mean_G_loss= 4.1571
[epoch=23/30   iter= 100   elapsed_time=  862 secs]     mean_D_loss= 0.2724     mean_G_loss= 4.2740
[epoch=23/30   iter= 200   elapsed_time=  873 secs]     mean_D_loss= 0.5848     mean_G_loss= 4.2816
[epoch=23/30   iter= 300   elapsed_time=  885 secs]     mean_D_loss= 0.2542     mean_G_loss= 3.9334
[epoch=24/30   iter= 100   elapsed_time=  897 secs]     mean_D_loss= 0.4654     mean_G_loss= 4.2763
[epoch=24/30   iter= 200   elapsed_time=  909 secs]     mean_D_loss= 0.2397     mean_G_loss= 4.0666
[epoch=24/30   iter= 300   elapsed_time=  920 secs]     mean_D_loss= 0.2962     mean_G_loss= 4.4284
[epoch=25/30   iter= 100   elapsed_time=  933 secs]     mean_D_loss= 0.5158     mean_G_loss= 4.3029
[epoch=25/30   iter= 200   elapsed_time=  943 secs]     mean_D_loss= 0.1948     mean_G_loss= 4.0082
[epoch=25/30   iter= 300   elapsed_time=  954 secs]     mean_D_loss= 0.3487     mean_G_loss= 4.2701
[epoch=26/30   iter= 100   elapsed_time=  966 secs]     mean_D_loss= 0.1818     mean_G_loss= 4.2525
[epoch=26/30   iter= 200   elapsed_time=  977 secs]     mean_D_loss= 0.3197     mean_G_loss= 4.5368
[epoch=26/30   iter= 300   elapsed_time=  988 secs]     mean_D_loss= 0.3471     mean_G_loss= 4.4376
[epoch=27/30   iter= 100   elapsed_time= 1000 secs]     mean_D_loss= 0.4319     mean_G_loss= 4.2111
[epoch=27/30   iter= 200   elapsed_time= 1011 secs]     mean_D_loss= 0.1357     mean_G_loss= 4.3046
[epoch=27/30   iter= 300   elapsed_time= 1022 secs]     mean_D_loss= 0.4335     mean_G_loss= 4.6527
[epoch=28/30   iter= 100   elapsed_time= 1034 secs]     mean_D_loss= 0.4362     mean_G_loss= 4.8044
[epoch=28/30   iter= 200   elapsed_time= 1045 secs]     mean_D_loss= 0.1712     mean_G_loss= 4.2544
[epoch=28/30   iter= 300   elapsed_time= 1057 secs]     mean_D_loss= 0.4288     mean_G_loss= 4.5637
[epoch=29/30   iter= 100   elapsed_time= 1070 secs]     mean_D_loss= 0.1673     mean_G_loss= 4.2213
[epoch=29/30   iter= 200   elapsed_time= 1081 secs]     mean_D_loss= 0.4271     mean_G_loss= 4.6863
[epoch=29/30   iter= 300   elapsed_time= 1092 secs]     mean_D_loss= 0.2066     mean_G_loss= 4.2965
[epoch=30/30   iter= 100   elapsed_time= 1104 secs]     mean_D_loss= 0.1700     mean_G_loss= 4.6917
[epoch=30/30   iter= 200   elapsed_time= 1115 secs]     mean_D_loss= 0.1828     mean_G_loss= 4.7702
[epoch=30/30   iter= 300   elapsed_time= 1126 secs]     mean_D_loss= 0.3553     mean_G_loss= 5.4143
```

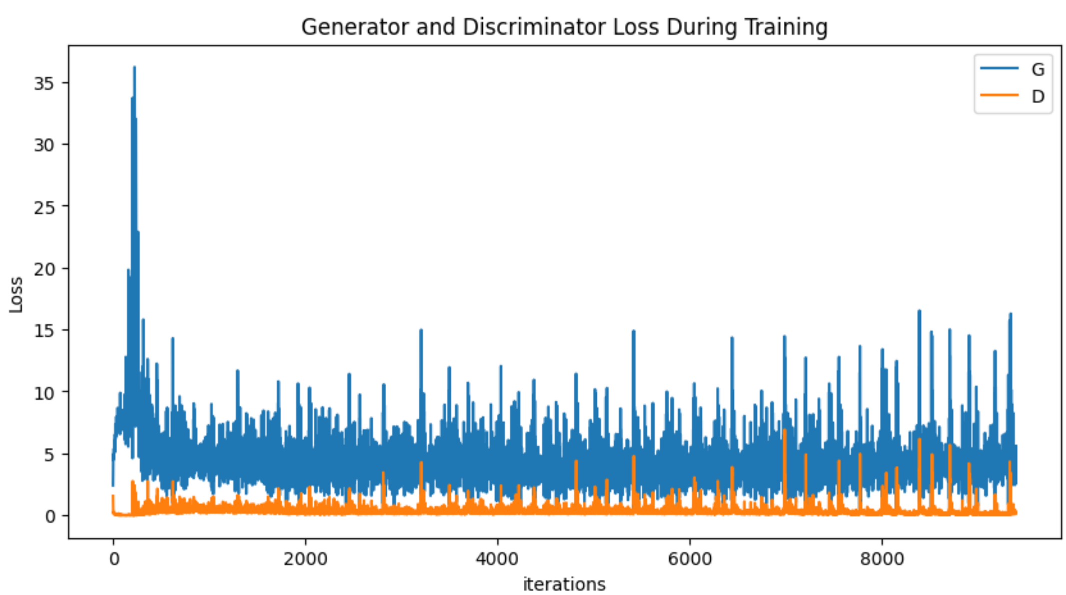

Figure 7

Figure 8 – Real images and GAN's fake images

Generating Fake Images

Figure 9 shows the code used to generate fake images using the trained generator network. A noise tensor with 2048 noise vectors (each with 100 channels) is first randomly initialized. After loading the trained generator, the noise tensor is then passed to it as input and 2048 corresponding fake images are generated. We then loop on those images and save each of them to our directory. (Note that the choice of 2048 as the number of fake images to be generated was chosen based on an error experienced when later calculating the FID value using a smaller number of images).

