ECE 695DL - Deep Learning - HW7 Report

Brad Fitzgerald

April 11, 2022

1 Introduction

Up to this point in the course, we have focused on using deep networks to learn how to characterize particular features of existing images. Now, we shift our attention to addressing the problem of how to create synthetic data which has the characteristics to imply that it was pulled from a particular probability distribution or set of real data. In the world of image processing, this looks like generating synthetic images which appear to be real. To tackle this task, we make use of the concept of a Generative Adversarial Network (GAN), which was first introduced by [1]. GANs operate by training two networks simultaneously. A discriminator network is trained to discriminate between real images from a training dataset and fake images generated by a second generator network. The generator network is trained to generate images which the discriminator will label as real. In this assignment, we use a GANs framework to generate synthetic face images using the celebrity image dataset provided in Brightspace.

2 Methodology

All programming completed for this assignment was done using Google Colab and followed the instructions provided for the assignment.

2.1 Dataset

Training data for this assignment was provided by the instructors via Brightspace. Both the training and testing sets from the dataset provided were combined (so all images were used for training), creating a training set of 89,931 64x64 pixel celebrity face images.

2.2 Implementation of GAN Training

Network Architecture: Two networks were created: a discriminator network, and a generator network. The purpose of the discriminator network is to discern whether input images are real or fake. The purpose of the generator network is to generate synthetic images which will appear ”real” to the discriminator network. The two networks were created with the architectures shown in Figure 1.

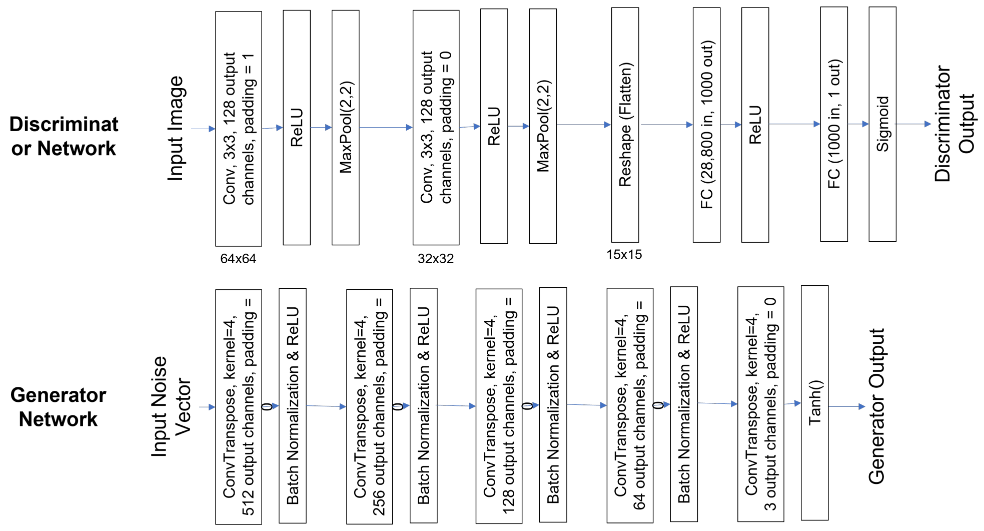
Discriminat or Network
Input Image
Conv, 3x3, 128 output channels, padding = 1
ReLU
MaxPool(2,2)
Conv, 3x3, 128 output channels, padding = 0
ReLU
MaxPool(2,2)
Reshape (Flatten)
FC (28,800 in, 1000 out)
ReLU
FC (1000 in, 1 out)
Sigmoid
Discriminator Output
64x64
32x32
15x15
Generator Network
Input Noise Vector
ConvTranspose, kernel=4, 512 output channels, padding = 0
Batch Normalization & ReLU
ConvTranspose, kernel=4, 256 output channels, padding = 0
Batch Normalization & ReLU
ConvTranspose, kernel=4, 128 output channels, padding = 0
Batch Normalization & ReLU
ConvTranspose, kernel=4, 64 output channels, padding = 0
Batch Normalization & ReLU
ConvTranspose, kernel=4, 3 output channels, padding = 0
Tanh()
Generator Output

Figure 1: Architecture of the designed Discriminator and Generator networks.

