BME 646/ ECE695DL: Homework 7

Karim A. ElSayed

12 Apr, 2022

1 Introduction

Generative Adversarial Networks (https://arxiv.org/abs/1406.2661) (GANs) are an exciting new inovation in machine learning. They are propalisitic data modeling approach that are based on neural networks and were first proposed by Ian Goodfellow and other researchers at the University of Montreal. Given training data, the goal of the GAN is to generate new data (synthetic data) that has very similar statistics to that of what it was trained on (i.e., the new synthetic data resemble the training data). GANs are able to achieve this by combining a Generator network, which learns to produce outputs similar to the inputs from random noise, with a Discriminator network, which classifies true data from the synthetic (output of the Generators). In simple terms, during the training process, the Generator tries to trick the Discriminator, and the Discriminator tries to correctly classify the real inputs from the sysnthetic ones.

In this homework, we are presented with the task of training a GAN with the Generator and Discriminator networks using the Large-scale CelebFaces Attributes (CelebA) Dataset (https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html). The goal of this GAN is to produce "Deep Fakes" of the CelebA dataset In the following section, the dataset, dataloader, Genererator and Discriminator architectures, and training of the adversarial network are discused.

2 Methodology

The focus of this assignment is on building and training a GAN that can produce "deep fakes" of the training dataset using random noise as an input to the Generator. The approach taken to complete the objectives of this homework can be broken into the following three tasks:

1. Loading the CelebA Dataset
2. Building the GAN - Generator and Discriminator
3. Training the GAN

The approaches taken to complete each task are detailed below.

2.1) Task 1: Loading the CelebA Dataset

CelebA dataset is a large-scale face attributes with more than 200,000 celebrity images, each with 40 attribute annotations. The images contain a large variety of poses and backgrounds (some celebrities have glasses, hats, etc...). For this homework, around 89,000 images are processed and cropped with final dimensions of 64x64. These images were divided into 'Train' and 'Test' directories that are in a directory named 'Data'. In order to load these data, I used the `torchvision.datasets` class and provided the path where both the 'Train' and 'Test' folders exist. Additionally, the dataset is scaled and normalized using the `transform` object. The batch size used for this assignment is 128.

2.2) Task 2: Building the GAN - Generator and Discriminator

In this assignment, I implemented two different Generators and Discriminators and compared their outputs. The first Generator-Discriminator pair is based on the PyTorch tutorial (https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html) by Nathan Inkawhich, which implements the network from the Deep Convolution GANs (DCGAN) paper (https://arxiv.org/pdf/1511.06434.pdf). In this architecture, the Discriminator is made up of convolution layers, batch norm layers, and LeakyReLU activations. The input is a 3x64x64 RGB image and the output is a scalar probability that the input is from the real data distribution. Each convolutional layer uses a 4x4 kernel, a 2x2 stride and a 1x1 padding for all but the final layer. On the other hand, the Generator is comprised of convolutional-transpose layers, batch norm layers, and ReLU activations. The input to the network is a latent vector, z, that is drawn from a normal distribution ($\mu = 0, \sigma^2 = 1$) and the output is a 3x64x64 RGB image. The convolution-transpose layers use a 4x4 kernel, a 2x2 stride and a 1x1 padding for all but the first layer to make the latent vector transform into a the desired image shape.

In the second set of Generator-Discriminator pair, I modified the previous architecture and incorborated linear layers and in the begining of the Generator and at the end of the Discriminator. This was inspired by the paper titled FCC-GAN (https://arxiv.org/pdf/1905.02417.pdf), in which the authors observed better results incorporating linear layers than using conventional CNN architectures. In the Discriminator, I added five fully-connected layers after the last convolution layer with LeakyReLu activations except for the final one. In the Generator's side, I added 3 fully-connected layers with ReLu activations in the begining followed by a batchnorm layer. My implementation of the FCC-GANs have a total of 15 layers for each network and contain about 7 million learnable parameters.

2.3) Task 3: Training the GAN

First, we give real images 1 as a label and fake images a 0 for a label. We use the binary cross-entropy loss function `nn.BCELoss()` and use Adam optimizars with learning rate 0.0002 and Beta1 = 0.5 as provided in the DCGAN paper. To train the GAN, we first train the discriminator network to maximize the propability of classifying inputs as real or fake. This is done in two steps (one with a batch of real images and the other is with a batch of fake images produced by the Generator. Once both losses and their gradient in a backward pass are calculated, the Discriminator's parameters are updated with an optimizer step.

Afterwords, the Generator is trained. Its output is first classified using the Discriminator with a target label of 1 (i.e., its goal), then its loss is computed along with the gradients in a backward pass. Then, the Generators parameters are updated with an optimizer step. For the training of both GAN networks, I used 5 epochs and generated images from both to compare at the end of each epoch. The results are shown in the end of section 3.

3 Implementation and Results

Below is the code and the results for this assignment

In [1]:
```
#import libraries
import os
import sys
import torchvision
import torchvision.transforms as tvt
import torchvision.transforms.functional as tvtF
from torchvision import datasets
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
%matplotlib inline
import numpy as np
import random
from PIL import Image
from torch.utils.data import Dataset
import pickle
import gc
import warnings


warnings.filterwarnings("ignore")
```

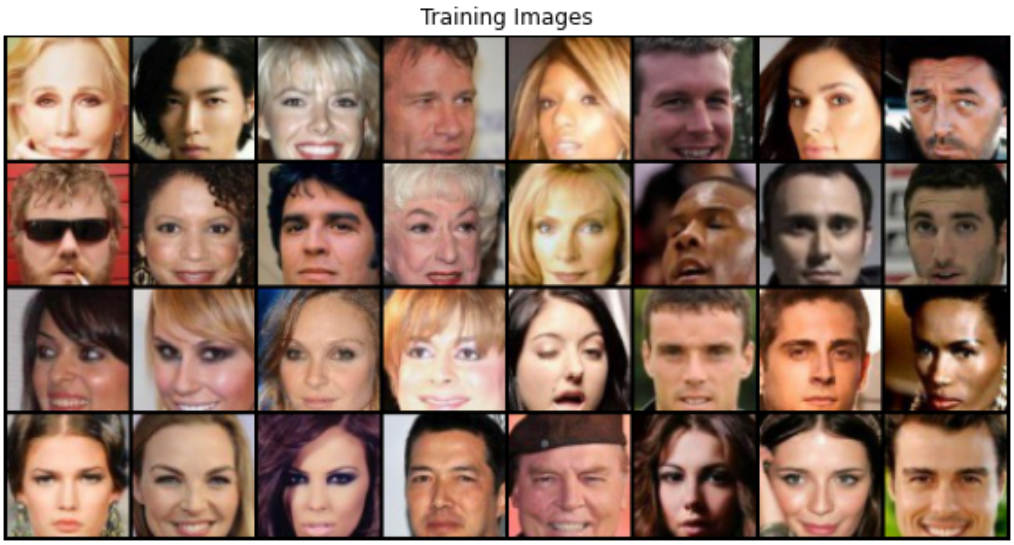
```
/home/tools/a/kelsayed/anaconda3/envs/DeepLearning/lib/python3.9/site-packages/torchvision/io/imag
e.py:11: UserWarning: Failed to load image Python extension: /home/tools/a/kelsayed/anaconda3/envs/
DeepLearning/lib/python3.9/site-packages/torchvision/image.so: undefined symbol: _ZNK2at10TensorBas
e21__dispatch_contiguousEN3c1012MemoryFormatE
  warn(f"Failed to load image Python extension: {e}")
```