```
# specify number of fake images to generate
num_fakes = 2048
# create a corresponding tensor of noise vectirs, each with 100 channels
noise_vectors = torch.randn(num_fakes, 100, 1, 1, device=dcgan.device)

# load saved parameters of trained generator
trained_gen =  gen()
trained_gen.load_state_dict(torch.load(my_dataroot + 'netG.pth'))
trained_gen = trained_gen.to(dcgan.device)

# pass the noise tensor to the trained generator to generate the fakes
fakes_2048 = trained_gen(noise_vectors)

# change every generated image to a PIL image and save it
for i in range(num_fakes):
  fake_image = fakes_2048[i]
  fake_image = torchvision.utils.make_grid(fake_image, padding=1, pad_value=1, normalize=True)
  fake_image = tvtF.to_pil_image(fake_image)
  fake_image.save(my_dataroot + "gan_fake_dataset/" + str(i) + ".jpg")
```

Figure 9

Calculate FID Values

Figure 10 shows the code used to calculate the FID value for the GAN's fake images. First, the paths of the real and the fake images are each stored in a list using *os.listdir()*. Next, the code provided in HW8 instructions is used to calculate the FID value: an object of the Inception model is instantiated, the mean and standard deviation of the distributions from the real and fake datasets are all calculated, and lastly the FID values are calculated using *calculate_frechet_distance.* As can be seen in Figure 10, the FID value was calculated to be 77.98.

```
# get paths of images in the real dataset
real_paths = os.listdir(my_datatoot + "real_dataset/")
real_paths = [my_datatoot + "real_dataset/" + i for i in real_paths]
```

```
# get paths of images in the fake dataset
fake_paths = os.listdir(my_datatoot + "gan_fake_dataset/")
fake_paths = [my_datatoot + "gan_fake_dataset/" + i for i in fake_paths]
```

```
# code copied from HW8 instructions
dims = 2048
block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
# instantiate an inception model
model = InceptionV3([block_idx]).to("cuda:0")
# calculate the mean and standard deviation of the distribution from the real dataset
m1, s1 = calculate_activation_statistics(real_paths[:2048], model, device = "cuda:0")
# calculate the mean and standard deviation of the distribution from the fake dataset
m2, s2 = calculate_activation_statistics (fake_paths, model, device = "cuda:0")
# calculate and pring the FID value
fid_value = calculate_frechet_distance(m1, s1, m2, s2)
print(f'FID: {fid_value:.2f}')
```