The Discriminator network uses the architecture of a network developed for Homework 4 (specifically, network 3 from Homework 4). Code for this network was copied from my Homework 4 code, with the minor adjustment that the final fully-connected layer was changed to output a single value and pass this value through a sigmoid function (in order to conduct binary classification). The discriminator network was trained using binary cross entropy (BCE) loss. In each iteration of training, the Discriminator was trained on 10 real images and 10 fake images (as produced by the Generator network).

The Generator network uses architecture presented by Prof. Kak in his lecture notes for GANs as presented in Week 11 of the Spring 2022 ECE 695 Deep Learning course. Specifically, the code for the definition of the Generator class variable was copied from slide 71 of these lecture notes, available at `https://engineering.purdue.edu/DeepLearn/pdf-kak/GAN.pdf`. The Generator takes an input of a 100-value noise vector and transforms it into a 64x64 pixel color image using five layers of transpose convolution. The Generator network is trained by computing the BCE loss between the Discriminator classification of images (produced by the Generator network) and "true" labels (i.e. if the Discriminator labelled every image as real with high confidence, the Generator loss would be low). My code for implementing the simultaneous training of these two networks was inspired by the code presented in the referenced lecture slides from Prof. Kak, but was implemented myself.

Network Parameters/Inputs: The following parameters were used for training the neural network: 89,931 training images, learning rate of `1e-3`, Adam optimizer, batch size of 10, and 5 epochs. It is important to note that running the `hw07_training.py` script assumes that the paths `hw07_data/Train/Data/Train/` and `hw07_data/Train/Data/Test/`exist in the working directory where the training script is run and contains the training data shared with us via Brightspace. The paths to these locations are coded in lines 253 and 254 of the script and can be adjusted if desired.

Training Outputs: During training, the training script will display the total computed loss every 500 iterations (batches). At each of these points, the network will also display five sample images reflecting the outputs of Generator network using fixed, common input noise vectors. An image which shows a sampling of these outputs, showing the evolution of these generated images over training, is shown in Figure 2.

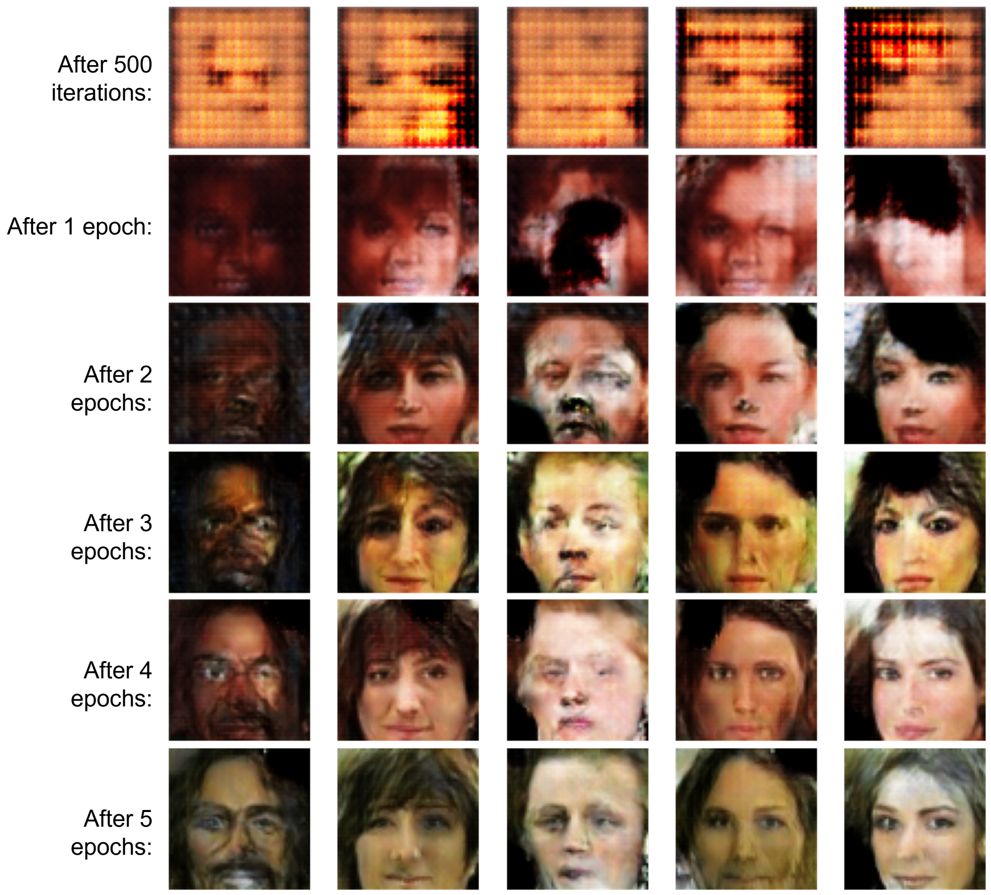

Figure 2: Sample images produced by the Generator network at the end of each training epoch. Images were generated using the same input noise vectors for each epoch.