3.1) Task 1: Loading the CelebA Dataset

Load the downloaded datasets

In [2]:
```
TrainPath = 'Data/'


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

train_data = datasets.ImageFolder(root = TrainPath, transform=transform)


batchsize = 128

train_data_loader = torch.utils.data.DataLoader(train_data, batch_size=batchsize, shuffle=True)
```

In [3]:
```
images,_ = next(iter(train_data_loader))
plt.figure(figsize=(10,10))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(torchvision.utils.make_grid(images[:32], padding=2, normalize=True),(1,2,0
)));
```



In [4]:
```
def weights_init(m):
    """
    Uses the DCGAN initializations for the weights. All model weights are
    randomly initialized from a Normal distribution with mu=0, sigma=0.02.
    """
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

In [16]:

```
'''
The following two networks, CNNGenerator() and CNNDiscrimintor(), are from teh DCGAN paper
and the implementation was used from the following PyTorch tutorial by Nathan Inkawhich
(https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html)
The input to the discriminator is a 64x64 color image (3x64x64) and the input to
to the generator is a latent vector of size 100
'''

class CNNGenerator(nn.Module):
    def __init__(self):
        super(CNNGenerator, self).__init__()

        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( 100, 64 * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(64 * 8),
            nn.ReLU(True),
            # state size. (64*8) x 4 x 4
            nn.ConvTranspose2d(64 * 8, 64 * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 4),
            nn.ReLU(True),
            # state size. (64*4) x 8 x 8
            nn.ConvTranspose2d( 64 * 4, 64 * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 2),
            nn.ReLU(True),
            # state size. (64*2) x 16 x 16
            nn.ConvTranspose2d( 64 * 2, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            # state size. (64) x 32 x 32
            nn.ConvTranspose2d( 64, 3, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (3) x 64 x 64
        )

    def forward(self, input):
        return self.main(input)

class CNNDiscriminator(nn.Module):
    def __init__(self):
        super(CNNDiscriminator, self).__init__()

        self.main = nn.Sequential(
            # input is (3) x 64 x 64
            nn.Conv2d(3, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (64) x 32 x 32
            nn.Conv2d(64, 64 * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (64*2) x 16 x 16
            nn.Conv2d(64 * 2, 64 * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (64*4) x 8 x 8
            nn.Conv2d(64 * 4, 64 * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (64*8) x 4 x 4
            nn.Conv2d(64 * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)
```

In [7]:
```
'''
The previous networks were modified to include linear layers and in the begining
of the generator and at the end of the discriminator. This was inspired by the
paper titled FCC-GAN (https://arxiv.org/pdf/1905.02417.pdf), in which the authors
observed better results incorporating linear layers than using conventional CNN architectures.
Both CNN and FCC networks were tested using the provided dataset
'''
class FCCGenerator(nn.Module):
    def __init__(self):
        super(FCCGenerator, self).__init__()

        self.fc_seq = nn.Sequential(
            nn.Linear(100, 64,bias=False),
            nn.ReLU(True),
            nn.Linear(64, 512,bias=False),
            nn.ReLU(True),
            nn.Linear(512, 64 * 8 * 4**2,bias=False),
            nn.BatchNorm1d(64 * 8 * 4**2)
        )

        self.convT_seq = nn.Sequential(

            nn.ConvTranspose2d(64 * 8, 64 * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 4),
            nn.ReLU(True),
            # state size. (64*4) x 8 x 8
            nn.ConvTranspose2d( 64 * 4, 64 * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 2),
            nn.ReLU(True),
            # state size. (64*2) x 16 x 16
            nn.ConvTranspose2d( 64 * 2, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            # state size. (64) x 32 x 32
            nn.ConvTranspose2d( 64, 3, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (3) x 64 x 64
        )

    def forward(self, x):
        out = self.fc_seq(x)
        out = torch.reshape(out, (out.shape[0], out.shape[1] // 4**2, 4, 4))
        out = self.convT_seq(out)
        return out

class FCCDiscriminator(nn.Module):
    def __init__(self):
        super(FCCDiscriminator, self).__init__()

        self.main = nn.Sequential(
            # input is (3) x 64 x 64
            nn.Conv2d(3, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (64) x 32 x 32
            nn.Conv2d(64, 64 * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (64*2) x 16 x 16
            nn.Conv2d(64 * 2, 64 * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (64*4) x 8 x 8
            nn.Conv2d(64 * 4, 64 * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 8),
            nn.LeakyReLU(0.2, inplace=True),
```

```
            # state size. (64*8) x 4 x 4
            nn.Flatten(),
            nn.Linear(64 * 8 * 4**2, 512,bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512,64,bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(64,16,bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(16,8,bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(8,1,bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)
```

In [8]:
```
#Ensure that the FCC outputs correct shape
gen = FCCGenerator()
z = torch.rand(64, 100)
print(gen(z).shape)

disc = FCCDiscriminator()
inputs = torch.rand(64, 3, 64, 64)
print(disc(inputs).shape)
```

```
torch.Size([64, 3, 64, 64])
torch.Size([64, 1])
```

In [11]:
```
##Finds the number of layers and the parameters in the DNN:
net = FCCDiscriminator()
number_of_learnable_params = sum(p.numel() for p in net.parameters() if p.requires_grad)
num_layers = len(list(net.parameters()))
print("\nThe number of layers in the FCC-Discriminator: %d" % num_layers)
print("\nThe number of learnable parameters in the FCC-Discriminator: %d" % number_of_learnable_par
ams)


net = FCCGenerator()
number_of_learnable_params = sum(p.numel() for p in net.parameters() if p.requires_grad)
num_layers = len(list(net.parameters()))
print("\n\nThe number of layers in the FCC-Generator: %d" % num_layers)
print("\nThe number of learnable parameters in the FCC-Generator: %d" % number_of_learnable_params)
```

```
The number of layers in the FCC-Discriminator: 15

The number of learnable parameters in the FCC-Discriminator: 6985608


The number of layers in the FCC-Generator: 15

The number of learnable parameters in the FCC-Generator: 7006336
```

In [13]:

```
def run_gan_code(discriminator, generator, train_dataloader,batchsize=64,epochs=10, Linear=True):
    """
    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.
    """

    #  Set the number of channels for the 1x1 input noise vectors for the Generator:
    nz = 100

    netD = discriminator.to(device)
    netG = generator.to(device)


