STAT3007 Tutorial Paper:
Generative Adversarial Networks

Chun-Chiao Huang 46395281

The University of Queensland

May, 2022

I give consent for this to be used as a teaching resource.

Contents

1 Introduction to Generative Adversarial Network

1.1 Overview

Consider building a deep learning model for image classification task, we often need to do data augmentation such as flipping, cropping, rotation and etc., to increase model's performance. Generative adversarial networks(GAN) could be applied to do data augmentation by learning from available samples and generating more samples.

A GAN consists of two networks: a generator that aims to approximate the data distribution and a discriminator that aims to distinguish either a sample is from the data distribution or from the approximated, by the generator, distribution. The two networks are trained and improved simultaneously, until the approximated distribution is close enough to the data distribution. In another word, the two networks are competing with each other; the generator is trying to synthesize samples that look similar to the input data and cheat the discriminator while the discriminator is trying to tell where the samples come from and not to be cheated. The competition ends until the discriminator cannot tell the difference of samples from the data distribution and the approximated distribution.

1.2 The Discriminator

Given a sample, the D needs to predict the probability that it is from the data distribution p_{Data}. Namely, it maximizes $E_{x \sim p_{Data}}[log(D(x))]$ and $E_{z \sim p_Z}[log(1 - D(G(z))]$. That is, predict a high value if a sample is from p_{Data} and a low value if it is from p_G.

$$Objective: \ Maximize \, \mathbb{E}_{\mathbf{x}} \sim \mathbf{p_{Data}}[log(D(x))] + \mathbb{E}_z \sim p_Z[log(1 - D(G(z))] \tag{1}$$

1.3 The Generator

Given $z \sim p_Z$, the G synthesis a sample $G(z)$ that can cheat the D. Namely, it minimizes the $\mathbb{E}_z \sim p_Z[log(1 - D(G(z))]$. That is, it aims to convince the D that $G(z)$ is from p_{Data}.

$$Objective: \ Minimize \, \mathbb{E}_{\mathbf{z}} \sim \mathbf{p_Z}[log(1 - D(G(z))] \tag{2}$$

Exercise 1: Why does G minimize $\mathbb{E}_{\mathbf{z}} \sim \mathbf{p_Z}[log(1 - D(G(z))]$ instead of maximize $\mathbb{E}_{\mathbf{z}} \sim \mathbf{p_Z}[log(D(G(z))]$?

2 PyTorch Implementation

In this section, we implement a Generative Adversarial Network using PyTorch and follow a overall structure from this tutorial https://pytorch.org/tutorials/beginner/dcgan_faces

_tutorial.html. Firstly, we introduce the whole model architecture and the dataset we will be using. Secondly, we construct the Generator and the Discriminator.Thirdly, we implement the loss function. Fourthly, we implement the training loop, and finally we will demonstrate the training results.

2.1 Network Architecture

The figure below shows the structure of a GAN. Given a vector z sampled from normal distribution, the Generator synthesizes a fake image. The Discriminator learns to differentiate between the real images and the synthetic images. In this tutorial paper, we use the MNIST dataset, which contains number digits from 0 to 9 and 60000 samples in total. However, in order to avoid our model from generating unrealistic digits, we use only images of one digit.
Note: The idea of using only images of one digit is not from the tutorial.
I got this idea from my own experience of training GAN for data augmentation.