Validation: Since we were only instructed to use visual inspection of the generated images as our validation, I chose to produce 10 new noise images at the end of training as a simple validation test. These images will be displayed in the next report section.

3 Implementation and Results

What follows is the code appearing in `hw07_training.py`, which contains all code used to train and validate the face generator network.

```
#####################################################
# ECE 695 - Deep Learning - Spring 2022
# Homework #7 - Final Version
######################################################
print('New Version!')

from torch import nn
import torch
import os
import numpy
import random
import argparse
import glob
from torch.utils.data import DataLoader, Dataset
from PIL import Image
from torchvision import transforms
import torch.nn.functional as functional
import matplotlib.pyplot as plt

# Make randomness reproducible (copied from Week 2 Kak slide #73)
seed = 0
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)

def seed_worker(worker_id):
        numpy.random.seed(0)

##############################
# BF_Dataset class - Copied from my Homework #4 scripts and modified
##############################

class BF_Dataset(Dataset):
  def __init__(self, data_path1, data_path2, transform):
    self.data_path = data_path
    self.transform = transform
    self.path_list = []
    self.label_list = []
    count = 0
    for path in glob.glob(f'{data_path1}/*'):
      self.path_list.append(path)
      count +=1
    for path in glob.glob(f'{data_path2}/*'):
      self.path_list.append(path)
      count += 1
    print('got' + str(count) + 'images for training')


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

  def __getitem__(self, idx):
    path = self.path_list[idx]
    img = Image.open(path)
    img_tsr = self.transform(img)
    return img_tsr


##############################
# TemplateNet class - Copied from my Homework #4 scripts and modified
##############################

class TemplateNet(nn.Module):
  def __init__(self, num_conv_layers, padding):
    super(TemplateNet, self).__init__()
    self.padding = padding
    self.num_conv_layers = num_conv_layers

    if self.num_conv_layers == 1:
      if self.padding==0:
        self.conv1 = nn.Conv2d(3, 128, 3) ## (A)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(31*31*128, 1000) ## (C)
        self.fc2 = nn.Linear(1000, 1)
      else:
        self.conv1 = nn.Conv2d(3, 128, 3, padding=self.padding)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(32*32*128, 1000) ## (C)
        self.fc2 = nn.Linear(1000, 1)
    elif self.num_conv_layers == 2:
      if self.padding==0:
        self.conv1 = nn.Conv2d(3, 128, 3) ## (A)
        self.conv2 = nn.Conv2d(128, 128, 3) ## (B)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(14*14*128, 1000) ## (C)
        self.fc2 = nn.Linear(1000, 1)
      else:
        self.conv1 = nn.Conv2d(3, 128, 3, padding=self.padding) ## (A)
        self.conv2 = nn.Conv2d(128, 128, 3) ## (B)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(15*15*128, 1000) ## (C)
        self.fc2 = nn.Linear(1000, 1)

  def forward(self, x):
    if self.num_conv_layers == 1:
      if self.padding == 0:
        x = self.pool(functional.relu(self.conv1(x)))
        x = x.view(-1, 31*31*128) ## (E)
      else:
        x = self.pool(functional.relu(self.conv1(x)))
        x = x.view(-1, 32*32*128) ## (E)
    elif self.num_conv_layers==2:
      if self.padding == 0:
        x = self.pool(functional.relu(self.conv1(x)))