    #  Initialize the parameters of the Discriminator and the Generator networks according to the
    #  definition of the "weights_init()" method:
    netD.apply(weights_init)
    netG.apply(weights_init)
    #  We will use a the same noise batch to periodically check on the progress made for the Genera
tor:
    fixed_noise = torch.randn(batchsize, nz,1,1, device=device)
    if Linear:
        fixed_noise = torch.randn(batchsize, nz, device=device)

    #  Establish convention for real and fake labels during training
    real_label = 1
    fake_label = 0
    #  Adam optimizers for the Discriminator and the Generator:
    optimizerD = torch.optim.Adam(netD.parameters(), lr=0.0002, betas=(0.5, 0.999))
    optimizerG = torch.optim.Adam(netG.parameters(), lr=0.0002, betas=(0.5, 0.999))
    #  Establish the criterion for measuring the loss at the output of the Discriminator network:
    criterion = nn.BCELoss()
    #  We will use these lists to store the results accumulated during training:
    img_list = []
    G_losses = []
    D_losses = []
    mean_D_losses = []
    mean_G_losses = []
    iters = 0
    print("\n\nStarting Training Loop...\n\n")
#     start_time = time.perf_counter()
    for epoch in range(epochs):
        g_losses_per_print_cycle = []
        d_losses_per_print_cycle = []
        # For each batch in the dataloader
        for i, data in enumerate(train_dataloader):
            #  As indicated in the DCGAN part of the doc section at the beginning of this file, the
GAN
            #  training boils down to carrying out a max-min 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 operatio
n
            #  affects both terms of the criterion shown in the doc section, it has two parts: In t
he
            #  first part we apply the Discriminator to the training images using 1.0 as the targe
t;
            #  and, in the second part, we supply to the Discriminator the output of the Generator
            #  and use -1.0 as the target. In what follows, the Discriminator is being applied to
```

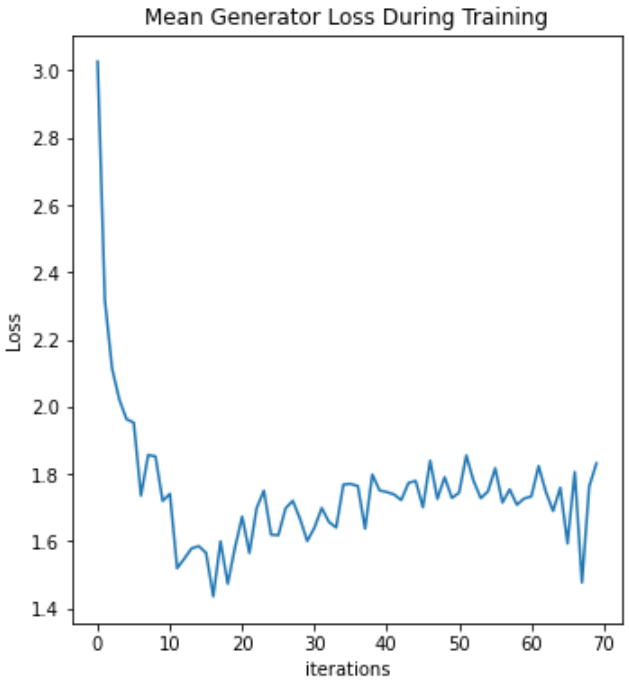
```
            #  the training images:
            netD.zero_grad()
            real_images_in_batch = data[0].to(device)
            #  Need to know how many images we pulled in since at the tailend of the dataset, the
            #  number of images may not equal the user-specified batch size:
            b_size = real_images_in_batch.size(0)
            label = torch.full((b_size,), real_label, dtype=torch.float, device=device)
            output = netD(real_images_in_batch).view(-1)
            lossD_for_reals = criterion(output, label)
            lossD_for_reals.backward()

            #  That brings us the second part of what it takes to carry out the max operation on th
e
            #  DCGAN criterion shown in the doc section at the beginning of this file.  This part
            #  calls for applying the Discriminator to the images produced by the Generator from no
ise:
            noise = torch.randn(b_size, nz,1,1, device=device)
            if Linear:
                noise = torch.randn(b_size, nz, device=device)
            fakes = netG(noise)
            label.fill_(fake_label)
            output = netD(fakes.detach()).view(-1)
            lossD_for_fakes = criterion(output, label)
            lossD_for_fakes.backward()
            lossD = lossD_for_reals + lossD_for_fakes
            d_losses_per_print_cycle.append(lossD)
            optimizerD.step()

            #  That brings to the min part of the max-min optimization described in 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 maximiz
e
            #  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:
            netG.zero_grad()
            label.fill_(real_label)
            output = netD(fakes).view(-1)
            lossG = criterion(output, label)
            g_losses_per_print_cycle.append(lossG)
            lossG.backward()
            optimizerG.step()

            if (i+1) % 50 == 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))
                mean_D_losses.append(mean_D_loss)
                mean_G_losses.append(mean_G_loss)
                if (i+1)%100 ==0:
                    print("[epoch=%d/%d   iter=%4d]     mean_D_loss=%7.4f    mean_G_loss=%7.4f" %
                                  ((epoch+1),epochs,(i+1), mean_D_loss, mean_G_loss))
                d_losses_per_print_cycle = []
                g_losses_per_print_cycle = []
            G_losses.append(lossG.item())
            D_losses.append(lossD.item())
            if (i == len(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

    #  At the end of training, make plots from the data in G_losses and D_losses:
    fig,axes = plt.subplots(1,2,figsize= (12,6))
    axes[0].plot(mean_G_losses)
    axes[0].set_xlabel("iterations")
```

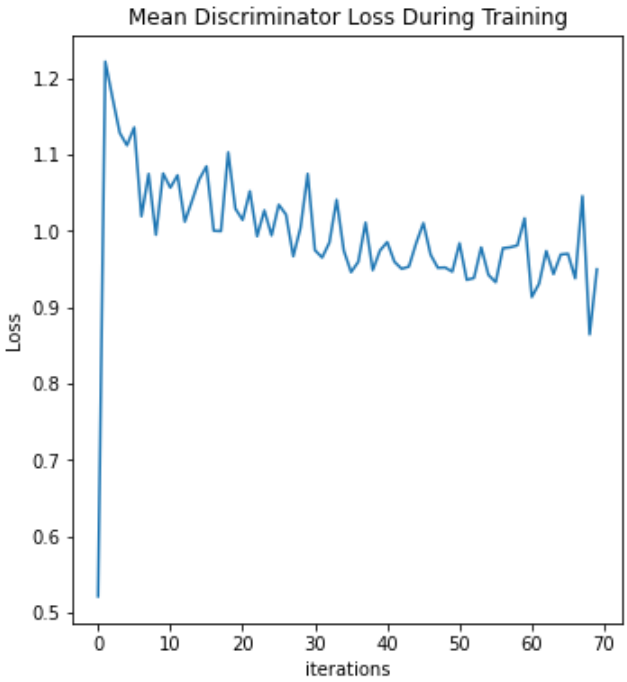
```
    axes[0].set_ylabel("Loss")
    axes[0].set_title("Mean Generator Loss During Training")
    axes[1].plot(mean_D_losses)
    axes[1].set_title("Mean Discriminator Loss During Training")
    axes[1].set_xlabel("iterations")
    axes[1].set_ylabel("Loss")
    plt.savefig("gen_and_disc_loss_training.png")
    plt.show()