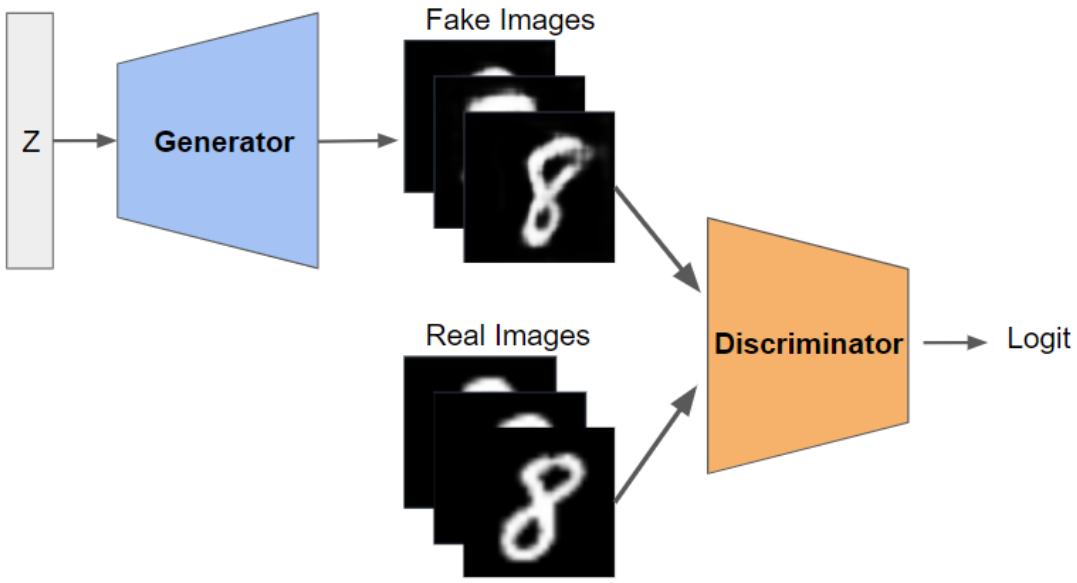

Figure 1: An illustration of the overall structure of a GAN

Exercise 2: Why do we use only images from one digits to avoid generating unrealistic digits? What can the result be if we use images of all digits?

2.2 Network Implementation

2.2.1 The Generator

The code below shows the Generator network. Firstly, we reshape the input vector to 4-dimension so it can be fed to the model. Secondly, we upsample each pixel to a square of

pixels. Thirdly, we pass them through a convolution layer but remain the sizes the same. By repeating the second and the third steps with different sizes for interpolation and numbers for feature maps, we can finally get a feature map of size 64*64. Before it is outputted, we pass it through a tanh function so its range is limited within[-1, 1]. Each of the convolution layer was followed by a batch-normalization layer and a relu activation function except for the last one.

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator,self).__init__()
        self.conv_proj = nn.Conv2d(100, 512, kernel_size=3, padding=1)
        self.bn_proj = nn.BatchNorm2d(512)
        self.conv1 = nn.Conv2d(512, 256, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(256)
        self.conv2 = nn.Conv2d(256, 128, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 64, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(64)
        self.conv4 = nn.Conv2d(64, 1, kernel_size=3, padding=1)

    def forward(self,x):
        x=x.reshape(x.shape[0], x.shape[1], 1, 1) # b* 100* 1* 1
        x=F.interpolate(x,[4,4]) #batch_size* 100* 4* 4
        x=F.relu(self.bn_proj(self.conv_proj(x)))
        x=F.interpolate(x,[8,8]) #batch_size* 512* 8* 8
        x=F.relu(self.bn1(self.conv1(x)))
        x=F.interpolate(x,[16,16]) #batch_size* 256* 16* 16
        x=F.relu(self.bn2(self.conv2(x)))
        x=F.interpolate(x,[32,32]) #batch_size* 128* 32* 32
        x=F.relu(self.bn3(self.conv3(x)))
        x=F.interpolate(x,[64,64]) #batch_size* 64* 64* 64
        x=self.conv4(x) #batch_size* 1* 64* 64
        out=torch.tanh(x)
        return out
```

Note: Instead of using transpose convolution, I use interpolation followed by a convolution layer. This idea is not from the tutorial or the paper. I was inspired by a tutor from another course.

2.2.2 The Discriminator

The code below show the Discriminator Network. The Discriminator is actually a binary CNN classifier, but without any pooling layer. Instead, we do the downsampling with convolution layers as this allows the model to learn its own pooling function. By setting the kernel size, stride and padding size according to the formula $OutSize = (InSize - k + 2p)/s + 1$, we reduce

the size of feature maps by 2 in width and height in each convolution layer. As for the last classification layer, we replace the usual fully-connected layer by a convolution layer. Before the result is outputted, we pass it through a sigmoid layer to limit its value within [0, 1], as it is a probability.