        x = self.pool(functional.relu(self.conv2(x))) ## (D)
        x = x.view(-1, 14*14*128) ## (E)
      else:
        x = self.pool(functional.relu(self.conv1(x)))
        x = self.pool(functional.relu(self.conv2(x))) ## (D)
        x = x.view(-1, 15*15*128) ## (E)

    x = functional.relu(self.fc1(x))
    x = self.fc2(x)
    x = nn.Sigmoid()(x)
    return x

##################################
# Generator_Net class - Fully Copied from Prof. Kak's slide 71 from GAN slides
# This definition of the class Generator_Net as shown below is attributed to Professor
# Kak at Purdue University, accessed through his lecture slides on GANs from
# the Deep Learning class for Spring 2022
# Link to slides: https://engineering.purdue.edu/DeepLearn/pdf-kak/GAN.pdf
##################################

class Generator_Net(nn.Module):
  def __init__(self):
    super(Generator_Net, self).__init__()
    self.latent_to_image = nn.ConvTranspose2d( 100, 512, kernel_size=4, stride=1, padding=0, bias=False)
    self.upsampler2 = nn.ConvTranspose2d( 512, 256, kernel_size=4, stride=2, padding=1, bias=False)
    self.upsampler3 = nn.ConvTranspose2d (256, 128, kernel_size=4, stride=2, padding=1, bias=False)
    self.upsampler4 = nn.ConvTranspose2d (128, 64, kernel_size=4, stride=2, padding=1, bias=False)
    self.upsampler5 = nn.ConvTranspose2d( 64, 3, kernel_size=4, stride=2, padding=1, bias=False)
    self.bn1 = nn.BatchNorm2d(512)
    self.bn2 = nn.BatchNorm2d(256)
    self.bn3 = nn.BatchNorm2d(128)
    self.bn4 = nn.BatchNorm2d(64)
    self.tanh = nn.Tanh()

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

##################################
# run_code_for_training function
# This code for training was heavily inspired by Prof. Kak's slides referenced
# above, but implemented myself
##################################

def run_code_for_training(D_net, G_net, batch_size, epochs):
  var = 500
  noise_channels = 100
  cpu = torch.device("cuda:0")
  D_net = D_net.to(cpu)
  G_net = G_net.to(cpu)

  #Create fixed noise vectors for testing throughout process
  test_noise = torch.randn(batch_size, noise_channels, 1, 1, device=cpu)

  real = 1
  fake = 0

  D_net_optimizer = torch.optim.Adam(D_net.parameters(), lr=1e-3)
  G_net_optimizer = torch.optim.Adam(G_net.parameters(), lr=1e-3)


  criterion = nn.BCELoss()

  full_D_net_loss = []
  full_G_net_loss = []
  iters = 0

  for epoch in range(epochs):
    D_net_running_loss = []
    G_net_running_loss = []

    for i, data in enumerate(train_data_loader):
      #Train Discriminator on real images
      D_net.zero_grad()
      real_imgs = data.to(cpu)
      iter_batch_size = real_imgs.size(0)
      label = torch.full((iter_batch_size,), real, dtype=torch.float, device=cpu)
      real_output = D_net(real_imgs).view(-1)
      D_net_real_error = criterion(real_output, label)
      D_net_real_error.backward(retain_graph=True)

      #Train Discriminator on fake images
      noise_vecs = torch.randn(iter_batch_size, noise_channels, 1, 1, device=cpu)
      fake_images = G_net(noise_vecs)
      label.fill_(fake)
      fake_output = D_net(fake_images.detach()).view(-1)
      D_net_fake_error = criterion(fake_output, label)
      D_net_fake_error.backward()
      D_net_total_error = D_net_real_error + D_net_fake_error
      D_net_running_loss.append(float(D_net_total_error))
      D_net_optimizer.step()

      #Train Generator
      G_net.zero_grad()
      label.fill_(real)
      output = D_net(fake_images).view(-1)
      G_net_error = criterion(output, label)
      G_net_running_loss.append(float(G_net_error))