    #  Make a side-by-side comparison of a batch-size sampling of real images drawn from the
    #  training data and what the Generator is capable of producing at the end of training:
    real_batch = next(iter(train_dataloader))
    real_batch = real_batch[0]
    plt.figure(figsize=(8,8))
    plt.axis("off")
    plt.title("Real Images")
    plt.imshow(np.transpose(torchvision.utils.make_grid(real_batch.to(device)[:64],
                                        padding=1, pad_value=1, normalize=True).cpu(),(1,2,0)))
    plt.savefig('Real_Images.jpg')
    plt.show()
    plt.figure(figsize=(8,8))
    plt.axis("off")
    plt.title("Fake Images")
    plt.imshow(np.transpose(img_list[-1][:,:521,:],(1,2,0)))
    plt.savefig('Fake_Images.jpg')
    plt.show()
    return mean_D_losses, mean_G_losses, img_list
```

In [15]:

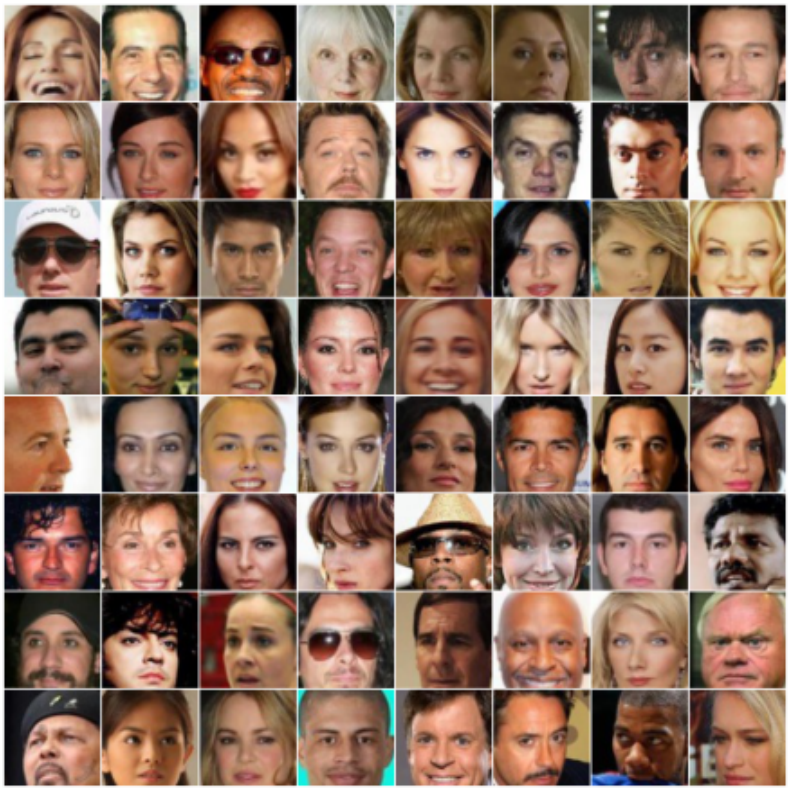
```
#Testing the FCC networks for 5 epochs
seed = 101
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
np.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)
gc.collect()
torch.cuda.empty_cache()
device = torch.device("cuda:0") #if torch.cuda.is_available() else "cpu")

Discriminator = FCCDiscriminator()
Generator = FCCGenerator()
D_losses, G_losses,fake_imgs =run_gan_code(Discriminator,Generator,train_data_loader,
                                           batchsize=batchsize, epochs=5,Linear=True)


file_name = "FCC_fakeimgs.pkl"

open_file = open(file_name, "wb")
pickle.dump(fake_imgs, open_file)
open_file.close()

file_name = "FCC_G_loss.pkl"

open_file = open(file_name, "wb")
pickle.dump(G_losses, open_file)
open_file.close()

file_name = "FCC_D_loss.pkl"

open_file = open(file_name, "wb")
pickle.dump(D_losses, open_file)
open_file.close()
```

```
Starting Training Loop...