```
class Discriminator(nn.Module):
    def __init__(self,n_features=512,n_channels=1):
        super(Discriminator,self).__init__()
        self.conv1 = nn.Conv2d(1, 64, kernel_size=4, stride=2, padding=1)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1)
        self.bn3 = nn.BatchNorm2d(256)
        self.conv4 = nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1)
        self.bn4 = nn.BatchNorm2d(512)
        self.classifier = nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=0)
        self.sigm=nn.Sigmoid()
    def forward(self,x):
        #initial x: batch_size* 1* 64* 64
        x = F.leaky_relu(self.conv1(x),0.2) #batch_size* 64* 32*32
        x = F.leaky_relu(self.bn2(self.conv2(x)),0.2) #batch_size* 128* 16*16
        x = F.leaky_relu(self.bn3(self.conv3(x)),0.2) #batch_size* 256* 8*8
        x = F.leaky_relu(self.bn4(self.conv4(x)),0.2) #batch_size* 512* 4*4
        out = self.sigm(self.classifier(x)) #out: batch_size* 1* 1*1
    return out
```

Exercise 3: Given a image of size 64*64, what would the output size be if we pass it through a convolution layer with kernel size 5, stride 3 and padding 2? Give an example of kernel size, stride and padding that remains the input size and the output size the same.

2.3 Loss function

We use the BCELoss from PyTorch to calculate the log components by setting the value of the labels. The definition of BCELoss is $loss(x, y) = -[y * logx + (1 - y) * log(1 - x)]$, where x is the predicted value and y is the target.

2.3.1 Discriminator Loss

The code below show the loss function of the Discriminator. As aforementioned, the Discriminator aims to Maximize $\mathbb{E}_{x\sim p_{Data}}[log(D(x))] + \mathbb{E}_{z\sim p_Z}[log(1 - D(G(z))]$. The first part is to handle the real images and the second part is for the fake images. For example, given $x \sim p_{Data}$ with

its label $y = 1$, the loss is $-[1 * log(D(x)) + (1 - 1) * log(1 - D(x))] = -log(D(x))$, which is equivalent to maximizing $log(D(x))$.

```
1  def D_loss(D, G, real_images, batch_size):
2      z=torch.randn([batch_size, 100],device=device)
3      fake_images=G(z)
4      real_labels=torch.ones(batch_size, device=device)
5      fake_labels=torch.zeros(batch_size, device=device)
6      criterion=nn.BCELoss()
7      #D_loss1: Maximize log(D(x)) -> Minimize -log(D(x))
8      #BCELoss: -[1*log(D(real)) + (1-1)*log(1-D(real))] -> -log(D(real))
9      D_loss1=criterion(D(real_images).squeeze(), real_labels)
10     #D_loss2: Maximize log(1-D(G(z))) -> Minimize -log(1-D(G(z)))
11     #BCELoss: -[0*log(D(fake)) + (1-0)*log(1-D(fake))] -> -log(1-D(fake))
12     D_loss2=criterion(D(fake_images).squeeze(), fake_labels)
13     return D_loss1+D_loss2
```

2.3.2 Generator Loss

The code below show the loss of the Generator. By using the same technique shown in the Discriminator loss section, the Generator can maximize $log(D(G(z)))$ given $z \sim p_Z$.

```
1  def G_loss(D, G, batch_size):
2      z=torch.randn([batch_size, 100],device=device)
3      fake_images=G(z)
4      real_labels=torch.ones(batch_size, device=device)
5      criterion=nn.BCELoss()
6      G_loss=criterion(D(fake_images).squeeze(), real_labels)
7      return G_loss
```

Exercise 4: Given $z \sim p_Z$, calculate the Generator loss using BCELoss. Is it equivalent to the Generator objective we mentioned before?

2.4 Training Loop

In this section, we demonstrate the training loop. We first create the two models and initialize them. Secondly, we set two Adam optimizer for the D and the G. The batch size for the dataloder is 64, however, the total number of the data may not be a multiple of 64. Therefore, we set the batch size in line 13 to avoid dimension problem. Lastly, we train the Discriminator and the Generator simultaneously for 10 epochs. Note that there is a $fixed\ z$, which remains the same during the whole training so we can use it to generate images and observe how they were

changing.

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
D = Discriminator().to(device)
G = Generator().to(device)
D.apply(weights_init),G.apply(weights_init)

optim_D = optim.Adam(D.parameters(), lr=0.0002, betas=(0.5,0.999))
optim_G = optim.Adam(G.parameters(), lr=0.0002, betas=(0.5,0.999))
dataloader = DataLoader(dataset, num_workers=0, batch_size=64, shuffle=True)
num_epochs = 10

fixed_z = torch.randn([16, 100],device=device)
for epoch in range(num_epochs):
    for i, (real_images, _) in enumerate(dataloader):
        batch_size = real_images.size(0)
        real_images = real_images.to(device)
        #train D
        optim_D.zero_grad()
        d_loss = D_loss(D, G, real_images, batch_size)
        d_loss.backward()
        optim_D.step()
        #train G
        optim_G.zero_grad()
        g_loss = G_loss(D, G, batch_size)
        g_loss.backward()
        optim_G.step()
```

2.5 Results

2.5.1 Training loss

From Figure 2, we can see that the losses of the two model. The loss of the Discriminator remained stable during the whole training process while that of the Generator fluctuated a lot. The lowest loss of the Generator appeared at epoch 5 and the highest was at epoch 9.

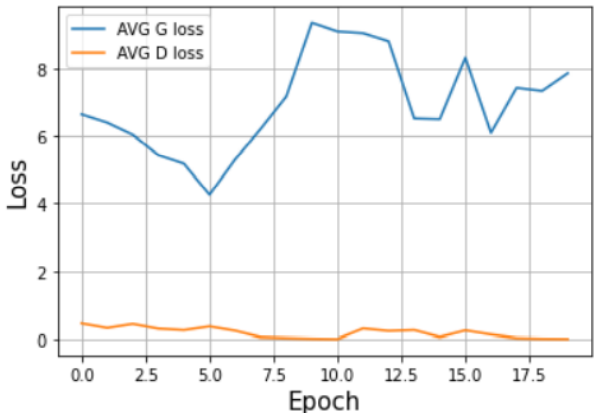

Figure 2: Loss plot of the two network

2.5.2 Generated images

From Figure 3 and Figure 4, we can see that the loss can indicate the performance of the Generator. However, this is not an accurate way to evaluate the Generator as a GAN contains two networks. Therefore, the losses are influenced by the performance of itself and the other.

Figure 3: Generated images at epoch 5

Figure 4: Generated images at epoch 9

3 Algorithm analyses

3.1 Strengths

Firstly, the realness of images is hard to define mathematically, while GANs define the realness by a neuron network. Secondly, GANs can produce better images than other models such as variational autoencoder. Lastly, no Markov chains is required during the training or the generating processes (Goodfellow et al., 2014).

3.2 Limitations

GANs are very hard to train. Firstly, since the losses of the two components, namely the Generator and the Discriminator, are not only influenced by the network itself, but also by the other part of the GAN, it is hard to evaluate the convergence of training by the usual metrics. Secondly, a vanilla GAN is unstable, it often collapses. From the Figure 4, we can see the images became weird because the model was collapsing. Lastly, as the Generator aims to cheat the Discriminator, therefore, sometimes the Generator may generate some very similar images with any input noise as those images may have higher chances to convince the Discriminator. In Figure 5, we can see those generated images are similar, but they are generated with different z.

4 Exercise Solutions

4.1 Exercise 1

Why does G minimize $\mathbb{E}_{\mathbf{z}} \sim \mathbf{p_Z}[log(1 - D(G(z))]$ instead of maximize $\mathbb{E}_{\mathbf{z}} \sim \mathbf{p_Z}[log(D(G(z))]$?

Higher $D(G(z))$ means the Discriminator is more convinced that $G(z)$ is from $pData$, so lower $1 - D(G(z))$ means the same. Therefore, minimizing $\mathbb{E}_{\mathbf{z}} \sim \mathbf{p_Z}[log(1 - D(G(z))]$ is equivalent to

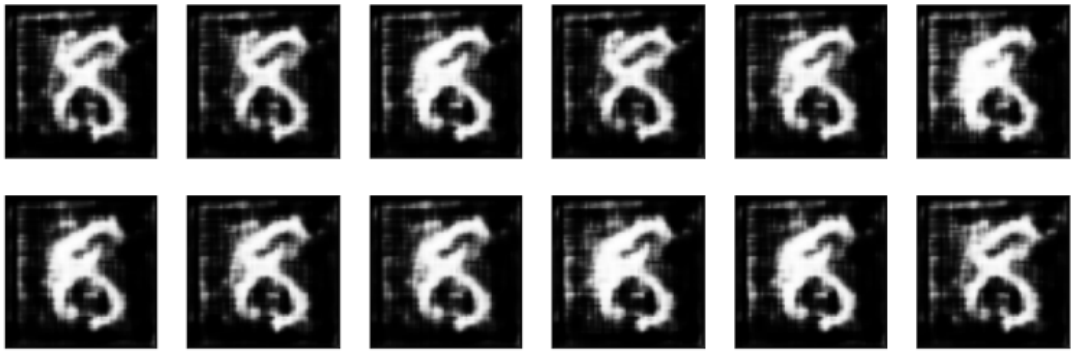

Figure 5: Generated images at epoch 19

maximizing $\mathbb{E}_{\mathbf{z}} \sim \mathbf{p_Z}[log(D(G(z))]$. In fact, $\mathbb{E}_{\mathbf{z}} \sim \mathbf{p_Z}[log(1 - D(G(z))]$ may not provide enough gradient for Generator during early training, therefore, one can train Generator by maximizing $\mathbb{E}_{\mathbf{z}} \sim \mathbf{p_Z}[log(D(G(z))]$ in practice.

4.2 Exercise 2

Why do we use only images from one digits to avoid generating unrealistic digits? What can the result be if we use images of all digits?

We can see from the Figure 1 that there is no labels of the dataset is provided during training, the only labels involved are 1 for real images and 0 for fake images. Therefore, the GAN does not know that there are 10 types of digits. For example, if we train the model use all data, the Generator may generate something look like an eight but also like a zero. We can solve this problem by implement a conditional GAN. For more information about conditional GAN, see https://learnopencv.com/conditional-gan-cgan-in-pytorch-and-tensorflow/ Figue 6 shows an example of unrealistic digit.

4.3 Exercise 3

Given a image of size 64*64, what would the output size be if we pass it through a convolution layer with kernel size 5, stride 3 and padding 2? Give an example of kernel size, stride and padding that remains the input size and the output size the same.

According to the formula $OutSize = (InSize - k + 2p)/s + 1$,

$$\begin{aligned} OutSize &= (64 - 5 + 4)/3 + 1 \\ &= 22 \end{aligned} \tag{3}$$

example 1: [kernel size, stride, padding] = [3, 1, 1]
example 2: [kernel size, stride, padding] = [5, 1, 2]

Figure 6: An example of unrealistic digit

4.4 Exercise 4

Given $z \sim p_Z$, calculate the Generator loss using BCELoss. Is it equivalent to the Generator objective we mentioned before?

Given $z \sim p_Z$, the generated sample is $G(z)$ and the goal of the Generator is to convince the Discriminator that $G(z)$ is a real image. Therefore, the label y is 1. According to the definition of BCELoss, $loss(x, y) = -[y * logx + (1 - y) * log(1 - x)]$.

$$\begin{aligned} loss(D(G(z), 1) &= -[1 * log(D(G(z))) + (1 - 1) * log(1 - D(G(Z)))] \\ &= -log(D(G(z))) \end{aligned} \tag{4}$$

Maximizing $log(D(G(z)))$ is equivalent to minimizing $-log(D(G(z)))$. Therefore, yes it is equivalent to the Generator objective.

5 References

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., … Bengio, Y. (2014). Generative adversarial nets.
https://arxiv.org/pdf/1406.2661.pdf
Gao, F., Yang, Y., Wang, J., Sun, J., Yang, E., Zhou, H. (2018). A deep convolutional generative adversarial networks (DCGANs)-based semi-supervised method for object recognition in synthetic aperture radar (SAR) images.
https://arxiv.org/pdf/1511.06434.pdf
DCGAN Tutorial:
https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html