      G_net_error.backward()
      G_net_optimizer.step()

      if (i+1) % var == 0:
        average_D_net_loss = torch.mean(torch.FloatTensor(D_net_running_loss))
        average_G_net_loss = torch.mean(torch.FloatTensor(G_net_running_loss))
        full_D_net_loss.append(average_D_net_loss)
        full_G_net_loss.append(average_G_net_loss)
        print(”Discriminator loss: \n[epoch:%d, batch:%5d] loss: %.3f” %(epoch + 1, i + 1, average_D_net_loss / float(var))
        print(”Generator loss: \n[epoch:%d, batch:%5d] loss: %.3f” %(epoch + 1, i + 1, average_G_net_loss / float(var)))
        G_net_running_loss = []
        D_net_running_loss = []

        #Show the images being produced by the generator
        xform = transforms.Compose([transforms.Normalize((0,0,0), (2, 2, 2)),
                                    transforms.Normalize((-0.5, -0.5, -0.5,), (1,1,1)),
                                    transforms.ToPILImage()])
        gen_imgs = G_net(test_noise)

        fig, axs = plt.subplots(1, 5, figsize=(15,15))
        for k in range(5):
          image = xform(gen_imgs[k])
          axs[k].imshow(image)
          axs[k].set_axis_off()
        plt.show()
  torch.save(D_net, ’D_net.pth’)
  torch.save(G_net, ’G_net.pth’)

  return full_D_net_loss, full_G_net_loss


##############################
# Main Loop
##############################

if __name__ == ’__main__’:

  data_path = ’hw07_data/Train/Data/Train/’
  data_path2 = ’hw07_data/Train/Data/Test/’

  #Create transform for dataloader
  xform = transforms.Compose([transforms.ToTensor(), \
                              transforms.Normalize((0.5, 0.5, 0.5), \
                                                   (0.5, 0.5, 0.5))])

  dataset = BF_Dataset(data_path, data_path2, xform)
  train_data_loader = torch.utils.data.DataLoader(dataset=dataset, batch_size=10, shuffle=True, num_workers=2, worker_init_
  network3 = TemplateNet(num_conv_layers=2, padding=1)
  network_Gen = Generator_Net()
  [full_D_net_loss, full_G_net_loss] = run_code_for_training(network3, network_Gen, 10, 1)

  plt.figure()
  plt.plot(full_D_net_loss, label = ’Discriminator Loss’)
  plt.plot(full_G_net_loss, label = ’Generator Loss’)
  plt.legend()
  plt.xlabel(’Iterations / 500’)
  plt.ylabel(’Training (BCE) Loss’)
  plt.savefig(’training_loss.jpg’)
  plt.show()

  plt.figure()
  plt.plot(full_D_net_loss, label = ’Discriminator Loss’)
  plt.legend()
  plt.xlabel(’Iterations / 500’)
  plt.ylabel(’Training (BCE) Loss’)
  plt.savefig(’D_loss.jpg’)
  plt.show()

  plt.figure()
  plt.plot(full_G_net_loss, label = 'Generator Loss')
  plt.legend()
  plt.xlabel('Iterations / 500')
  plt.ylabel('Training (BCE) Loss')
  plt.savefig('G_loss.jpg')
  plt.show()


  #Validation Section
  cpu = torch.device("cuda:0")
  new_test_noise = torch.randn(10, 100, 1, 1, device=cpu)
  Gen_Net = torch.load('G_net.pth')
  xform2 = transforms.Compose([transforms.Normalize((0,0,0), (2, 2, 2)),
                                    transforms.Normalize((-0.5, -0.5, -0.5,), (1,1,1)),
                                    transforms.ToPILImage()])

  gen_imgs = Gen_Net(new_test_noise)
  fig, axs = plt.subplots(1, 5, figsize=(15,15))
  for k in range(5):
    image = xform2(gen_imgs[k])
    axs[k].imshow(image)
    axs[k].set_axis_off()
  plt.savefig('Validation_Imgs1.jpg')
  plt.show()

  fig2, axs2 = plt.subplots(1, 5, figsize=(15,15))
  for k in range(5):
    image = xform2(gen_imgs[k+5])
    axs2[k].imshow(image)
    axs2[k].set_axis_off()
  plt.savefig('Validation_Imgs2.jpg')
  plt.show()
```

The resulting total loss computed via running this script, for both the discriminator and generator networks, is shown in Figure 3.

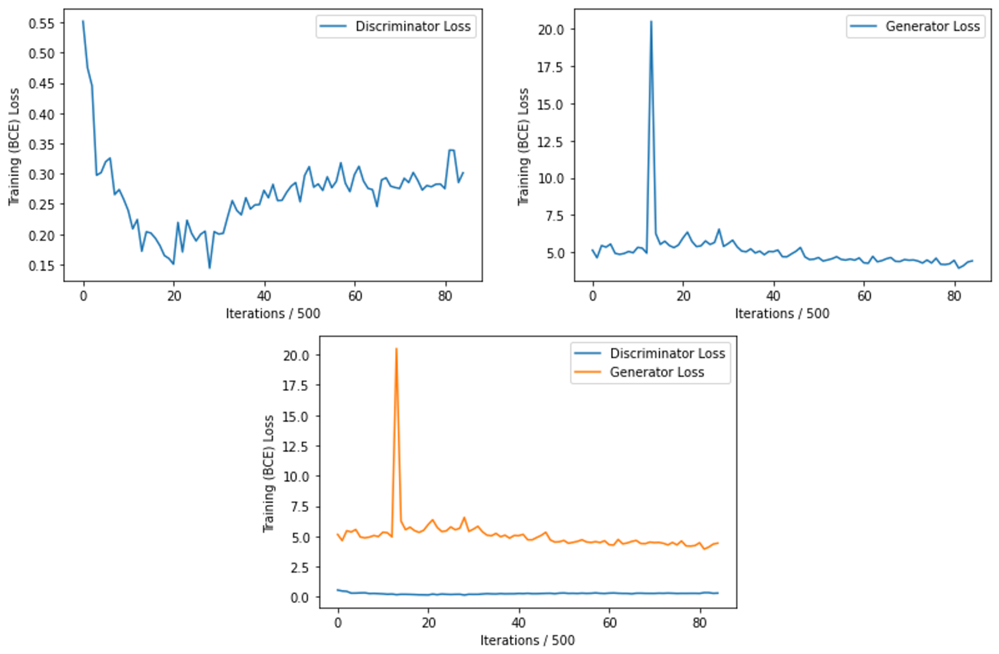

Figure 3: Training loss of discriminator and generator networks over five epochs of training. Loss was computed using BCE loss.

The final testing images produced at the end of training are shown in Figure 4. Though not perfect, the results show promising performance in that the images do reflect many realistic features.

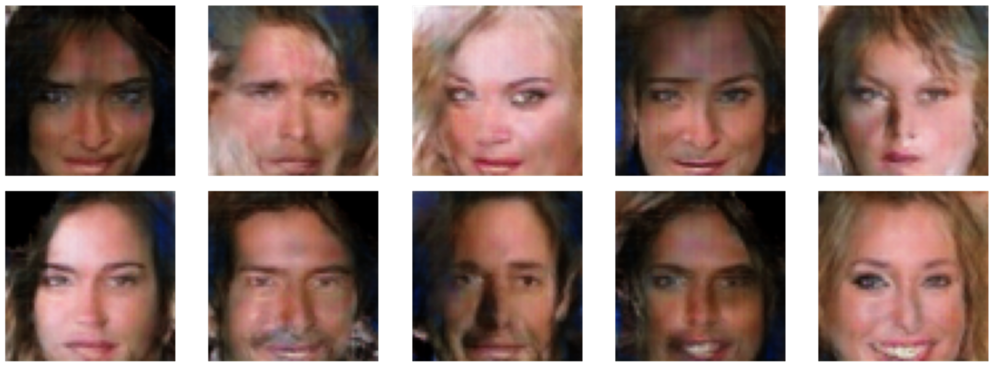

Figure 4: Examples of final performance of face image generation by the Generator network.

4 Lessons Learned

One of the main lessons I learned while working on this assignment was how deep neural network training can lead to getting "stuck" in possibly sub-optimal performance. While testing out my code, I found several times that the training would start out in a promising way, and the generator network would begin to produce improving results (that were starting to look like faces), but at a certain point the generator would take a sudden turn and begin outputting very poor images that did not resemble faces at all. Sometimes the generator would always output strange patterns, or just output a mostly black image. While I'm still not certain of the cause of this, I hypothesized that the generator might be locking onto some strange image type which is able to "trick" the discriminator while also not resembling true faces. This may have been causing issues with a then vanishing gradient. I had to work this out by testing different methods of gradient step optimization.

5 Suggested Enhancements

I do not have any major suggestions for improving this homework assignment. I found the assignment to be a bit easier than the last few assignments, but this was a very welcome surprise at this stage in the semester. I feel that the assignment is good because it allows us to see realistic and promising performance of GANs with a relatively simple network architectures. My only minor suggestion would be to clarify the testing expectations and better present these within the assignment document.

References

[1] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.