[epoch=1/5   iter= 100]     mean_D_loss= 1.2214    mean_G_loss= 2.3186
[epoch=1/5   iter= 200]     mean_D_loss= 1.1280    mean_G_loss= 2.0214
[epoch=1/5   iter= 300]     mean_D_loss= 1.1351    mean_G_loss= 1.9524
[epoch=1/5   iter= 400]     mean_D_loss= 1.0743    mean_G_loss= 1.8564
[epoch=1/5   iter= 500]     mean_D_loss= 1.0749    mean_G_loss= 1.7194
[epoch=1/5   iter= 600]     mean_D_loss= 1.0725    mean_G_loss= 1.5193
[epoch=1/5   iter= 700]     mean_D_loss= 1.0391    mean_G_loss= 1.5782
[epoch=2/5   iter= 100]     mean_D_loss= 1.0842    mean_G_loss= 1.5650
[epoch=2/5   iter= 200]     mean_D_loss= 0.9994    mean_G_loss= 1.5993
[epoch=2/5   iter= 300]     mean_D_loss= 1.0287    mean_G_loss= 1.5807
[epoch=2/5   iter= 400]     mean_D_loss= 1.0516    mean_G_loss= 1.5645
[epoch=2/5   iter= 500]     mean_D_loss= 1.0265    mean_G_loss= 1.7500
[epoch=2/5   iter= 600]     mean_D_loss= 1.0339    mean_G_loss= 1.6175
[epoch=2/5   iter= 700]     mean_D_loss= 0.9664    mean_G_loss= 1.7199
[epoch=3/5   iter= 100]     mean_D_loss= 1.0743    mean_G_loss= 1.6004
[epoch=3/5   iter= 200]     mean_D_loss= 0.9648    mean_G_loss= 1.6985
[epoch=3/5   iter= 300]     mean_D_loss= 1.0403    mean_G_loss= 1.6403
[epoch=3/5   iter= 400]     mean_D_loss= 0.9456    mean_G_loss= 1.7702
[epoch=3/5   iter= 500]     mean_D_loss= 1.0105    mean_G_loss= 1.6370
[epoch=3/5   iter= 600]     mean_D_loss= 0.9740    mean_G_loss= 1.7509
[epoch=3/5   iter= 700]     mean_D_loss= 0.9591    mean_G_loss= 1.7389
[epoch=4/5   iter= 100]     mean_D_loss= 0.9529    mean_G_loss= 1.7729
[epoch=4/5   iter= 200]     mean_D_loss= 1.0099    mean_G_loss= 1.7005
[epoch=4/5   iter= 300]     mean_D_loss= 0.9514    mean_G_loss= 1.7256
[epoch=4/5   iter= 400]     mean_D_loss= 0.9463    mean_G_loss= 1.7285
[epoch=4/5   iter= 500]     mean_D_loss= 0.9355    mean_G_loss= 1.8549
[epoch=4/5   iter= 600]     mean_D_loss= 0.9780    mean_G_loss= 1.7277
[epoch=4/5   iter= 700]     mean_D_loss= 0.9324    mean_G_loss= 1.8168
[epoch=5/5   iter= 100]     mean_D_loss= 0.9782    mean_G_loss= 1.7536
[epoch=5/5   iter= 200]     mean_D_loss= 1.0162    mean_G_loss= 1.7268
[epoch=5/5   iter= 300]     mean_D_loss= 0.9305    mean_G_loss= 1.8232
[epoch=5/5   iter= 400]     mean_D_loss= 0.9432    mean_G_loss= 1.6896
[epoch=5/5   iter= 500]     mean_D_loss= 0.9697    mean_G_loss= 1.5928
[epoch=5/5   iter= 600]     mean_D_loss= 1.0453    mean_G_loss= 1.4766
[epoch=5/5   iter= 700]     mean_D_loss= 0.9494    mean_G_loss= 1.8315
```

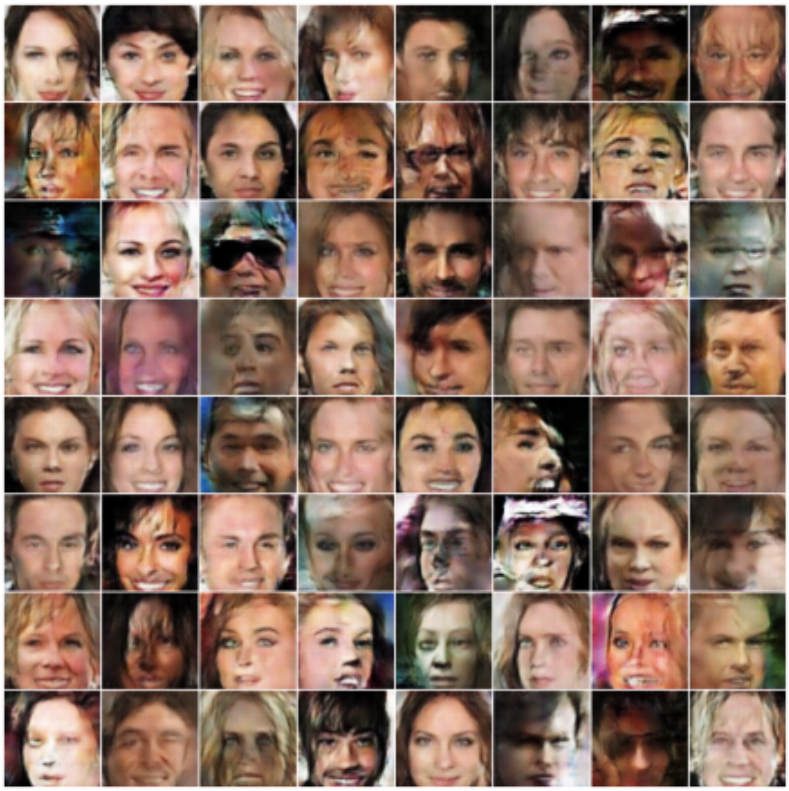




Real Images



Fake Images

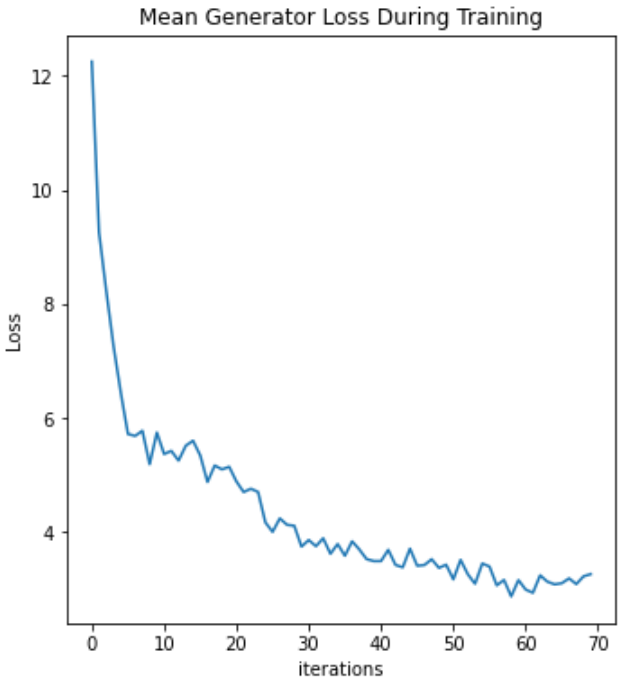


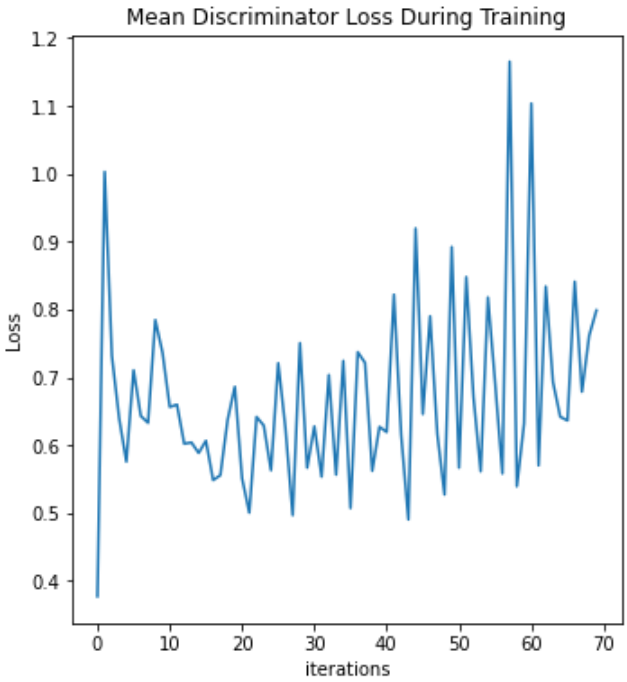
Testing the CNN Networks

In [18]:

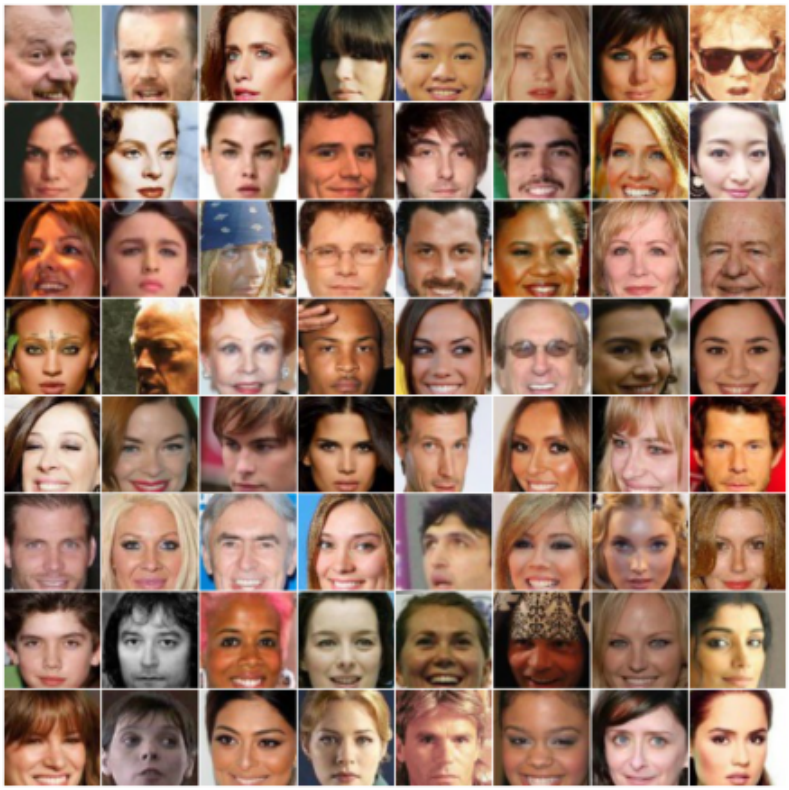
```
#Testing the CNN networks for 5 epochs
seed = 101
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
np.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)
gc.collect()
torch.cuda.empty_cache()
device = torch.device("cuda:0") #if torch.cuda.is_available() else "cpu")

