Assignment 4

IEORE 4742 Deep Learning for OR and FE

Ahmad Shayaan[1]
as5948

{ahmad.shayaan@columbia.edu}@columbia.edu
Columbia University
November 25, 2019

[1]All authors contributed equally to this work.

Question 1

Part 1

In this question code we had to train the GAN with the entire MNIST training dataset and see the output. The Tensorflow MNIST dataset contains 55000 images in the training dataset. The images generate by the generator are shown below.

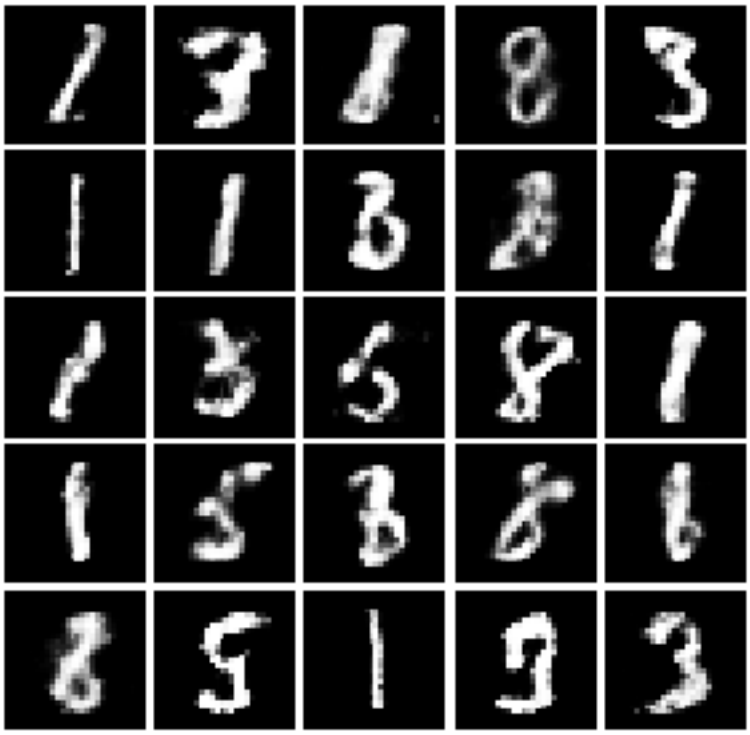

Figure 1: Images generated using the entire training set

From figure 1 we can see that the GAN is able to capture most of the details in the training data and is able to generate a mapping between the distribution of the MNIST dataset to the distribution of the input noise.

Part 2

In this question we had to train the Generative Adversarial Network with only the first 5000 images in the MNIST training dataset. The function to generate the batches is shown below.

```
def next_batch(data, batchSize):
  idx = np.arange(0 , len(data))
  np.random.shuffle(idx)
  idx = idx[:batchSize]
  data_shuffle = np.array([data[i] for i in idx])
  return data_shuffle
```

In input the function are the first 5000 images of the MNIST training dataset. The images generated by the generator after the training is completed are shown below.

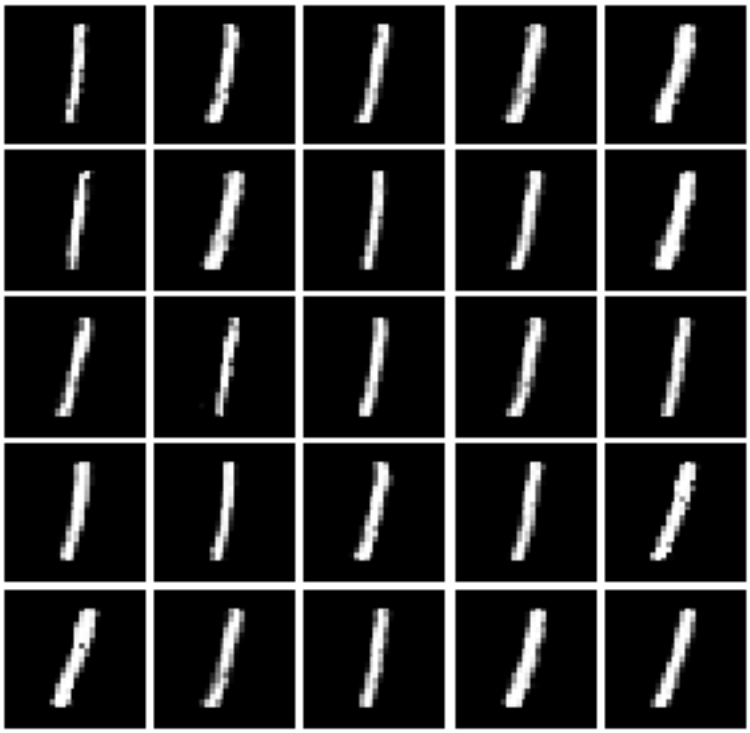

Figure 2: Images generated using first 5000 images

From figure 2 we can see that the GAN suffers from mode collapse and generates only a limited type of samples. This is because when we use only the first 5000 images the gradients of the discriminator fail to make the generator adequately learn the true distribution.

Part 3

In this question we had to use an autoencoder to generate additional training samples from the first 5000 images of the MNIST dataset. I trained autoencoders with three layers, two layers and one layer in the encoder and decoder. The images generated by the autoencoders after training are shown below.

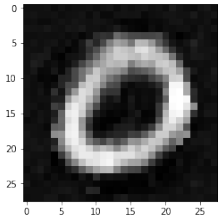

Figure 3: Image generate by the autoencoder with one layer

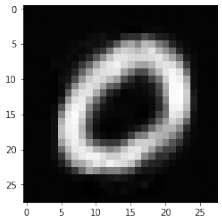

Figure 4: Image generate by the autoencoder with two layers

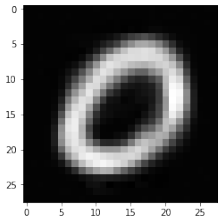

Figure 5: Image generate by the autoencoder with three layer

It can observed from the figure that as we increase the number of layers in the autoencoders we are able to capture more information and are able to generate better samples.

We train three separate GANs with the first 5000 images of the MNIST training set and 5000 images generated from the different auto encoders . The images generated by the generator after the training is complete are shown below.

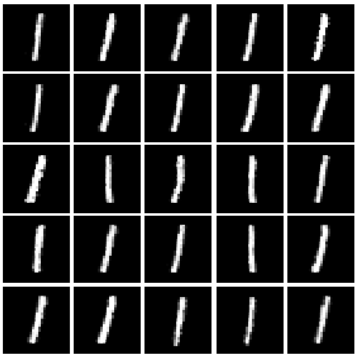

Figure 6: Images generated using samples from the autoencoder with a single layer

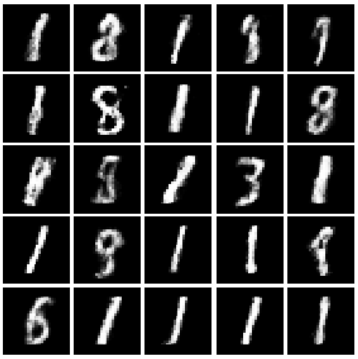

Figure 7: Images generated using samples from the autoencoder with two layers

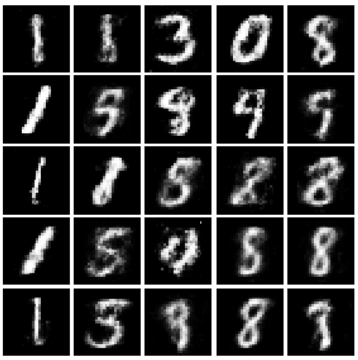

Figure 8: Images generated using samples from the autoencoder with three layers

Figure 6 we can see that the generator suffers from mode collapse and is not able to learn sufficiently from the gradient of the discriminator. The results are better in the case when we use an autoencoder with two layers as can be seen from figure 7
We can observe from figure 8 that the generator is able to capture details about the MNIST dataset and is able to create a mapping between the distribution of the MNIST dataset to

the distribution of the input noise when we use samples generate by an autoencoder with three layers. We can thus conclude that Autoencoders with sufficient number of layers can be used to generate additional data points.

Part 4

In this question we had augment the first 5000 images of the MNIST data set by adding noise to them and then train the GAN on these images. We generated new images by adding Gaussian noise, Salt & Pepper noise and Poisson noise.

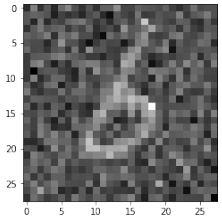

Figure 9: Image with salt and pepper noise

Figure 9 shows and image with salt and pepper noise. We then combined the first 5000 images of MNIST with the images created by adding noise to create a dataset with 20000 images and then used them to train the GAN. The images generated by the generator after training is complete are shown below.

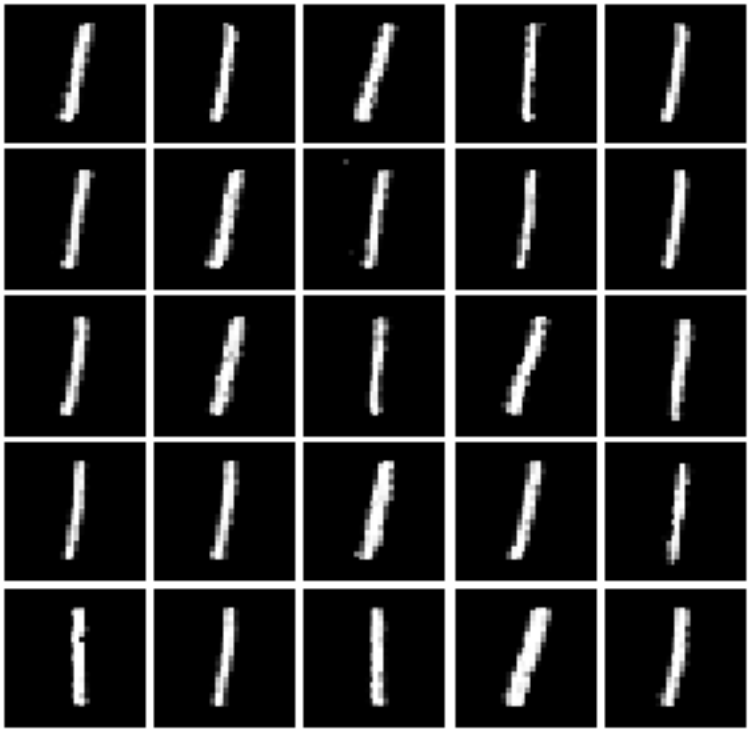

Figure 10: The images generated using noisy samples

From figure 10 we can see that the GAN suffers from mode collapse and generates only a limited type of samples. The noisy images do not contain enough distinct information so that the gradient of the discriminator can make the generator learn the true distribution. We can thus conclude that adding noise to images is not a good way to augment data points.

Question 2

We had to build an architecture for a 5 layer deep convolutional GAN to generate images whose sizes are 512×512. In our architecture the discriminator is made up of convolutional layers with batch normalization and leaky ReLU as the activation function. The input to the discriminator is an image of dimension $3 \times 512 \times 512$ and the output is the probability that the image belongs to the original data.

The generator is made of convolutional transpose layer with batch normalization and ReLU activation function. The input is a noise that is sampled from a standard normal and the output are images of dimensions $3 \times 512 \times 512$.

We use the binary cross entropy loss to train the model. We define two types of labels, 1 for real data and 0 for generated data. The loss of the generator comes from the images that were correctly classified by the discriminator to be fake. Whereas the loss of the discriminator comes from the images that were classified to be real but were generated by the generator. The code for the generator and discriminator is show below.

```
batch_size = 1
nz = 100
nc= 3
ngf = 512
ndf = 64
lr = 0.0002

class Generator(nn.Module):
def __init__(self):
  super(Generator,self).__init__()
  self.generator = nn.Sequential(

    nn.ConvTranspose2d( nz, ngf * 8, 8, 1, 0, bias=False),
    nn.BatchNorm2d(ngf * 8),
    nn.ReLU(True),

    nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 4, 0, bias=False),
    nn.BatchNorm2d(ngf * 4),
    nn.ReLU(True),

    nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 4, 0, bias=False),
```

```
        nn.BatchNorm2d(ngf * 2),
        nn.ReLU(True),

        nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
        nn.BatchNorm2d(ngf),
        nn.ReLU(True),

        nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
        nn.Tanh()
        )

    def forward(self,data):
      return self.generator(data)

    class Discriminator(nn.Module):
    def __init__(self):
      super(Discriminator,self).__init__()
      self.discriminator = nn.Sequential(

        nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
        nn.LeakyReLU(0.2, inplace=True),

        nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
        nn.BatchNorm2d(ndf * 2),
        nn.LeakyReLU(0.2, inplace=True),

        nn.Conv2d(ndf * 2, ndf * 4, 4, 4, 0, bias=False),
        nn.BatchNorm2d(ndf * 4),
        nn.LeakyReLU(0.2, inplace=True),

        nn.Conv2d(ndf * 4, ndf * 8, 4, 4, 0, bias=False),
        nn.BatchNorm2d(ndf * 8),
        nn.LeakyReLU(0.2, inplace=True),

        nn.Conv2d(ndf * 8, 1, 8, 1, 0, bias=False),
        nn.Sigmoid()
        )

    def forward(self,data):
      return self.discriminator(data)
```

The kernel size, padding and the stride size were computed using the following formulas.

$$H_{out} = (H_{in} - 1) \times stride - 2 \times padding + kernel\_height \tag{1}$$

$$W_{out} = (W_{in} - 1) \times stride - 2 \times padding + kernel\_width \tag{2}$$

Figure 11 shows all the parameters of the discriminator and the generator.

```
(deep) shayaan@shayaan-ThinkPad-E470:~/Columbia/sem_1/Deep-Learning-for-OR-and-FE/assign4$ python question4.py
Generator(
  (generator): Sequential(
    (0): ConvTranspose2d(100, 4096, kernel_size=(8, 8), stride=(1, 1), bias=False)
    (1): BatchNorm2d(4096, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(4096, 2048, kernel_size=(4, 4), stride=(4, 4), bias=False)
    (4): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(2048, 1024, kernel_size=(4, 4), stride=(4, 4), bias=False)
    (7): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(1024, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(512, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
Discriminator(
  (discriminator): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(4, 4), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(4, 4), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(8, 8), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

Figure 11: Generator and Discriminator definitions

The code that was used to train the model is given below.

```
def trainingGAN(data_generator,data_discriminator):
    criterion = nn.BCELoss()
    real_label = 1
    fake_label = 0

    gen = Generator()
    dis = Discriminator()

    print (gen)
    print (dsi)

    optimizerD = optim.Adam(dis.parameters(),lr=lr)
    optimizerG = optim.Adam(gen.parameters(),lr=lr)

```

```
16    for epoch in range(1):

18      dis.zero_grad()
        label_real = torch.tensor(real_label,dtype=torch.float32)
20
        output = dis.forward(data_discriminator)
22      errD_real = criterion(output, label_real)
        errD_real.backward()
24      D_x = output.mean().item()

26      fake_data = gen.forward(data_generator)
        label_fake = torch.tensor(fake_label,dtype=torch.float32)
28
        output = dis.forward(fake_data.detach()).view(-1)
30      errD_fake = criterion(output, label_fake)
        errD_fake.backward()
32
        D_G_z1 = output.mean().item()
34      errD = errD_real + errD_fake
        optimizerD.step()
36
        gen.zero_grad()
38      output = dis.forward(fake_data).view(-1)
        errG = criterion(output, label_real)
40      errG.backward()
        D_G_z2 = output.mean().item()
42      optimizerG.step()


44
        print ('Loss_D: %.4f Loss_G: %.4f'%(errD,errG))
```