```
100%|██████████| 41/41 [02:07<00:00,  3.11s/it]
100%|██████████| 41/41 [00:08<00:00,  5.09it/s]
FID: 77.98
```

Figure 10

Diffusion:

Generate Fake Images

As instructed in HW8, we use the network weights provided with the homework instructions to generate fake images using diffusion. To do that, we run *GenerateNewImageSamples.py* from DLStudio (https://engineering.purdue.edu/kak/distDLS/) and change the following:

- The *num_diffusion_steps* is changed to 200 instead of originally 1000. This change was made because the diffusion network was taking too long to generate the fake images. Therefore, for computational constraints, the number of diffusion steps was reduced to 200.
- The *num_samples* is changed to 2048. (Note that the choice of 2048 as the number of fake images to be generated was chosen based on an error experienced when later calculating the FID value using a smaller number of images).
- The model path was changed as shown in Figure 11 so that the provided model weights are loaded.

```
model_path = my_dataroot + "diffusion.pt"
network.load_state_dict(torch.load(model_path))
```

Figure 11

Visualize Fake Images

After generating the fake images, we run the *VisualizeSamples.py* from DLStudio (https://engineering.purdue.edu/kak/distDLS/) and change the following:

- The *npz_archive* is changed to be *my_dataroot + "samples_2048x64x64x3.npz"* to reflect the newly generated fake images.
- The *visualization_dir* is changed to *my_dataroot + "diffusion_fake_images"* to indicate the directory where we want the generated images to be saved.
- When calling *img.save()*, we give the location where we want the image to be saved in our *my_dataroot + "diffusion_fake_images"* directory.

Calculate FID Values

Figure 12 shows the code used to calculate the FID value for the Diffusion's fake images. First, the paths of the real and the fake images are each stored in a list using *os.listdir()*. Next, the code provided in HW8 instructions is used to calculate the FID value: an object of the Inception model is instantiated, the mean and standard deviation of the distributions from the real and fake datasets are all calculated, and lastly the FID values are calculated using *calculate_frechet_distance.* As can be seen in Figure 13, the FID value was calculated to be 153.39.

```
# get paths of images in the real dataset
real_paths = os.listdir(my_datatoot + "real_dataset/")
real_paths = [my_datatoot + "real_dataset/" + i for i in real_paths]

# get paths of images in the fake dataset
fake_paths = os.listdir(my_datatoot + "diffusion_fake_dataset/")
fake_paths = [my_datatoot + "diffusion_fake_dataset/" + i for i in fake_paths]

# code copied from HW8 instructions
dims = 2048
block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
# instantiate an inception model
model = InceptionV3([block_idx]).to("cpu")
# calculate the mean and standard deviation of the distribution from the real dataset
m1, s1 = calculate_activation_statistics(real_paths[:2048], model, device = "cpu")
# calculate the mean and standard deviation of the distribution from the fake dataset
m2, s2 = calculate_activation_statistics (fake_paths, model, device = "cpu")
# calculate and pring the FID value
fid_value = calculate_frechet_distance(m1, s1, m2, s2)
print(f'FID: {fid_value:.2f}')
```

Figure 12

```
100%|██████████| 41/41 [14:43<00:00, 21.55s/it]
100%|██████████| 41/41 [14:45<00:00, 21.60s/it]
FID: 153.39
```

Figure 13

Evaluating Models

Qualitative Evaluation

Figure 14 and Figure 15 show the 4x4 images generated by the GAN and Diffusion networks respectively. We can see that both models were actually able to generate images from the input noise vectors. As we can qualitatively spot from Figure 14, the generator from the GAN network was able to generate semi-decent face images with similar color distributions to those in the real dataset. However, the faces still look atypical and non-normal. As seen from Figure 15, the Diffusion network was able to generate more normal-looking faces with much less atypicality. However, we can see that the color distribution in the images is different from that in the real dataset. Another point that can be noticed is that images generated by the Diffusion model are generally less diverse than those generated by the GAN's generator. In other words, all the Diffusion's fake images seem to be more or less similar.

Figure 14

Figure 15

Quantitative Evaluation

The FID values calculated for the GAN and the Diffusion networks are 77.98 and 153.39 respectively. Generally speaking, a lower FID value indicates that the distributions are more similar; that the generated images are closer to the real ones when it comes to the visual quality and diversity (https://www.linkedin.com/pulse/fr%C3%A9chet-inception-distance-yeshwanth-n/). This means that the fake images generated by the GAN's generator was closer in visual quality and diversity to the real images that those generated by the Diffusion network. This interpretation partially matches the qualitative observation because we can confirm that the generator's images do better match the color distribution in the real dataset as well as its diversity. On the other hand, perhaps it qualitatively seems that the quality of the Diffusion's fake images is generally better than those of the generator's (it is more apparent qualitatively than reflected by the FID values).

General Discussion

Overall, the Diffusion network seems to generate images with more normally looking images. It seems to be closer to human faces in general, but not necessarily close to those in the celebrity dataset in particular. In general, the output of the Diffusion network is expected to be enhanced by increasing the number of diffusion steps (which was greatly reduced here for computational cost). The GAN's generator seems to capture the overall essence, color distribution, and diversity of the dataset of real images. However, perhaps it could use some more epochs of training to enhance the quality of the generated images and the typicality of the generated faces.

Full Source Code:

```
# -*- coding: utf-8 -*-
"""hw8_NadineAmin.ipynb

# BME646 and ECE 60146 - Homework 8 - Nadine Amin

## Libraries
"""

# import libraries, DLStudio, and AdversarialLearning
import random
import numpy as np
import os
import sys
import copy
import torch
import torch.nn as nn
import torch.optim as optim
import time
import pickle
from pytorch_fid.fid_score import calculate_activation_statistics,
calculate_frechet_distance
```

```python
from pytorch_fid.inception import InceptionV3
from DLStudio import *
from AdversarialLearning import *

"""# GAN

## Networks

### Discriminator
"""

# a class for the discriminator network
# copied from the DiscriminatorDG1 class in Prof. Kak's AdversarialLearning.py
# (https://engineering.purdue.edu/kak/distDLS/) with added comments
class disc(nn.Module):
  """
  This is an implementation of the DCGAN Discriminator. I refer to the DCGAN
network topology as
  the 4-2-1 network.  Each layer of the Discriminator network carries out a
strided
  convolution with a 4x4 kernel, a 2x2 stride and a 1x1 padding for all but the
final
  layer. The output of the final convolutional layer is pushed through a sigmoid
to yield
  a scalar value as the final output for each image in a batch.

  Class Path:  AdversarialLearning  ->   DataModeling  ->  DiscriminatorDG1
  """
  def __init__(self):
    super(disc, self).__init__()
    # 4 convolutional layers (different numbers of channels) with a kernel size
of 4, stride of 2, and padding of 1
    self.conv_in = nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1)
    self.conv_in2 = nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1)
    self.conv_in3 = nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1)
    self.conv_in4 = nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1)
    # 1 convolutional layer with a single node as the discriminator's output for
each image
    self.conv_in5 = nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=0)
    # 3 batch normalization layers with different numbers of channels
    self.bn1  = nn.BatchNorm2d(128)
    self.bn2  = nn.BatchNorm2d(256)
    self.bn3  = nn.BatchNorm2d(512)
    # a Sigmoid activation function to output a probability
    self.sig = nn.Sigmoid()
```

```
  def forward(self, x):
    # pass input into 1st convolutional layer + leaky ReLU activation (to avoid
dying ReLU problem)
    x = torch.nn.functional.leaky_relu(self.conv_in(x), negative_slope=0.2,
inplace=True)
    # pass through 2nd convolutional layer + 1st batch normalization + leaky ReLU
    x = self.bn1(self.conv_in2(x))
    x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
    # pass through 3rd convolutional layer + 2nd batch normalization + leaky ReLU
    x = self.bn2(self.conv_in3(x))
    x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
    # pass through 4th convolutional layer + 3rd batch normalization + leaky ReLU
    x = self.bn3(self.conv_in4(x))
    x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
    # pass through last convolutional layer + Sigmoid activation function to
output probability
    x = self.conv_in5(x)
    x = self.sig(x)
    return x

"""### Generator"""