Discriminator = CNNDiscriminator()
Generator = CNNGenerator()
D_losses2, G_losses2,fake_imgs2 =run_gan_code(Discriminator,Generator,train_data_loader,
                                              batchsize=batchsize, epochs=5,Linear=False)


file_name = "CNN_fakeimgs.pkl"

open_file = open(file_name, "wb")
pickle.dump(fake_imgs2, open_file)
open_file.close()

file_name = "CNN_G_loss.pkl"

open_file = open(file_name, "wb")
pickle.dump(G_losses2, open_file)
open_file.close()

file_name = "CNN_D_loss.pkl"

open_file = open(file_name, "wb")
pickle.dump(D_losses2, open_file)
open_file.close()
```

```
Starting Training Loop...

[epoch=1/5   iter= 100]     mean_D_loss= 1.0029    mean_G_loss= 9.2526
[epoch=1/5   iter= 200]     mean_D_loss= 0.6398    mean_G_loss= 7.2665
[epoch=1/5   iter= 300]     mean_D_loss= 0.7104    mean_G_loss= 5.7201
[epoch=1/5   iter= 400]     mean_D_loss= 0.6328    mean_G_loss= 5.7775
[epoch=1/5   iter= 500]     mean_D_loss= 0.7362    mean_G_loss= 5.7464
[epoch=1/5   iter= 600]     mean_D_loss= 0.6597    mean_G_loss= 5.4239
[epoch=1/5   iter= 700]     mean_D_loss= 0.6039    mean_G_loss= 5.5150
[epoch=2/5   iter= 100]     mean_D_loss= 0.6064    mean_G_loss= 5.3441
[epoch=2/5   iter= 200]     mean_D_loss= 0.5557    mean_G_loss= 5.1683
[epoch=2/5   iter= 300]     mean_D_loss= 0.6860    mean_G_loss= 5.1457
[epoch=2/5   iter= 400]     mean_D_loss= 0.5003    mean_G_loss= 4.7034
[epoch=2/5   iter= 500]     mean_D_loss= 0.6289    mean_G_loss= 4.7041
[epoch=2/5   iter= 600]     mean_D_loss= 0.7208    mean_G_loss= 4.0050
[epoch=2/5   iter= 700]     mean_D_loss= 0.4966    mean_G_loss= 4.1294
[epoch=3/5   iter= 100]     mean_D_loss= 0.5666    mean_G_loss= 3.7471
[epoch=3/5   iter= 200]     mean_D_loss= 0.5534    mean_G_loss= 3.7546
[epoch=3/5   iter= 300]     mean_D_loss= 0.5563    mean_G_loss= 3.6213
[epoch=3/5   iter= 400]     mean_D_loss= 0.5070    mean_G_loss= 3.5865
[epoch=3/5   iter= 500]     mean_D_loss= 0.7214    mean_G_loss= 3.6963
[epoch=3/5   iter= 600]     mean_D_loss= 0.6270    mean_G_loss= 3.4934
[epoch=3/5   iter= 700]     mean_D_loss= 0.8219    mean_G_loss= 3.6913
[epoch=4/5   iter= 100]     mean_D_loss= 0.4902    mean_G_loss= 3.3841
[epoch=4/5   iter= 200]     mean_D_loss= 0.6457    mean_G_loss= 3.4115
[epoch=4/5   iter= 300]     mean_D_loss= 0.6161    mean_G_loss= 3.5267
[epoch=4/5   iter= 400]     mean_D_loss= 0.8922    mean_G_loss= 3.4298
[epoch=4/5   iter= 500]     mean_D_loss= 0.8478    mean_G_loss= 3.5151
[epoch=4/5   iter= 600]     mean_D_loss= 0.5613    mean_G_loss= 3.0965
[epoch=4/5   iter= 700]     mean_D_loss= 0.6910    mean_G_loss= 3.3977
[epoch=5/5   iter= 100]     mean_D_loss= 1.1653    mean_G_loss= 3.1630
[epoch=5/5   iter= 200]     mean_D_loss= 0.6321    mean_G_loss= 3.1605
[epoch=5/5   iter= 300]     mean_D_loss= 0.5699    mean_G_loss= 2.9377
[epoch=5/5   iter= 400]     mean_D_loss= 0.6927    mean_G_loss= 3.1329
[epoch=5/5   iter= 500]     mean_D_loss= 0.6362    mean_G_loss= 3.1055
[epoch=5/5   iter= 600]     mean_D_loss= 0.6786    mean_G_loss= 3.0905
[epoch=5/5   iter= 700]     mean_D_loss= 0.7986    mean_G_loss= 3.2649
```





Real Images



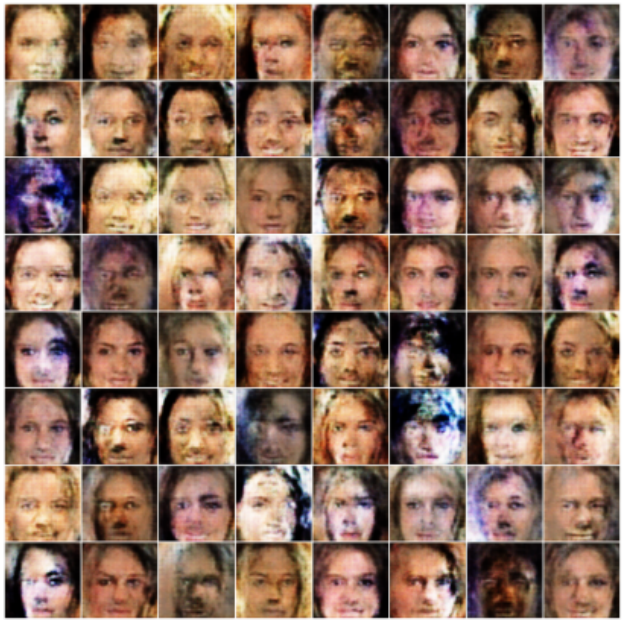
Fake Images

Show Progress of fake images at end of each epoch for both FCC and CNN networks and Compare them

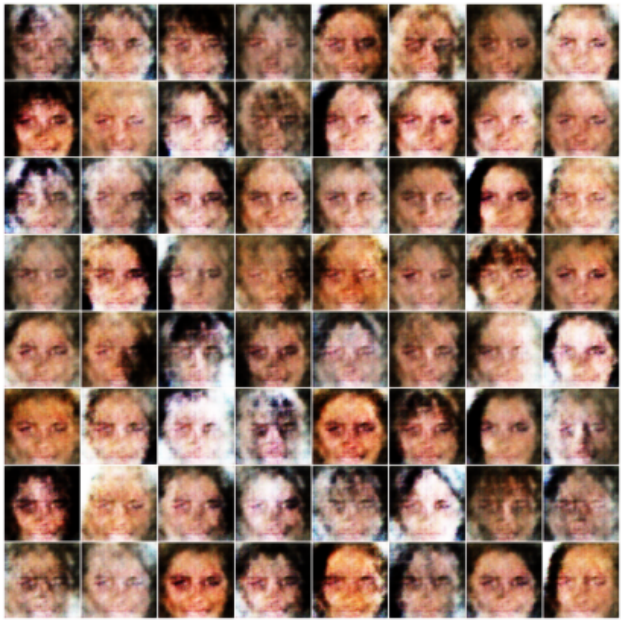
In [24]:

```
for idx, (imgsFCC,imgsCNN) in enumerate(zip(fake_imgs,fake_imgs2)):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16,8))
#     plt.figure(figsize=(8,8))
    ax1.axis("off")
    ax1.set_title(f"FCC Fake Images for Epoch {idx+1}")
    ax1.imshow(np.transpose(imgsFCC[:,:521,:],(1,2,0)))  #only showing 64 images
    ax2.axis("off")
    ax2.set_title(f"CNN Fake Images for Epoch {idx+1}")
    ax2.imshow(np.transpose(imgsCNN[:,:521,:],(1,2,0)))  #only showing 64 images
```

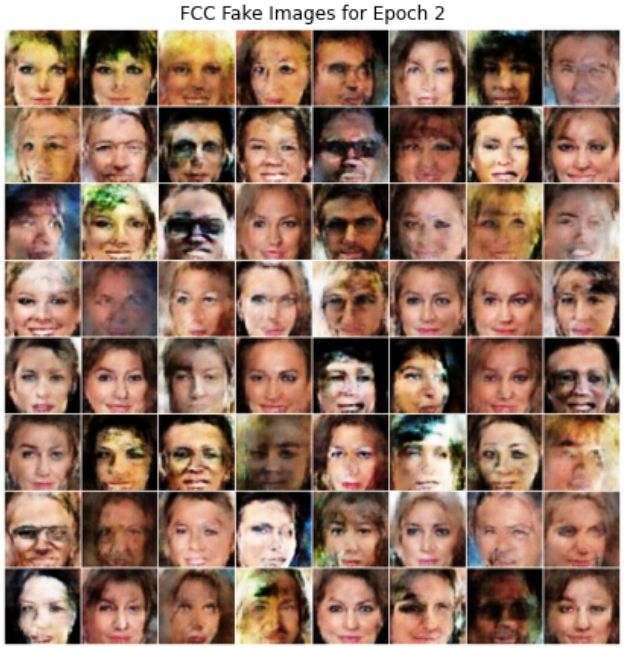
FCC Fake Images for Epoch 1



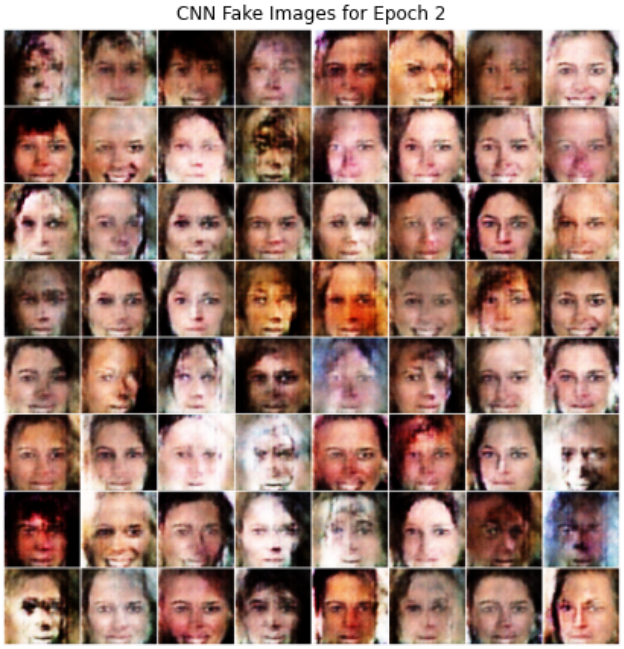
CNN Fake Images for Epoch 1



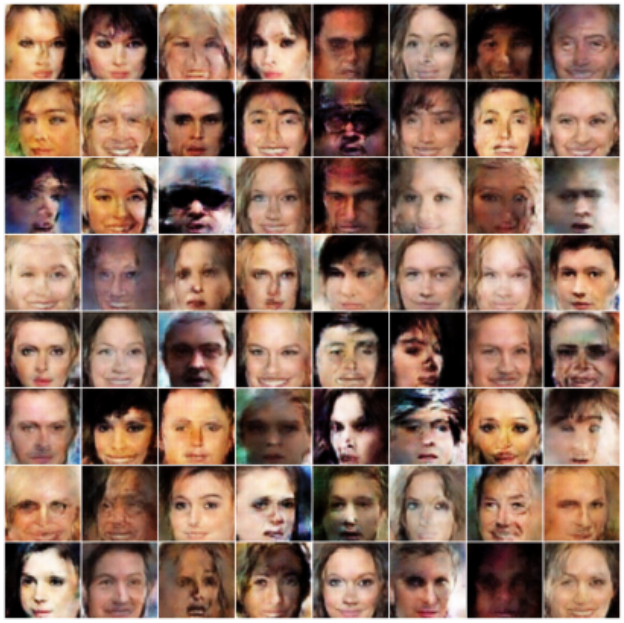
FCC Fake Images for Epoch 2



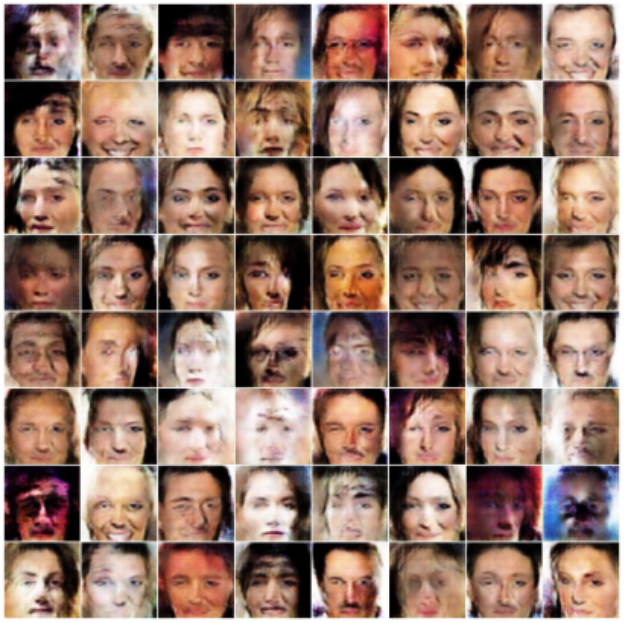
CNN Fake Images for Epoch 2



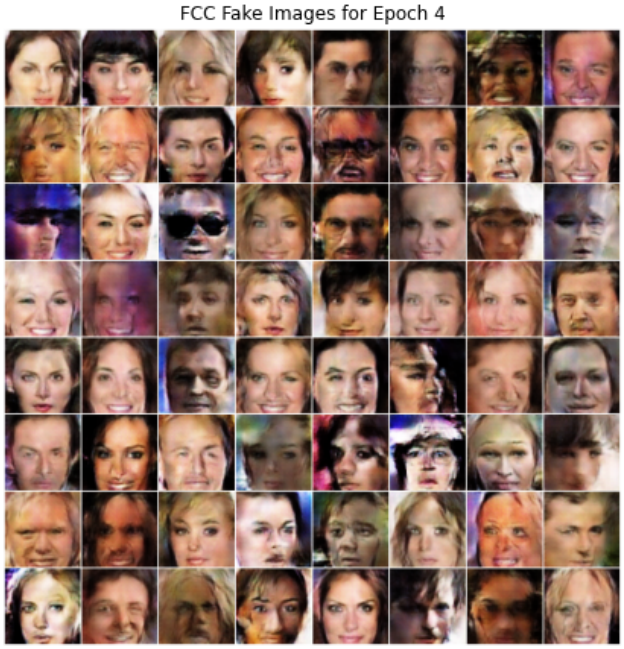
