HW 7: GAN

David E. Farache, Email ID: dfarache@purdue.edu

April 6, 2023

1 Introduction

The goal of this homework is to create a pizza-generative adversarial network (GAN). In this case, we experiment with both Deep Convolutions GAN (DCGAN) and a Wasserstein GAN (WGAN) which use different approximation methods to calculate the difference between images and reduce those to create proper fake images.

2 Explanation of Networks

2.1 GAN

An adversarial network refers to a method whereby a generator is utilized to create fake images that are then evaluated by a discriminator, with the goal being that the discriminator is no longer capable of distinguishing between real and fake images. In this case, we assume that each image is a probability distribution $\rho_x(x)$, where x is the image, and that each generated image, G(z, θ) where z is the noise vector and θ are the parameters, is a probability of $\rho_\theta(z)$. The goal is for the difference between distributions to be near zero, meaning that the probability of the image being true or false is about equivalent.

The generator is trained to minimize the ability of the discriminator to differentiate between the real and fake images:

$$\theta = \min_{\theta} E[log(1 - D(G(z)))] \tag{1}$$

The discriminator is trained by maximizing the probability of giving the proper label to the image:

$$\theta = \max_{\theta} E[log(D(x)] \tag{2}$$

This creates a minmax equation for the GAN network as seen below:

$$\min_{\theta_g} \max_{\theta_d} [E[log(1 - D(G(z))) + E[log(D(x)]] \tag{3}$$

2.2 DCGAN

The deep convolution GAN, is a network that uses convolutional layers within both the generator and the discriminator. In the case the DCGAN is trained using Binary Cross-Entropy (BCE) loss with the prediction being a binary label of the real or fake image. This equation for the loss can be seen below:

$$BCE = -(y \log(p) + (1 - y) \log(1 - p)) \tag{4}$$

2.3 WGAN

Wasserstein GAN utilizes a different method from the previously mentioned networks, as it does not use a discriminator but a critic, which differs by the objective not to evaluate the generator results via a binary classification but instead calculate the Wasserstein distance between training and generated data distribution. Wasserstein distance is based on the marginal distribution. This is done via the equation below:

$$dw(P_r, P_\theta) = \sup_{||f||_L \leq 1} [E(f_w(x)) - E(f_w(g_\theta(z)))] \tag{5}$$

In this equation marginals of the real data to the generated data are compared for all 1-Lipschitz Functions described by:

$$|f(x_1) - f(x_2)| \leq 1 * d(x_1, x_2) \forall x_1, x_2 \in X \tag{6}$$

The designation of the WGAN is to reduce the Wasserstein distance between the distribution of the generated set and the real set. While doing so, critic C attempts to maximize the distance to increase the diversity of the images capable of being generated so as to not simply be a copier. This again leads to a min-max problem for the framework as described below:

$$\min_g \max_c [E[C(x)] - E[C(G(z))]] \tag{7}$$

This was further improved via the implementation of a gradient penalty, introducing a method to find optimal critic C by minimizing critic loss as seen below:

$$CriticLoss = E[C(G(z))] - E[C(x)] + [||\nabla_{\hat{x}} C(\hat{x})||^2 1]^2 \tag{8}$$

3 Task 1: DCGAN

3.1 Setup Code

```python
# %%
# Libraries
import numpy as np
import torch
import torchvision.transforms as tvt
import torch.utils.data
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
from PIL import Image
import os
from pprint import pprint
from torchinfo import summary
import torchvision.datasets
import time
import datetime
from pytorch_fid.fid_score import calculate_activation_statistics,
    calculate_frechet_distance
from pytorch_fid.inception import InceptionV3
from torchvision.utils import save_image
```

```python

device = 'cuda'
device = torch.device(device)
root_dir = "/scratch/gilbreth/dfarache/ece60146/David/HW7/"
train_data_path = root_dir + "pizza_train"
test_data_path = root_dir + "pizza_eval"

# Create Data Loader
def createDataLoader(root, batch_size, image_shape):
    transform = tvt.Compose([tvt.Resize(image_shape),
                             tvt.CenterCrop(image_shape),
                             tvt.ToTensor(),
                             tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
    data_set = torchvision.datasets.ImageFolder(root, transform=transform)

    DataLoader = torch.utils.data.DataLoader(data_set, batch_size=batch_size,
   num_workers=2, shuffle=True, drop_last=True)
    return DataLoader
```

Listing 1: *Setup Code*

3.2 DCGAN Network

```python
# Discriminator
# Based on DLStudio Discriminator-Generator DG1
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        # Conv Layers
        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)
        self.conv_in5 = nn.Conv2d( 512, 1024, kernel_size=4, stride=2, padding
   =1)
        self.conv_in6 = nn.Conv2d( 1024, 1, kernel_size=4, stride=1, padding=1)

        # Batch Layers
        self.bn1 = nn.BatchNorm2d(128)
        self.bn2 = nn.BatchNorm2d(256)
        self.bn3 = nn.BatchNorm2d(512)
        self.bn4 = nn.BatchNorm2d(1024)

        # Sig
        self.sig = nn.Sigmoid()

    def forward(self, x):
        x = torch.nn.functional.leaky_relu(self.conv_in(x), negative_slope=0.2,
   inplace=True)
        x = self.bn1(self.conv_in2(x))

        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.bn2(self.conv_in3(x))

        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.bn3(self.conv_in4(x))
```

```python

        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.bn4(self.conv_in5(x))

        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.conv_in6(x)

        x = self.sig(x)
        return x

# Generator
# Based on slides in class
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        # Conv Layers
        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)

        # Batch Layers
        self.bn1 = nn.BatchNorm2d(512)
        self.bn2 = nn.BatchNorm2d(256)
        self.bn3 = nn.BatchNorm2d(128)
        self.bn4 = nn.BatchNorm2d(64)

        # Tanh
        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
```

Listing 2: *DCGAN generator and discriminator*

3.3 DCGAN Training

```python
# Training DCGAN
def weights_init(m):
    """
    From the DCGAN paper, the authors specify that all model weights shall be
    randomly initialized from a Normal distribution with mean=0, stdev=0.02.
    The weights_init function takes an initialized model as input and reinitializes
    all convolutional, convolutional-transpose, and batch normalization layers to
    meet this criteria. This function is applied to the models immediately after
    initialization.

    https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html
    """

    classname = m.__class__.__name__
    if(classname.find('Conv') != -1): # If Conv not found in the classname
        nn.init.normal_(m.weight.data, mean=0.0, std=0.02)
    elif(classname.find('BatchNorm') != -1): # If BatchNorm not found in the classname
        nn.init.normal_(m.weight.data, mean=1.0, std=0.02)
        nn.init.constant_(m.bias.data, val=0)

# Based on lecture slides
def TrainDCGAN(netD, netG, epochs, betas, lr, trainDataLoader):
    # Number of channcel for noise vector
    nz = 100

    # Optimizer to device
    netD = netD.to(device)
    netG = netG.to(device)

    # Apply weight
    netD.apply(weights_init)
    netG.apply(weights_init)

    # We will use the same noise batch to periodically check on the progress made for the Generator:
    fixed_noise = torch.randn(batch_size, nz, 1, 1, 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=lr, betas=betas) # Adam Optimizer for Discriminator
    optimizerG = torch.optim.Adam(netG.parameters(), lr=lr, betas=betas) # Adam Optimizer for Generator

    # Criterion BCE
    criterion = nn.BCELoss()

    # Lists for training data
    img_list = []
```

```
    G_losses = []
    D_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 i, data in enumerate(trainDataLoader, 0):

            # Get Real Images
            netD.zero_grad()
            real_images_in_batch = data[0].to(device)

            # Train Discriminator on real images
            label = torch.full((real_images_in_batch.size(0),), real_label, dtype=torch.float, device=device)
            output = netD(real_images_in_batch).view(-1)

            real_image_Dlosses = criterion(output, label)
            real_image_Dlosses.backward()

            # Train Discriminator on fakes
            noise = torch.randn(real_images_in_batch.size(0), nz, 1, 1, device=device)
            fakes = netG(noise) # Create fakes
            label.fill_(fake_label) # Fill label with fakes

            output = netD(fakes.detach()).view(-1) # Get outputs of discriminator

            fake_image_Dlosses = criterion(output, label)
            fake_image_Dlosses.backward()
            total_Dlosses = real_image_Dlosses + fake_image_Dlosses
            d_losses_per_print_cycle.append(total_Dlosses)

            optimizerD.step() # Only the Discriminator weights are incremented

            # Minimize 1 - D(G(z)) by maximize D(G(z)) with generator of target value 1
            netG.zero_grad()

            label.fill_(real_label)
            output = netD(fakes).view(-1)

            total_Glosses = criterion(output, label)
            g_losses_per_print_cycle.append(total_Glosses)

            total_Glosses.backward()
            optimizerG.step()

            if i % 100 == 99:
                os.makedirs(root_dir + "./model", exist_ok = True)

```

```python
                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("[epoch=%d/%d iter=%4d elapsed_time=%5d secs] mean_D_loss=%7.4f mean_G_loss=%7.4f" %
                ((epoch+1),epochs,(i+1),time.time(),mean_D_loss,mean_G_loss))

                d_losses_per_print_cycle = []
                g_losses_per_print_cycle = []

                torch.save(netG.state_dict(), root_dir + "./model/DCGAN_gen.pt")
                torch.save(netD.state_dict(), root_dir + "./model/DCGAN_disc.pt")

                with torch.no_grad():
                    fake = netG(fixed_noise).detach().cpu()
                img_list.append(torchvision.utils.make_grid(fake, padding=1, pad_value=1, nrow=4, normalize=True))

            # Get All Loses
            G_losses.append(total_Glosses.item())
            D_losses.append(total_Dlosses.item())

    print("Traing Time %s sec" % (time.time() - start_time))
    return G_losses, D_losses, img_list
```

Listing 3: *DCGAN training*

4 Task 2: WGAN

4.1 WGAN Network

For WGAN the same generator was used as that in DCGAN

```python
# WGAN
# Based on DLStudio Critic-Generator CG2
class Critic(nn.Module):
    def __init__(self):
        super(Critic, self).__init__()
        self.DIM = 64
        self.net = nn.Sequential(
            nn.Conv2d(3, self.DIM, kernel_size=5, stride=2, padding=2),
            nn.ReLU(True),
            nn.Conv2d(self.DIM, 2*self.DIM, kernel_size=5, stride=2, padding=2),
            nn.ReLU(True),
            nn.Conv2d(2*self.DIM, 4*self.DIM, kernel_size=5, stride=2, padding=2),
            nn.ReLU(True),
            nn.Conv2d(4*self.DIM, 8*self.DIM, kernel_size=5, stride=2, padding=2),
            nn.ReLU(True),
        )
        self.output = nn.Linear(4*4*4*self.DIM, 1)

```

```
    def forward(self, x):
        x = x.view(-1, 3, 64, 64)
        x = self.net(x)

        x = x.view(-1, 4*4*4*self.DIM)
        x = self.output(x)

        x = x.mean(0)
        x = x.view(1)
        return x
```

Listing 4: *WGAN generator and critic*

4.2 WGAN Traning

```
# WGAN
# Based on lecture slides
def calc_gradient_penalty(netC, real_data, fake_data, LAMBDA=10):
    epsilon = torch.rand(1).cuda()

    interpolates = epsilon * real_data + ((1 - epsilon) * fake_data)
    interpolates = interpolates.requires_grad_(True).cuda()

    critic_interpolates = netC(interpolates)

    gradients = torch.autograd.grad(outputs = critic_interpolates, inputs=
   interpolates,
                                grad_outputs = torch.ones(critic_interpolates.size()
   ).cuda(),
                                create_graph = True, retain_graph=True,
                                only_inputs=True)[0]

    gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean() * LAMBDA
    return gradient_penalty

# Based on lecture slides
def TrainWGAN(netC, netG, epochs, betas, lr, trainloader):
    nz = 100 # Set the number of channels for the 1x1 input noise vectors for
   the Generator

    netG = netG.to(device)
    netC = netC.to(device)

    netG.apply(weights_init) # initialize network parameters
    netC.apply(weights_init) # initialize network parameters

    fixed_noise = torch.randn(batch_size, nz, 1, 1, device=device) # Make noise
   vector

    one = torch.tensor([1], dtype=torch.float).to(device)
    minus_one = torch.tensor([-1], dtype=torch.float).to(device)

    # Adam optimizers
    optimizerC = torch.optim.Adam(netC.parameters(), lr=lr, betas=betas)
    optimizerG = torch.optim.Adam(netG.parameters(), lr=lr, betas=betas)

```

```
    img_list = []
    G_losses = []
    C_losses = []

    iters = 0
    gen_iterations = 0

    print(f"Training started at time {datetime.datetime.now().time()}")
    start_time = time.time()

    for epoch in range(epochs):

        data_iter = iter(trainDataLoader)
        i = 0
        ncritic = 5

        while i < len(trainDataLoader):

            for p in netC.parameters():
                p.requires_grad = True
            ic = 0

            while ic < ncritic and i < len(trainDataLoader):
                ic += 1

                # Training with real images
                netC.zero_grad()

                real_images_in_batch = next(data_iter)
                real_images_in_batch = real_images_in_batch[0].to(device)

                i += 1

                # Mean value for all images
                critic_for_reals_mean = netC(real_images_in_batch)

                # Target gradient -1
                critic_for_reals_mean.backward(minus_one)

                # Train with fake images
                noise = torch.randn(real_images_in_batch.size(0), nz, 1, 1, device=device)
                fakes = netG(noise)

                # Mean value for batch
                critic_for_fakes_mean = netC(fakes.detach())

                # Aim for target of 1
                critic_for_fakes_mean.backward(one)

                #Gradient penalty
                gradient_penalty = calc_gradient_penalty(netC, real_images_in_batch, fakes)
                gradient_penalty.backward()

                # Calc distance
                wasser_dist = critic_for_reals_mean - critic_for_fakes_mean
```

```
                loss_critic = -wasser_dist + gradient_penalty

                # Update the Critic
                optimizerC.step()

            for p in netC.parameters():
                p.requires_grad = False

            # Train generator
            netG.zero_grad()

            noise = torch.randn(real_images_in_batch.size(0), nz, 1, 1, device=device)
            fakes = netG(noise)

            critic_for_fakes_mean = netC(fakes)
            loss_gen = critic_for_fakes_mean
            critic_for_fakes_mean.backward(minus_one)

            # Update the Generator
            optimizerG.step()
            gen_iterations += 1

            if i % (ncritic * 20) == 0:
                os.makedirs(root_dir + "./models", exist_ok = True)

                print("[epoch=%d/%d iter=%4d elapsed_time=%5d secs] mean_C_loss=%7.4f mean_G_loss=%7.4f wass_dist=%7.4f" %
                ((epoch+1),epochs,(i+1),time.time(), loss_critic.data[0], loss_gen.data[0], wasser_dist.data[0]))

                torch.save(netG.state_dict(), root_dir + "./model/WGAN_gen.pt")
                torch.save(netC.state_dict(), root_dir + "./model/WGAN_crit.pt")

            # Get All Loses
            G_losses.append(loss_gen.data[0].item())
            C_losses.append(loss_critic.data[0].item())

            with torch.no_grad():
                fake_image = netG(fixed_noise).detach().cpu()
            img_list.append(torchvision.utils.make_grid(fake_image, padding=1, pad_value=1, nrow=4, normalize=True))


    print("Traing Time %s sec" % (time.time() - start_time))
    return G_losses, C_losses, img_list
```

Listing 5: *WGAN training*

5 Task 3: Evaluation

```
# Plotting
def plotLoss(lossGen, lossDis, epochs):
    # Plot the training losses
    iterations = range(len(lossDis))
```

```python

    fig = plt.figure(1)
    plt.plot(iterations, lossGen, label="Generator Loss")
    plt.plot(iterations, lossDis, label="Discriminator Loss")

    plt.legend()

    plt.xlabel("Iterations", fontsize = 16)
    plt.ylabel("Loss", fontsize = 16)

    plt.show()
```

Listing 6: *Plotting Results*

```python
# Save Fake Images

def ProduceFakes(netG, netGW):
    # Make dir if not there
    os.makedirs(root_dir + "/DCGAN_fakes", exist_ok = True)
    os.makedirs(root_dir + "/WGAN_fakes", exist_ok = True)

    #Noise
    num_images = 1000

    # Load Networks
    netG.load_state_dict(torch.load(os.path.join(root_dir, "model/DCGAN_gen.pt")
   ))
    netG.eval()

    netGW.load_state_dict(torch.load(os.path.join(root_dir, "model/WGAN_gen.pt")
   ))
    netGW.eval()

    #fake_images = generated_fake_images.detach().cpu()

    #for i in range(num_images):
    validation_noise = torch.randn(num_images, 100, 1, 1, device=device)

    # Generate image with dcgan and wgan
    dcgan_img = netG(validation_noise)
    wgan_img = netGW(validation_noise)

    # Convert to cpu
    fake_dcgan_images = dcgan_img.detach().cpu()
    fake_wgan_images = wgan_img.detach().cpu()

    for i in range(len(fake_dcgan_images)):
        image_dcgan = tvt.ToPILImage()(fake_dcgan_images[i] / 2 + 0.5)
        image_dcgan.save(os.path.join(root_dir + "./DCGAN_fakes/", "DCGAN_image_
   {0}.png".format(i+1)))

        image_wgan = tvt.ToPILImage()(fake_wgan_images[i] / 2 + 0.5)
        image_wgan.save(os.path.join(root_dir + "./WGAN_fakes/", "WGAN_image_
   {0}.png".format(i+1)))
```

Listing 7: *Producing Fake Images for FID*

```
# Validation
def compute_fid_score(fake_paths, real_paths, dims=2048):
    block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
    model = InceptionV3([block_idx]).to(device)

    m1, s1 = calculate_activation_statistics( real_paths, model, device=device)
    m2, s2 = calculate_activation_statistics( fake_paths, model, device=device)

    fid_value = calculate_frechet_distance(m1, s1, m2, s2)
    print(f'FID: {fid_value:.2f}')

    return fid_value
```

Listing 8: *FID Score Calculator*

6 Task 4: Results and Plots

Plotting for loss over iteration and the resulting image predictions are shown below.

```
# Main

# Get Trainloader
batch_size = 16
image_shape = 64

trainDataLoader = createDataLoader(train_data_path, batch_size, image_shape)
testDataLoader = createDataLoader(test_data_path, batch_size, image_shape)

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

# Set Params

lr = 1e-5*4
betas = (0.5, 0.999)
epochs = 87

netG = Generator()
netD = Discriminator()

G_losses, D_losses, DCGAN_image_list = TrainDCGAN(netD, netG, epochs, betas, lr,
    trainDataLoader)

netGW = Generator()
netC = Critic()

lr = 1e-3
epochs = 250

GW_losses, C_losses, WGAN_image_list = TrainWGAN(netC, netGW, epochs, betas, lr,
    trainDataLoader)
```

```

plotLoss(G_losses, D_losses, epochs) #DCGAN

plotLoss(GW_losses, C_losses, epochs) #WGAN

plot_fake_real_images(DCGAN_image_list, trainDataLoader) # DCGAN

plot_fake_real_images(WGAN_image_list, trainDataLoader) # WGAN

ProduceFakes(netG, netGW)

real_imgs = [test_data_path + "/eval/" + i for i in os.listdir(test_data_path + "/eval/")]
dcgan_imgs = [root_dir + "./DCGAN_fakes/" + i for i in os.listdir(root_dir + "./DCGAN_fakes/")]
wgan_imgs = [root_dir + "/WGAN_fakes/" + i for i in os.listdir(root_dir + "./WGAN_fakes/")]

dcgan_fid = compute_fid_score(dcgan_imgs, real_imgs, dims=2048)
wgan_fid = compute_fid_score(wgan_imgs, real_imgs, dims=2048)
```

Listing 9: *Main To Run Code*

The results indicate that the WCGAN performs better than the DCGAN network. This is evident by the FID score being lower than $110 < 150$ and the images being more diverse and clear. This result makes sense as the WCGAN uses a more sophisticated method of the Wasserstein distance paired with the critic that is just a discriminator. It can be seen in Figure 3 vs Figure 2 that the images are less blurry and have fewer replication effects than the DCGAN. Furthermore, as seen in Figure 1, the loss of WCGAN is much lower and with less noise than that of DCGAN which spikes, significantly more which could be due to the issue of the vanishing gradient problem that more may have been resolvable by adding skip connections.

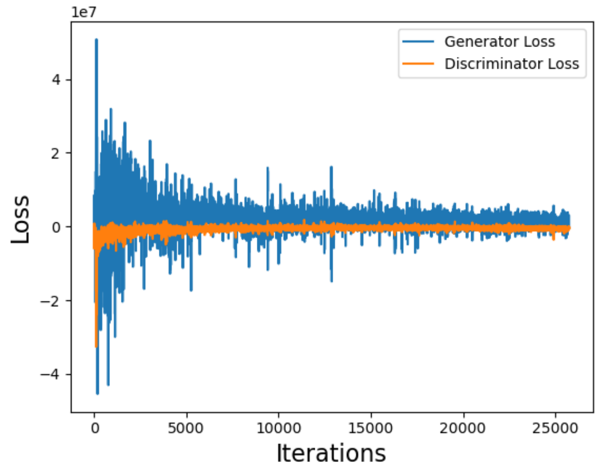

Figure 1: *WCGAN*

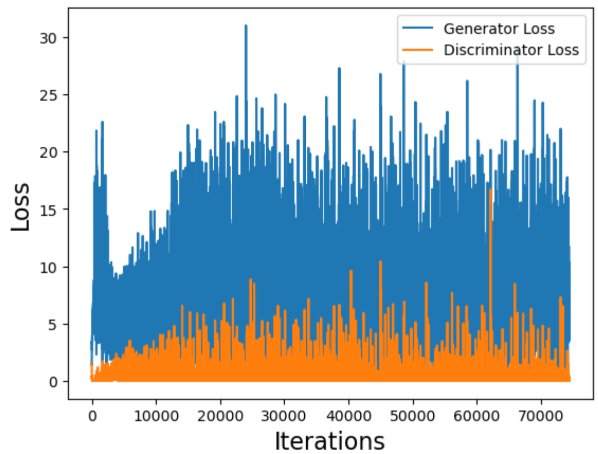

Figure 2: *DCGAN*

Figure 3: *Loss per iterations*

Figure 4: *DCGAN Fake and Real Images*

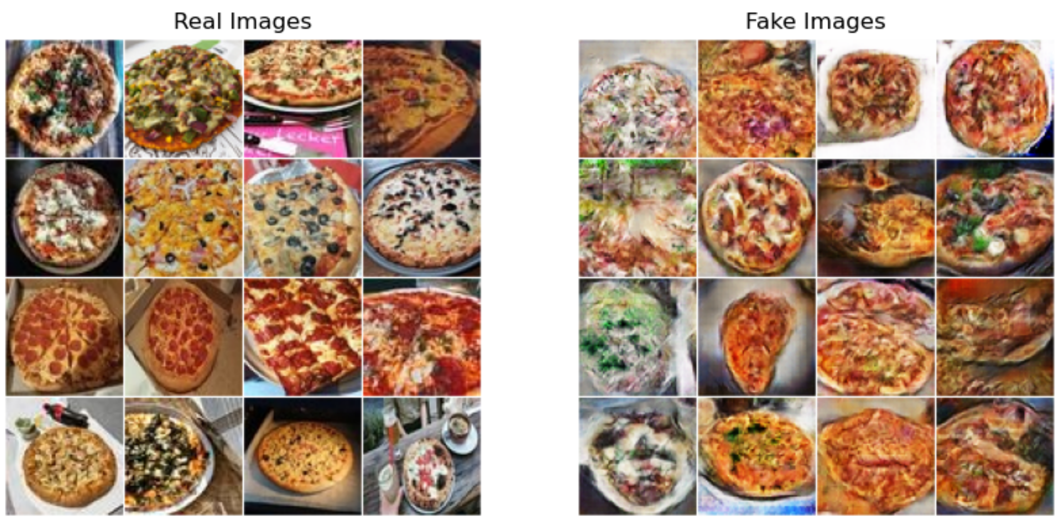

Figure 5: *WGAN Fake and Real Images*

```
1  dcgan_fid = compute_fid_score(dcgan_imgs, real_imgs, dims=2048)

100%|██████████| 20/20 [00:04<00:00,  4.70it/s]
100%|██████████| 20/20 [00:01<00:00, 10.10it/s]

FID: 150
                                                click to scroll output; double click to hide
1  wgan_fid = compute_fid_score(wgan_imgs, real_imgs, dims=2048)

100%|██████████| 20/20 [00:01<00:00, 10.31it/s]
100%|██████████| 20/20 [00:02<00:00,  9.91it/s]

FID: 110.46
```

Figure 6: *FID Score*

7 Lessons learned

From this assignment, I learned how to create a DCGAN and WGAN network along with creating a proper discriminator and critic for producing proper results.