# a class for the generator network
# copied from the GeneratorDG1 class in Prof. Kak's AdversarialLearning.py
# (https://engineering.purdue.edu/kak/distDLS/) with added comments
class gen(nn.Module):
  """
  This is an implementation of the DCGAN Generator. As was the case with the
Discriminator network,
  you again see the 4-2-1 topology here.  A Generator's job is to transform a
random noise
  vector into an image that is supposed to look like it came from the training
dataset. (We refer
  to the images constructed from noise vectors in this manner as fakes.)  As you
will see later
  in the "run_gan_code()" method, the starting noise vector is a 1x1 image with
100 channels.  In
  order to output 64x64 output images, the network shown below use the Transpose
Convolution
  operator nn.ConvTranspose2d with a stride of 2.  If (H_in, W_in) are the height
and the width
  of the image at the input to a nn.ConvTranspose2d layer and (H_out, W_out) the
same at the
  output, the size pairs are related by
```

```
                H_out   =   (H_in - 1) * s   +   k   -   2 * p
                W_out   =   (W_in - 1) * s   +   k   -   2 * p
  were s is the stride and k the size of the kernel.  (I am assuming square
strides, kernels, and
  padding). Therefore, each nn.ConvTranspose2d layer shown below doubles the size
of the input.
  Class Path:  AdversarialLearning  ->   DataModeling  ->  GeneratorDG1
  """
  def __init__(self):
    super(gen, self).__init__()
    # a transpose convolutional layer taking in a 1x1 input of 100 channels
    self.latent_to_image = nn.ConvTranspose2d(100, 512, kernel_size=4, stride=1,
padding=0, bias=False)
    # 3 transpose convolutional layers (different numbers of channels) with a
kernel size of 4, stride of 2, and padding of 1
    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)
    # 1 transpose convolutional layer with 3 output channels resembling the
generator's fake image
    self.upsampler5 = nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2,
padding=1, bias=False)
    # 4 batch normalization layers with different numbers of channels
    self.bn1 = nn.BatchNorm2d(512)
    self.bn2 = nn.BatchNorm2d(256)
    self.bn3 = nn.BatchNorm2d(128)
    self.bn4 = nn.BatchNorm2d(64)
    # a tanh activation function
    self.tanh  = nn.Tanh()

  def forward(self, x):
    # pass input into first transpose convolutional layer + batch normalization +
ReLU activation
    x = self.latent_to_image(x)
    x = torch.nn.functional.relu(self.bn1(x))
    # pass through each of 3 transpose convolutional layers + batch normalization
+ ReLU activation
    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))
    # pass through last transpose convolutional layer outputting the generated
fake image
    x = self.upsampler5(x)
    x = self.tanh(x)
    return x

"""## Training"""

# a function that trains the discriminator and the generator networks
(adversarial training logic)
# copied from Prop. Kak's AdversarialLearning.py
(https://engineering.purdue.edu/kak/distDLS/) with
# some edits and added comments
def run_gan_code(self, dlstudio, adversarial, discriminator, generator,
results_dir):
            """
            This function is meant for training a Discriminator-Generator based
Adversarial Network.
            The implementation shown uses several programming constructs from the
"official" DCGAN
            implementations at the PyTorch website and at GitHub.

            Regarding how to set the parameters of this method, see the following
script

                         dcgan_DG1.py

            in the "ExamplesAdversarialLearning" directory of the distribution.
            """

            # prepare directory to hold results
            dir_name_for_results = results_dir
            if os.path.exists(dir_name_for_results):
                files = glob.glob(dir_name_for_results + "/*")
                for file in files:
                    if os.path.isfile(file):
                        os.remove(file)
                    else:
                        files = glob.glob(file + "/*")
                        list(map(lambda x: os.remove(x), files))
            else:
                os.mkdir(dir_name_for_results)

            # the number of channels of the generator's 1x1 noise input vector
```

```
            nz = 100
            # move both the discriminator and generator networks to the device
            netD = discriminator.to(self.device)
            netG = generator.to(self.device)

            # initialize the parameters of the discriminator and generator
networks in an attempt to
            # mitigate training instability
            netD.apply(self.weights_init)
            netG.apply(self.weights_init)

            # define a fixed noise vector to check the generator's training
progress
            fixed_noise = torch.randn(self.dlstudio.batch_size, nz, 1, 1,
device=self.device)

            # set real and fake labels to 1 and 0 respectively
            real_label = 1
            fake_label = 0

            # Adam optimizers for each of the discriminator and the generator
            optimizerD = optim.Adam(netD.parameters(), lr=dlstudio.learning_rate,
betas=(adversarial.beta1, 0.999))
            optimizerG = optim.Adam(netG.parameters(), lr=dlstudio.learning_rate,
betas=(adversarial.beta1, 0.999))

            # binary cross entropy loss criterion
            criterion = nn.BCELoss()

            # lists for storing accumulated results during training
            img_list = []
            D_losses = []
            G_losses = []

            iters = 0
            print("\n\nStarting Training Loop...\n\n")
            start_time = time.perf_counter()
            # for every training epoch
            for epoch in range(dlstudio.epochs):
                # lists for storing running losses for each of the discriminator
and generator
                d_losses_per_print_cycle = []
                g_losses_per_print_cycle = []

                # For each batch of images
```

```
                for i, data in enumerate(self.train_dataloader, 0):
                    ##  Maximization Part of the Min-Max Objective of Eq. (3):
                    ##
                    ##  As indicated by Eq. (3) in the DCGAN part of the doc
section at the beginning of this
                    ##  file, the GAN training boils down to carrying out a min-
max optimization. Each iterative
                    ##  step of the max part results in updating the
Discriminator parameters and each iterative
                    ##  step of the min part results in the updating of the
Generator parameters.  For each
                    ##  batch of the training data, we first do max and then do
min.  Since the max operation
                    ##  affects both terms of the criterion shown in the doc
section, it has two parts: In the
                    ##  first part we apply the Discriminator to the training
images using 1.0 as the target;
                    ##  and, in the second part, we supply to the Discriminator
the output of the Generator
                    ##  and use 0 as the target. In what follows, the
Discriminator is being applied to
                    ##  the training images:

                    # set gradients of discriminator's learnable parameters to
zero
                    netD.zero_grad()
                    # move dataset images to device
                    real_images_in_batch = data[0].to(self.device)
                    # get number of images in batch
                    b_size = real_images_in_batch.size(0)
                    # populate a tensor with values of the real label (1) with
that size
                    label = torch.full((b_size,), real_label, dtype=torch.float,
device=self.device)
                    # pass dataset images through the discriminator network
                    output = netD(real_images_in_batch).view(-1)
                    # calculate the discriminator's loss for the real images
using the BCE criterion
                    lossD_for_reals = criterion(output, label)
                    # perform back propagation for the discriminator network (1st
time)
                    lossD_for_reals.backward()

                    #############################################################
###################################
```

```
                    ##  That brings us the second part of what it takes to carry
out the max operation on the
                    ##  min-max criterion shown in Eq. (3) in the doc section at
the beginning of this file.
                    ##  part calls for applying the Discriminator to the images
produced by the Generator from noise:

                    # initialize a noise tensor with the previously specified
number of channels
                    noise = torch.randn(b_size, nz, 1, 1, device=self.device)
                    # pass the noise tensor through the generator network
                    fakes = netG(noise)
                    # populate the label tensor with the value of the fake label
(0)
                    label.fill_(fake_label)

                    ##  The call to fakes.detach() in the next statement returns
a copy of the 'fakes' tensor
                    ##  that does not exist in the computational graph. That is,
the call shown below first
                    ##  makes a copy of the 'fakes' tensor and then removes it
from the computational graph.
                    ##  The original 'fakes' tensor continues to remain in the
computational graph.  This ploy
                    ##  ensures that a subsequent call to backward() in the 3rd
statement below would only
                    ##  update the netD weights.

                    # detach the generator output from the computational graph
then pass it through the discriminator
                    output = netD(fakes.detach()).view(-1)
                    # calculate the discriminator's loss for the fake images
using the BCE criterion
                    lossD_for_fakes = criterion(output, label)

                    # perform back propagation for the discriminator network (2nd
time)
                    lossD_for_fakes.backward()
                    # store combined loss for the discriminator for displaying
purposes
                    lossD = lossD_for_reals + lossD_for_fakes
                    d_losses_per_print_cycle.append(lossD)
                    # update the discriminator's optimizer step
                    optimizerD.step()
```

```
                    ##############################################################
###################################

                    ##  Minimization Part of the Min-Max Objective of Eq. (3):
                    ##
                    ##  That brings to the min part of the max-min optimization
described in Eq. (3) the doc
                    ##  section at the beginning of this file.  The min part
requires that we minimize
                    ##  "1 - D(G(z))" which, since D is constrained to lie in the
interval (0,1), requires that
                    ##  we maximize D(G(z)).  We accomplish that by applying the
Discriminator to the output
                    ##  of the Generator and use 1 as the target for each image:

                    # set gradients of generator's learnable parameters to zero
                    netG.zero_grad()
                    # populate the label tensor with the value of the real label
(1)
                    label.fill_(real_label)
                    # pass the generator output through the discriminator one
more time, this time keeping it
                    # in the computational graph
                    output = netD(fakes).view(-1)
                    # calculate the generator's loss using the BCE criterion
                    lossG = criterion(output, label)
                    # store the generator's loss for displaying purposes
                    g_losses_per_print_cycle.append(lossG)
                    # perform back propagation for the generator network
                    lossG.backward()
                    # update the generator's optimizer step
                    optimizerG.step()

                    # for displaying purposes
                    if i % 100 == 99:
                        current_time = time.perf_counter()
                        elapsed_time = current_time - start_time
                        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 number of epochs, iterations, the elapsed time,
and the average losses of both netwotks
```

```
                        print("[epoch=%d/%d   iter=%4d   elapsed_time=%5d
secs]     mean_D_loss=%7.4f    mean_G_loss=%7.4f" %
                                      ((epoch+1),dlstudio.epochs,(i+1),elapsed_ti
me,mean_D_loss,mean_G_loss))
                        # reinitialize the lists of running losses
                        d_losses_per_print_cycle = []
                        g_losses_per_print_cycle = []
                    # add calculated losses to lists for plotting
                    G_losses.append(lossG.item())
                    D_losses.append(lossD.item())
                    # check generator's progress on the fixed noise vector every
500 iterations
                    if (iters % 500 == 0) or ((epoch == dlstudio.epochs-1) and (i
== len(self.train_dataloader)-1)):
                        with torch.no_grad():
                            fake = netG(fixed_noise).detach().cpu()
                        img_list.append(torchvision.utils.make_grid(fake,
padding=1, pad_value=1, normalize=True))
                    iters += 1

            # plot discriminator and generator training losses
            plt.figure(figsize=(10,5))
            plt.title("Generator and Discriminator Loss During Training")
            plt.plot(G_losses,label="G")
            plt.plot(D_losses,label="D")
            plt.xlabel("iterations")
            plt.ylabel("Loss")
            plt.legend()
            plt.savefig(dir_name_for_results + "/gen_and_disc_loss_training.png")
            plt.show()
            #  display a batch-size sample of real images and another of the
generator's outputs at the end of training
            real_batch = next(iter(self.train_dataloader))
            real_batch = real_batch[0]
            plt.figure(figsize=(15,15))
            plt.subplot(1,2,1)
            plt.axis("off")
            plt.title("Real Images")
            plt.imshow(np.transpose(torchvision.utils.make_grid(real_batch.to(sel
f.device),
                                                   padding=1, pad_value=1,
normalize=True).cpu(),(1,2,0)))
            plt.subplot(1,2,2)
            plt.axis("off")
            plt.title("Fake Images")
```

```
            plt.imshow(np.transpose(img_list[-1],(1,2,0)))
            plt.savefig(dir_name_for_results + "/real_vs_fake_images.png")
            plt.show()

            # save discriminator and generator parameters
            torch.save(netD.state_dict(), my_dataroot + 'netD.pth')
            torch.save(netG.state_dict(), my_dataroot + 'netG.pth')

# training code is copied from DLStudio's dcgan_DG1.py
# (https://engineering.purdue.edu/kak/distDLS/) with added comments

# create an instance of DLStudio specifying image size, learning rate, number of
epochs, and batch size
# specify the dataroot where the celebrity dataset is saved
dls = DLStudio(dataroot = my_dataroot,
               image_size = [64,64],
               path_saved_model = "./saved_model",
               learning_rate = 1e-4,
               epochs = 30,
               batch_size = 32,
               use_gpu = True,)
# create an instance of AdversarialLearning specifying size of the latent vector
(1x1 noise image) and beta value for Adam optimizer
adversarial = AdversarialLearning(
    dlstudio = dls,
    ngpu = 1,
    latent_vector_size = 100,
    beta1 = 0.5,)
# create an instance of AdversarialLearning.DataModeling
dcgan =   AdversarialLearning.DataModeling(dlstudio = dls, adversarial =
adversarial)

# create instances of the discriminator and generator networks
discriminator =  disc()
generator =  gen()
# print number of learnable parameters and number of layers for each of the
discriminator and generator networks
num_learnable_params_disc = sum(p.numel() for p in discriminator.parameters() if
p.requires_grad)
print("\n\nThe number of learnable parameters in the Discriminator: %d\n" %
num_learnable_params_disc)
num_learnable_params_gen = sum(p.numel() for p in generator.parameters() if
p.requires_grad)
print("\nThe number of learnable parameters in the Generator: %d\n" %
num_learnable_params_gen)
```

```
num_layers_disc = len(list(discriminator.parameters()))
print("\nThe number of layers in the discriminator: %d\n" % num_layers_disc)
num_layers_gen = len(list(generator.parameters()))
print("\nThe number of layers in the generator: %d\n\n" % num_layers_gen)

# set the dataloader and show one batch from the dataset
dcgan.set_dataloader()
print("\n\nHere is one batch of images from the training dataset:")
dcgan.show_sample_images_from_dataset(dls)

# run the training loop for the GAN networks
run_gan_code(dcgan, dls, adversarial, discriminator=discriminator,
generator=generator, results_dir="gan_results")

"""## Generate Fake Images"""

# specify number of fake images to generate
num_fakes = 2048
# create a corresponding tensor of noise vectirs, each with 100 channels
noise_vectors = torch.randn(num_fakes, 100, 1, 1, device=dcgan.device)

# load saved parameters of trained generator
trained_gen =  gen()
trained_gen.load_state_dict(torch.load(my_dataroot + 'netG.pth'))
trained_gen = trained_gen.to(dcgan.device)

# pass the noise tensor to the trained generator to generate the fakes
fakes_2048 = trained_gen(noise_vectors)

# change every generated image to a PIL image and save it
for i in range(num_fakes):
  fake_image = fakes_2048[i]
  fake_image = torchvision.utils.make_grid(fake_image, padding=1, pad_value=1,
normalize=True)
  fake_image = tvtF.to_pil_image(fake_image)
  fake_image.save(my_dataroot + "gan_fake_dataset/" + str(i) + ".jpg")

"""## Calculate FID Values"""

# get paths of images in the real dataset
real_paths = os.listdir(my_datatoot + "real_dataset/")
real_paths = [my_datatoot + "real_dataset/" + i for i in real_paths]

# get paths of images in the fake dataset
fake_paths = os.listdir(my_datatoot + "gan_fake_dataset/")
```

```
fake_paths = [my_datatoot + "gan_fake_dataset/" + i for i in fake_paths]

# code copied from HW8 instructions
dims = 2048
block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
# instantiate an inception model
model = InceptionV3([block_idx]).to("cuda:0")
# calculate the mean and standard deviation of the distribution from the real
dataset
m1, s1 = calculate_activation_statistics(real_paths[:2048], model, device =
"cuda:0")
# calculate the mean and standard deviation of the distribution from the fake
dataset
m2, s2 = calculate_activation_statistics (fake_paths, model, device = "cuda:0")
# calculate and pring the FID value
fid_value = calculate_frechet_distance(m1, s1, m2, s2)
print(f'FID: {fid_value:.2f}')

"""# Diffusion Model

## Calculate FID Values
"""

# get paths of images in the real dataset
real_paths = os.listdir(my_datatoot + "real_dataset/")
real_paths = [my_datatoot + "real_dataset/" + i for i in real_paths]

# get paths of images in the fake dataset
fake_paths = os.listdir(my_datatoot + "diffusion_fake_dataset/")
fake_paths = [my_datatoot + "diffusion_fake_dataset/" + i for i in fake_paths]

# code copied from HW8 instructions
dims = 2048
block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
# instantiate an inception model
model = InceptionV3([block_idx]).to("cpu")
# calculate the mean and standard deviation of the distribution from the real
dataset
m1, s1 = calculate_activation_statistics(real_paths[:2048], model, device =
"cpu")
# calculate the mean and standard deviation of the distribution from the fake
dataset
m2, s2 = calculate_activation_statistics (fake_paths, model, device = "cpu")
# calculate and pring the FID value
fid_value = calculate_frechet_distance(m1, s1, m2, s2)
```

```
print(f'FID: {fid_value:.2f}')
```