FCC Fake Images for Epoch 3



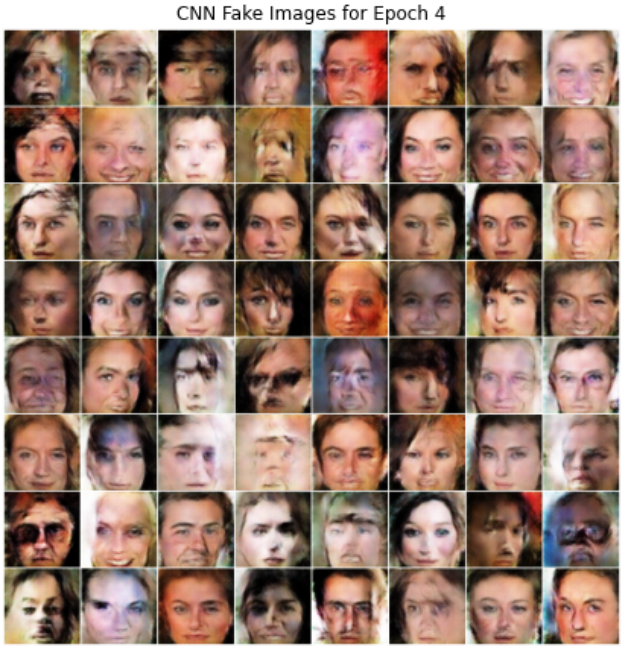
CNN Fake Images for Epoch 3

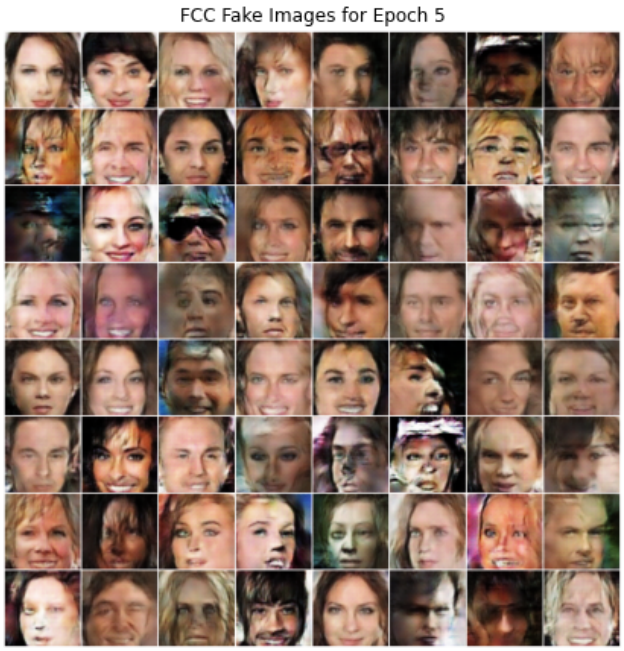


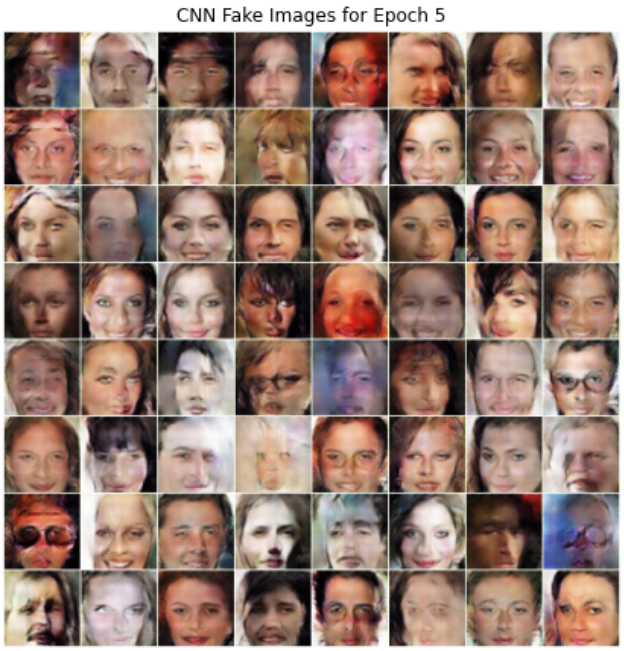
FCC Fake Images for Epoch 4



CNN Fake Images for Epoch 4







4 Lesson Learned

Through this assignment, I learned the how to implement a Generator network and use it along a Discriminator building a GAN. I was able to understand how to train an adverserial network using max-min optimizaiton. The plots of the loss vs. iterations look good and it does not seem like mode collapse occured. I plan on tweeking some of the parameters here to see how I could potentially improve the final reults. Possible changes are number of epochs, initializations, batch size, etc.

5 Suggested Enhancements

Great challenging assignment.

In [ ]: