Homework 7 - Deep Learning (ECE 60146)

Souradip Pal
pal43@purdue.edu
PUID 0034772329

April 6, 2023

1 Introduction

In this homework, the programming tasks give an overview of Generative Adversarial Networks for data generation and modeling. It introduces the idea of how generator and discriminator networks can be trained in a mini-max fashion to learn the underlying probability distributions of the training data which can then be used to generate synthetic data. In this assignment, two different GANs were created having different loss criteria (1)**Binary Cross Entropy Loss** and (2) **Wasserstein distance**. An appropriate training routine was implemented to train the network and generate fake images. The network was trained and validated using images present in the **Pizza** dataset provided along with the homework instructions. Finally, the performance of the network was compared by computing the **Frechet-Inception-Distance(FID)** for the generated images against a set of validation images. Some ideas related to the logic of the training and validation routines were taken from the source code of the **AdversarialLearning** directory from *DLStudio* module and from the previous year's solutions for completing this assignment.

2 Methodology

Initially, to get familiar with the code for training and testing GAN models, the scripts *dcgan_DG1.py* and *wgan_CG1.py* from the **ExamplesAdversarialLearning** directory in the **DLStudio** module were run. These scripts illustrated how synthetic images were generated by the Generator network and the Discriminator network (Critic network in the case of WGAN) was trained to distinguish between the real and fake images. Using a similar approach for the programming tasks, a custom training routine was created and used along with the *Generator*, *Discriminator* networks for DCGAN and *GeneratorCG*, *CriticCG* networks for WGAN training. The training routine contains the necessary code to log the losses in periodic checkpoints. For evaluation, a method was created to generate fake pizza images from the trained Generator networks and compare the values of the FID(Frechet-Inception-Distance) scores with respect to the real validation images as per the instructions provided. The following section gives a detailed description of each of the approaches and the results obtained from the experiments performed on the GAN models.

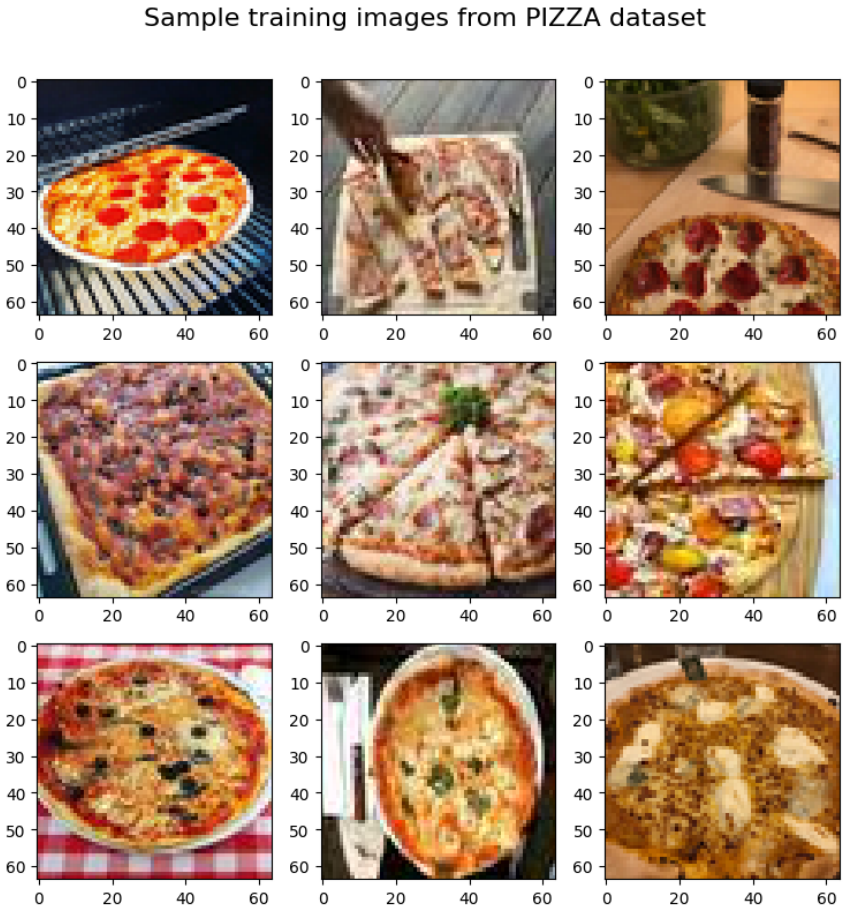

Figure 1: Sample of training images from the Pizza dataset

3 Implementation and Results

3.1 Task 3: Programming Tasks

3.1.1 Preparing the Dataset

- In this assignment, a pizza dataset was provided to train the pizza-generating GAN model. Each of the RGB images in the dataset was of 64×64 size. A total of **8157** training images and **1000** testing images were stored in separate directories. Shown in Fig. 1 and Fig. 2 are samples of the images from the training and validation set. A custom dataset class called *PizzaDataset* was created from *torch.utils.data.Dataset* class to load the images from the directories after applying the necessary transforms. Finally, a *Dataloader* was created to wrap the dataset for processing the images in batches of **64** for training.

3.1.2 Building and Training GAN

- **DCGAN with BCE Loss**: In this task, custom models for the Generator(*Generator*) and Discriminator(*Discriminator*) were created similar to the ones present in the *DLStudio* module. The Generator consists of a series of transposed convolutional layers followed by BatchNorm

Figure 2: Sample of validation images from the Pizza dataset

layers to upsample the latent features into generated images. In addition to that, ReLU layers were used as activation functions. TanH activation function was also used in the last layer. The network consisted of 13 layers in total with $\sim$**3.5M** parameters. The input to the network is a latent vector z, having 100 states, that is drawn from a normal distribution ($\mu = 0, \sigma^2 = 1$), and the output is a $3 \times 64 \times 64$ RGB image. The following code block gives a description of the *Generator* network.

```
# DCGAN Generator
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.model = nn.Sequential(
            nn.ConvTranspose2d(100, 64 * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(64 * 8),
            nn.ReLU(True),
            nn.ConvTranspose2d(64 * 8, 64 * 4, 4, 2, 1, bias=False),
```

```
            nn.BatchNorm2d(64 * 4),
            nn.ReLU(True),
            nn.ConvTranspose2d( 64 * 4, 64 * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 2),
            nn.ReLU(True),
            nn.ConvTranspose2d(64 * 2, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 3, 4, 2, 1, bias=False),
            nn.Tanh()
            )

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

In the Discriminator model, convolutional layers were used followed by BatchNorm and LeakyReLU layers with a negative slope of 0.2. The last layer of the model was followed by a Sigmoid activation function to obtain the probability of predicting whether the image was real vs fake. The code block below gives a description of the *Discriminator* network. The total number of learnable layers of the entire network came out to be **11** and the total number of learnable parameters was $\sim$**2.7M**. The following code snippet shows the Discriminator network.

```
# DCGAN Discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.model = nn.Sequential(
            nn.Conv2d(3, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 64 * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64 * 2, 64 * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 4),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64 * 4, 64 * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64 * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
            )

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

Using the training routine created, the two networks were trained for 50 epochs with BCE Loss as the loss criterion and optimizer as Adam having parameters $\beta_1 = 0.5$ and $\beta_2 = 0.999$ and learning rate 0.0002. In the training routine, the discriminator losses were computed based on the labels of the real and fake images predicted. Now keeping the discriminator fixed, the Generator loss was computed based on the fake images generated considering them as real images, and the parameters of the generator were subsequently updated. Fig. 3 shows the training loss variation with the number of iterations while training *DCGAN* with BCE Loss.

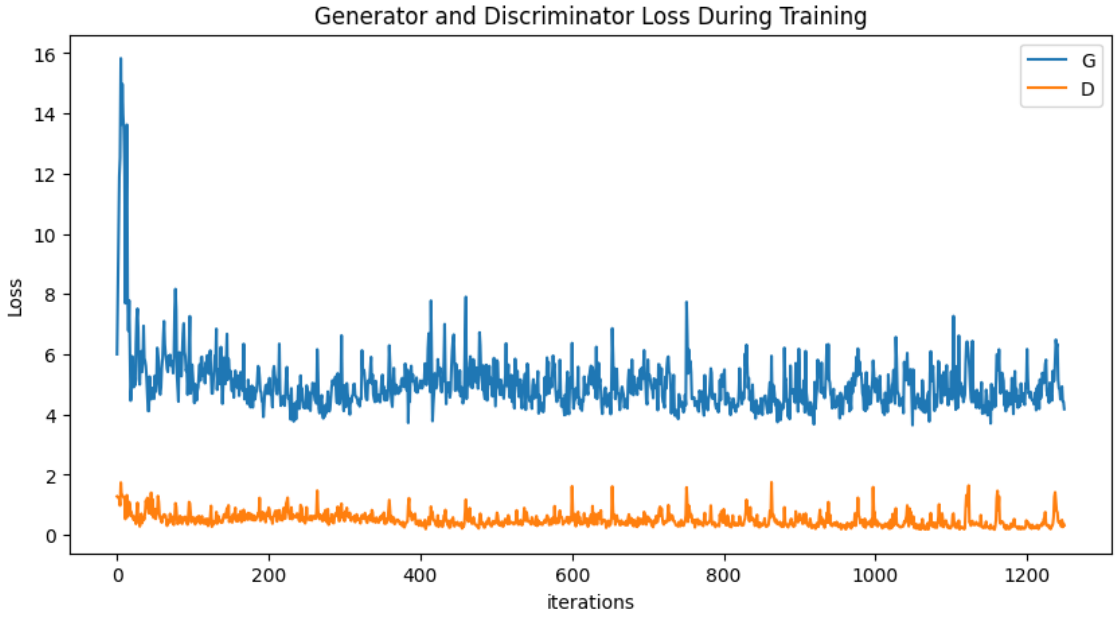

Figure 3: Training loss vs Iterations for *DCGAN* with BCE Loss

- **WGAN**: For Wasserstein GAN, custom Generator(*GeneratorCG*) and Critic(*CriticCG*) models were created similar to the one in DCGAN. The Generator consists of a series of transposed convolutional layers followed by BatchNorm layers to upsample the latent features into generated images. In addition to that, ReLU layers were used as activation functions in the initial layers and the TanH activation function was used in the last layer. The network consisted of 13 layers in total with **∼3.5M** parameters. The input to the network is a latent vector z, having 100 states, that is drawn from a normal distribution ($\mu = 0, \sigma^2 = 1$), and the output is a $3 \times 64 \times 64$ RGB image. The following code block gives a description of the *GeneratorCG* network.

```
# Generator for WGAN
class GeneratorCG(nn.Module):
    def __init__(self):
        super(GeneratorCG, 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.relu = nn.ReLU()
        self.tanh  = nn.Tanh()
```

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

In the Critic model, convolutional layers were used followed by BatchNorm and LeakyReLU layers with a negative slope of 0.2. A skip block was also used in the initial layer. The final layer is a Linear layer that takes in the flattened image features and produces a single value. These values are then averaged over the entire batch which is then used for calculating the Wasserstein distance. The code block below gives a description of the *Discriminator* network. The total number of learnable layers of the entire network came out to be **56** and the total number of learnable parameters was $\sim$**3.6M**. The following code snippet shows the Critic network.

```
# Critic for WGAN
class CriticCG(nn.Module):
    def __init__(self):
        super(CriticCG, self).__init__()
        self.conv_in = SkipBlock(3, 64, downsample=True, skip_connections=True)
        self.conv_in2 = SkipBlock(64, 128, downsample=True, skip_connections=False)
        self.conv_in3 = SkipBlock(128, 256, downsample=True, skip_connections=False)
        self.conv_in4 = SkipBlock(256, 512, downsample=True, skip_connections=False)
        self.bn1 = nn.BatchNorm2d(128)
        self.bn2 = nn.BatchNorm2d(256)
        self.bn3 = nn.BatchNorm2d(512)
        self.leaky_relu = nn.LeakyReLU(0.2, inplace=True)
        self.final = nn.Linear(512*4*4, 1)

    def forward(self, x):
        x = self.leaky_relu(self.conv_in(x))
        x = self.bn1(self.conv_in2(x))
        x = self.leaky_relu(x)
        x = self.bn2(self.conv_in3(x))
        x = self.leaky_relu(x)
        x = self.bn3(self.conv_in4(x))
        x = self.leaky_relu(x)
        x = x.flatten(1)
        x = self.final(x)
        x = x.mean(0)
        x = x.view(1)
        return x
```

Using a different training routine created for WGAN, the two networks were trained for 100 epochs with Wasserstein distance as the loss criterion and optimizer as Adam having parameters $\beta_1 = 0.5$ and $\beta_2 = 0.999$ and learning rate 0.0002. The clipping threshold was kept at 0.01. In the training routine, the discriminator losses were computed based on the average of the model outputs and taking their differences for the real and fake images to calculate the Wasserstein distance. Now keeping the discriminator fixed, the Generator loss was computed based on the fake images generated, and the parameters of the generator were subsequently updated. The critic was trained for 100 iterations initially to 5 iterations for each Generator update. Fig. 4 shows the training loss variation with the number of iterations while training *WGAN*.

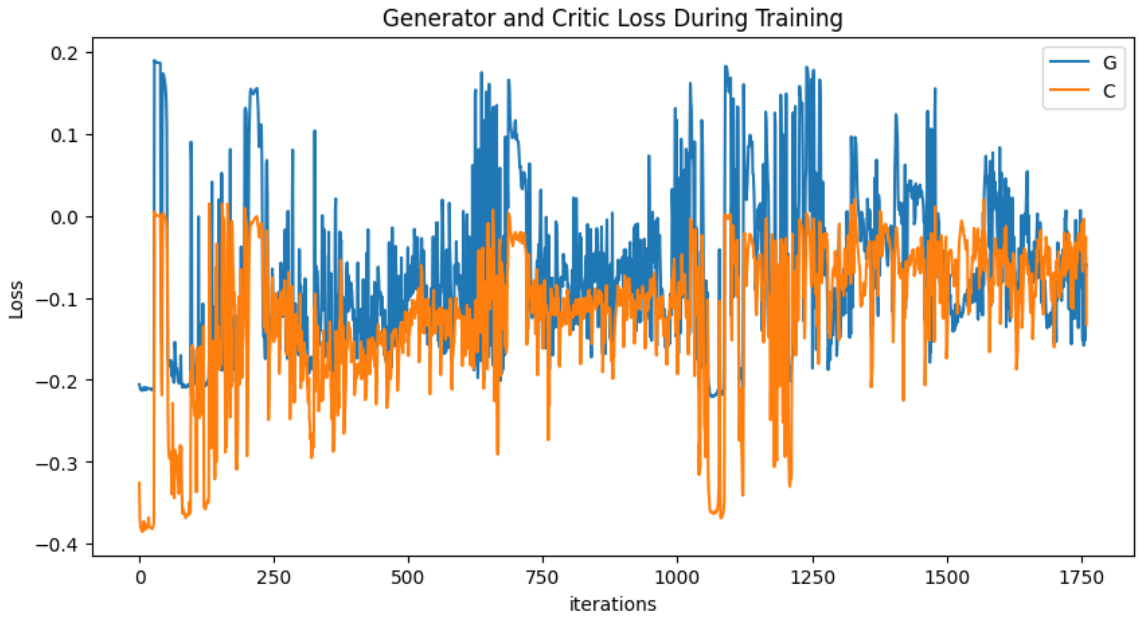

Figure 4: Training loss vs Iterations for *WGAN*

3.1.3 Evaluating the GANs

- **FID Score**: In order to validate the networks, a validation routine was created where 1000 fake pizza images were generated using the Generator network of the GAN models. Figure 5 and Figure 6 show the resulting fake pizza images generated by the corresponding Generator networks for DCGAN with BCE Loss and WGAN respectively. The quality and diversity of the generated images were then evaluated using Frechet Inception Distance(FID) metric over the validation images. To use the FID metric, a Python library called *pytorch-fid* was first installed and then used to calculate the FID score based on those 1k generated and validation images as per the instruction. The FID score for the images generated by **DCGAN with BCE Loss** was **129.88**. The FID score for the images generated by **WGAN** was **181.83**.
- **Results & Comparison**: It can be observed that the training of the WGAN is much slower in comparison to the DCGAN. The main issue encountered is in the convergence of the WGAN model. The loss is somewhat unstable and oscillates quite frequently which is mainly due to vanishing gradients and often leads to Mode collapse if the parameters of the model are chosen appropriately. In general WGAN can provide better diverse images than DCGAN which is not the case here. These issues can be avoided by employing BatchNorm or Instance norm layers on top of the Convolutional layers. Better results can also be obtained by applying a gradient penalty rather than gradient clipping for WGAN. Moreover, due to resource and time constraints, the DCGAN models were run only up to 50 epochs and WGAN up to 100 epochs hence it was able to provide a somewhat decent level of FID score. Different learning rates and different optimizers could also be used to train the Generator and Discriminator modules which might lead to faster training and better performance.

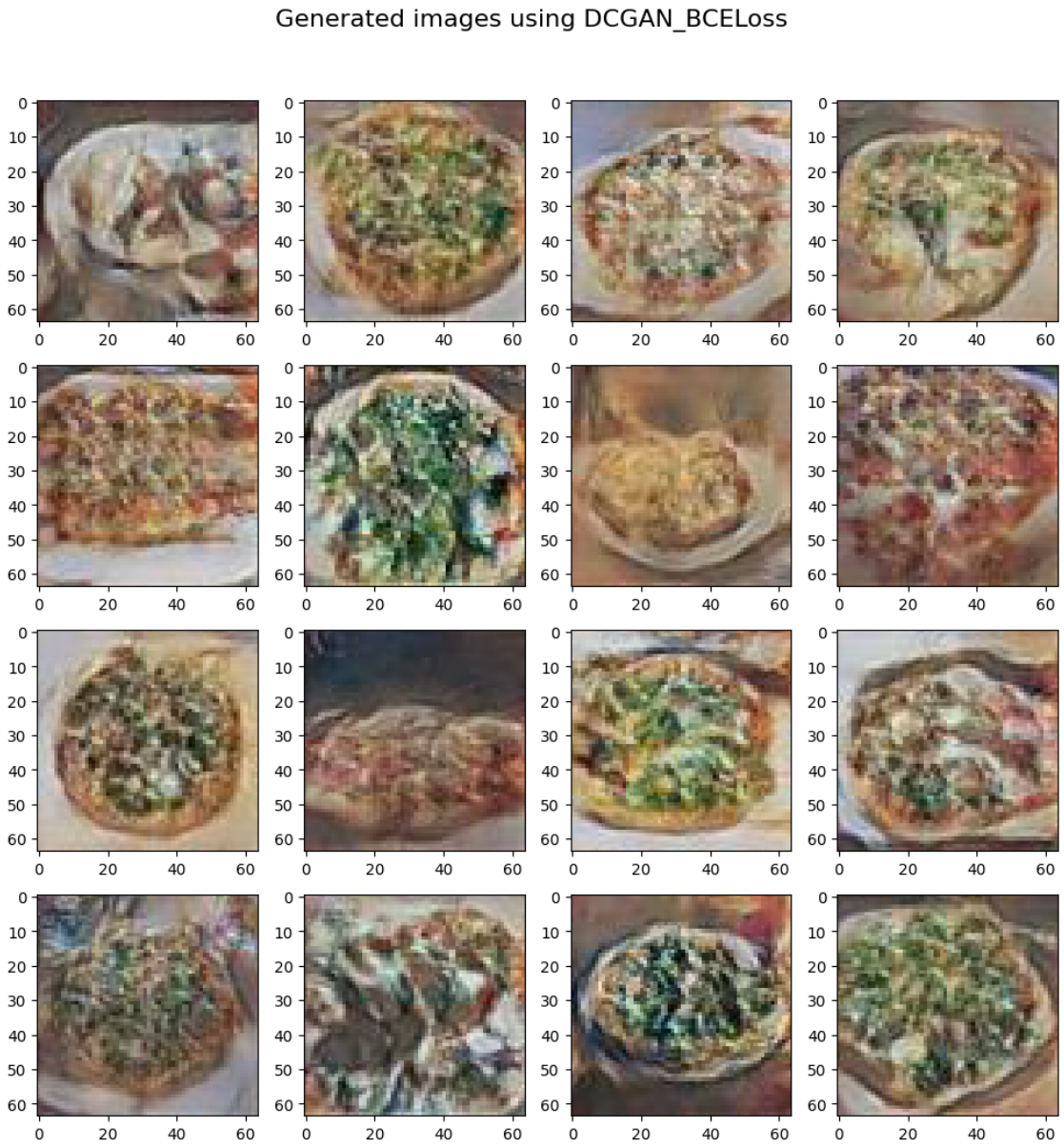

Figure 5: Sample fake Pizza images generated using *DCGAN* with BCE Loss

4 Conclusion

In conclusion, Generative Adversarial Networks are powerful tools to capture image features and generate similar synthetic images which can be used for data modeling. However, sufficient care must be taken in the training of GAN models. BatchNorm layers in the Generator and Discriminator networks provide a way to control the convergence of the losses. Moreover, for Wasserstein GAN, gradient clipping or gradient penalty should be applied to effectively enforce the Lipschitz condition on the Critic. Hence, well-designed GAN architectures containing these elements along with appropriate hyper-parameters are essential in making the training much more efficient and stable producing good

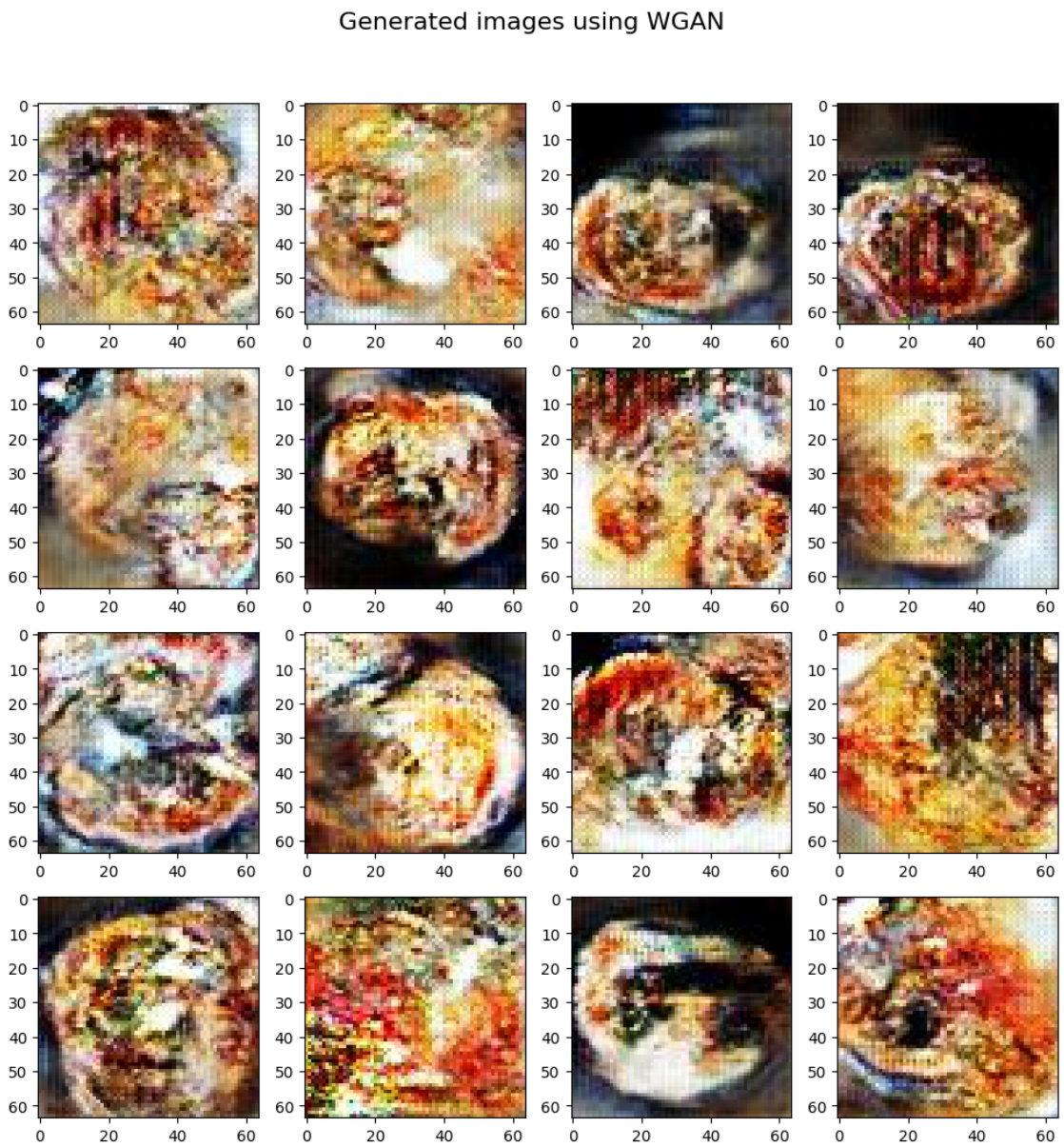

Figure 6: Sample fake Pizza images generated using *WGAN*

quality images with higher FID score.

5 Source Code

```
# -*- coding: utf-8 -*-
"""hw7_SouradipPal.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1V-aEyQyL4ry1kcADf4lvxBAmta7uM_8y
"""

from google.colab import drive
drive.mount('/content/drive')

# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/MyDrive/Purdue/ECE60146/

!wget -O DLStudio-2.2.5.tar.gz \
    https://engineering.purdue.edu/kak/distDLS/DLStudio-2.2.5.tar.gz?download

!tar -xvf DLStudio-2.2.5.tar.gz

# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/MyDrive/Purdue/ECE60146/DLStudio-2.2.5/

!pip install pymsgbox

!python setup.py install

# %load_ext autoreload
# %autoreload 2

import os
import torch
import random
import numpy as np
import requests
import matplotlib.pyplot as plt

from tqdm import tqdm
from PIL import Image

seed = 0
random.seed(seed)
np.random.seed(seed)

from DLStudio import *

# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/MyDrive/Purdue/ECE60146/DLStudio-2.2.5/ExamplesAdversarialLearning/

!wget -O /content/datasets_for_AdversarialNetworks.tar.gz \
    https://engineering.purdue.edu/kak/distDLS/datasets_for_AdversarialNetworks.tar.gz

!tar -xvf /content/datasets_for_AdversarialNetworks.tar.gz -C \
    /content/drive/MyDrive/Purdue/ECE60146/DLStudio-2.2.5/ExamplesAdversarialLearning/dataGAN

!tar -xvf ./dataGAN/PurdueShapes5GAN-20000.tar.gz -C ./dataGAN/

!python dcgan_DG1.py

!python wgan_CG1.py

!mkdir /content/drive/MyDrive/Purdue/ECE60146/HW7/data/
!mkdir /content/drive/MyDrive/Purdue/ECE60146/HW7/saved_models

!unzip /content/pizzas.zip -d /content/drive/MyDrive/Purdue/ECE60146/HW7/data/

# Plotting sample training images
fig, axes = plt.subplots(3, 3, figsize=(9, 9))

root = '/content/drive/MyDrive/Purdue/ECE60146/HW7/data/train/'
img_paths = os.listdir(root)
for i in range(3):
```

```
    for j in range(3):
        path = img_paths[3*i+j]
        im = Image.open(os.path.join(root, path)).convert('RGB')
        im = np.ascontiguousarray(im, dtype=np.uint8)
        axes[i][j].imshow(im)

fig.suptitle('Sample training images from PIZZA dataset', fontsize=16, y=0.95)
plt.show()

# Plotting sample validation images
fig, axes = plt.subplots(3, 3, figsize=(9, 9))

root = '/content/drive/MyDrive/Purdue/ECE60146/HW7/data/eval/'
img_paths = os.listdir(root)
for i in range(3):
    for j in range(3):
        path = img_paths[3*i+j]
        im = Image.open(os.path.join(root, path)).convert('RGB')
        im = np.ascontiguousarray(im, dtype=np.uint8)
        axes[i][j].imshow(im)

fig.suptitle('Sample validation images from PIZZA dataset', fontsize=16, y=0.95)
plt.show()

import os
import torch

# Custom dataset class for Pizza GAN
class PizzaDataset(torch.utils.data.Dataset):
    def __init__(self, root, transforms=None, mode = 'train'):
        super().__init__()
        self.mode = mode
        self.root_dir = root
        self.img_paths = os.listdir(self.root_dir)
        self.transforms = transforms

    def __len__(self):
        # Return the total number of images
        return len(self.img_paths)

    def __getitem__(self, index):
        index = index % len(self.img_paths)
        img_path = self.img_paths[index]
        im = Image.open(os.path.join(self.root_dir, img_path)).convert('RGB')
        im_transformed = self.transforms(im)

        return im_transformed

import torchvision.transforms as tvt

reshape_size = 64
transforms = tvt.Compose([
    tvt.ToTensor(),
    tvt.Resize((reshape_size, reshape_size), antialias=True),
    tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

# Create custom training dataset
train_dataset = PizzaDataset('/content/drive/MyDrive/Purdue/ECE60146/HW7/data/train/', transforms=transforms)

len(train_dataset)

# Create custom validation dataset
val_dataset = PizzaDataset('/content/drive/MyDrive/Purdue/ECE60146/HW7/data/eval/', transforms=transforms, mode = 'test')

len(val_dataset)

# Create custom training/validation dataloader
train_data_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True, num_workers=2)
val_data_loader = torch.utils.data.DataLoader(val_dataset, batch_size=64, shuffle=False, num_workers=2)

# Check dataloader
train_loader_iter = iter(train_data_loader)
img = next(train_loader_iter)

print('img has length: ', len(img))
```

```
#print('target has length: ', len(label))
print(img[0].shape)

# Method to initialize Conv and BatchNorm weights
def weights_init(m):
    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)

import torch.nn as nn
import torch.optim as optim
import torchvision
import imageio

# Routine to train a DCGAN
def train_gan(device, generator, discriminator, data_loader, model_name,
              dir_name_for_results = None, epochs = 10, display_interval = 100):

    batch_size = 64
    nz = 100
    netD = discriminator.to(device)
    netG = generator.to(device)

    fixed_noise = torch.randn(batch_size, nz, 1, 1, device=device)
    real_label = 1
    fake_label = 0

    # Adam optimizers for the Discriminator and the Generator:
    optimizerD = optim.Adam(netD.parameters(), lr=0.0002, betas=(0.5, 0.999))
    optimizerG = optim.Adam(netG.parameters(), lr=0.0002, betas=(0.5, 0.999))

    # Loss
    criterion = nn.BCELoss()

    img_list = []
    G_losses = []
    D_losses = []
    mean_D_losses = []
    mean_G_losses = []

    iters = 0
    print("\n\nStarting Training Loop...\n\n")
    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(data_loader, 0):
            # Training Discriminator with Real Images
            netD.zero_grad()
            real_images_in_batch = data.to(device)
            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()

            # Training Discriminator with Fake Images
            noise = torch.randn(b_size, nz, 1, 1, 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)

            # Update Discriminator
            optimizerD.step()

            # Training Generator
            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) % display_interval == 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.item())
            mean_G_losses.append(mean_G_loss.item())
            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(data_loader)-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(dir_name_for_results + "gen_and_disc_loss_training.png")
plt.show()

plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(mean_G_losses,label="G")
plt.plot(mean_D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.savefig(dir_name_for_results + "gen_and_disc_loss_training2.png")
plt.show()

# Comparison of Real and Fake Images
real_batch = next(iter(data_loader))
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Real Images")
r_imgs = np.transpose(torchvision.utils.make_grid(real_batch.to(device), \
                        padding=1, pad_value=1, normalize=True).cpu(),(1,2,0))
plt.imshow(r_imgs)
plt.savefig(dir_name_for_results + 'Real_Images.jpg')
plt.show()

plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(img_list[-1],(1,2,0)))
plt.savefig(dir_name_for_results + 'Fake_Images.jpg')
plt.show()

images = []
for imgobj in img_list:
    img = tvt.ToPILImage()(imgobj)
    images.append(img)
imageio.mimsave(dir_name_for_results + "generation_animation.gif", images, fps=5)

# Save model weights
checkpoint_path_G = os.path.join('/content/drive/MyDrive/Purdue/ECE60146/HW7/saved_models',
                                 f'{model_name}_G.pt')
torch.save(netG.state_dict(), checkpoint_path_G)
```

```
    checkpoint_path_D = os.path.join('/content/drive/MyDrive/Purdue/ECE60146/HW7/saved_models',
                                     f'{model_name}_D.pt')
    torch.save(netD.state_dict(), checkpoint_path_D)

    return mean_D_losses, mean_G_losses, img_list

# Plotting training loss
def plot_loss(mean_D_losses, mean_G_losses, display_interval, model_name):
    plt.figure()
    plt.plot(np.arange(len(mean_D_losses))*display_interval, mean_D_losses, label="Discriminator Loss");
    plt.plot(np.arange(len(mean_G_losses))*display_interval, mean_G_losses, label="Generator Loss");
    plt.title(f'Training Loss for {model_name}')
    plt.xlabel('Iterations')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

!pip install pytorch-fid

# Routine to validate GAN
from pytorch_fid.fid_score import calculate_activation_statistics, calculate_frechet_distance
from pytorch_fid.inception import InceptionV3

def validate_gan(device, netG, real_root_dir, fake_root_dir, model_name = None, model_path = None):
    if model_path is not None:
        netG.load_state_dict(torch.load(model_path))
    netG = netG.to(device)
    netG.eval()

    nz = 100
    noise = torch.randn(1000, nz, 1, 1, device=device)
    img_paths = []

    invTrans = tvt.Compose([ tvt.Normalize(mean = [ 0., 0., 0. ], std = [ 1/0.5, 1/0.5, 1/0.5 ]),
                             tvt.Normalize(mean = [ -0.5, -0.5, -0.5 ], std = [ 1., 1., 1. ]),
                           ])
    # generate fake images
    with torch.no_grad():
        fakes = netG(noise)
        for i in range(noise.shape[0]):
            img = invTrans(fakes[i])
            img = tvt.ToPILImage()(img)
            im_path = os.path.join(fake_root_dir, f"gen_{i:03d}.jpeg")
            img.save(im_path)
            img_paths.append(im_path)

    # calculate FID score
    real_img_paths = os.listdir(real_root_dir)
    fake_img_paths = os.listdir(fake_root_dir)
    real_paths = [os.path.join(real_root_dir, path) for path in real_img_paths]
    fake_paths = [os.path.join(fake_root_dir, path) for path in fake_img_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}')

    # Plotting sample generated images
    fig, axes = plt.subplots(4, 4, figsize=(12, 12))
    for i in range(4):
        for j in range(4):
            path = img_paths[4*i+j]
            im = Image.open(os.path.join(root, path)).convert('RGB')
            im = np.ascontiguousarray(im, dtype=np.uint8)
            axes[i][j].imshow(im)

    fig.suptitle(f'Generated images using {model_name}', fontsize=16, y=0.95)
    plt.show()

# DCGAN Generator
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
```

```
        self.model = nn.Sequential(
            nn.ConvTranspose2d(100, 512, 4, 1, 0, bias=False),
            nn.BatchNorm2d(64 * 8),
            nn.ReLU(True),
            nn.ConvTranspose2d(512, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            nn.ConvTranspose2d(256, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 3, 4, 2, 1, bias=False),
            nn.Tanh()
            )

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

# DCGAN Discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.model = nn.Sequential(
            nn.Conv2d(3, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(128, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(256, 512, 4, 2, 1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(512, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
            )

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

# Initialize device
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
device

# Initialize DCGAN
netD = Discriminator()
netG = Generator()

netD.apply(weights_init)
netG.apply(weights_init)

# netG.load_state_dict(torch.load('/content/drive/MyDrive/Purdue/ECE60146/HW7/saved_models/HW7_BCE_2_G.pt'))
# netD.load_state_dict(torch.load('/content/drive/MyDrive/Purdue/ECE60146/HW7/saved_models/HW7_BCE_2_D.pt'))

epochs = 50
display_interval = 5

num_learnable_params_disc = sum(p.numel() for p in netD.parameters() if p.requires_grad)
print("\n\nThe number of learnable parameters in the Discriminator: %d\n" % num_learnable_params_disc)
num_learnable_params_gen = sum(p.numel() for p in netG.parameters() if p.requires_grad)
print("\nThe number of learnable parameters in the Generator: %d\n" % num_learnable_params_gen)
num_layers_disc = len(list(netD.parameters()))
print("\nThe number of layers in the Discriminator: %d\n" % num_layers_disc)
num_layers_gen = len(list(netG.parameters()))
print("\nThe number of layers in the Generator: %d\n\n" % num_layers_gen)

# Train DCGAN with BCE Loss
net1_losses = train_gan(device, netG, netD, data_loader = train_data_loader,
                        model_name = 'HW7_BCE_3', dir_name_for_results =
                         '/content/drive/MyDrive/Purdue/ECE60146/HW7/data/bce_gan/',
                        epochs=epochs, display_interval = display_interval)
```

```
# Validate DCGAN with BCE Loss
save_path = '/content/drive/MyDrive/Purdue/ECE60146/HW7/saved_models/HW7_BCE_3_G.pt'
validate_gan(device, netG,'/content/drive/MyDrive/Purdue/ECE60146/HW7/data/eval', \
                '/content/drive/MyDrive/Purdue/ECE60146/HW7/data/bce_gan/images',
             model_name = 'DCGAN_BCELoss', model_path = save_path)

# Skip Block for WGAN
class SkipBlock(nn.Module):
    def __init__(self, in_ch, out_ch, downsample=False, skip_connections=True):
        super(SkipBlock, self).__init__()
        self.downsample = downsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.convo1 = nn.Conv2d(in_ch, out_ch, kernel_size=3, stride=1, padding=1)
        self.convo2 = nn.Conv2d(in_ch, out_ch, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)
        self.downsampler1 = nn.Conv2d(in_ch, out_ch, kernel_size=1, stride=2)
        self.downsampler2 = nn.Conv2d(out_ch, out_ch, kernel_size=1, stride=2)
        self.leaky_relu = nn.LeakyReLU(0.2)

    def forward(self, x):
        identity = x
        out = self.convo1(x)
        out = self.bn1(out)
        out = torch.nn.functional.relu(out)
        if self.in_ch == self.out_ch:
            out = self.convo2(out)
            out = self.bn2(out)
            out = self.leaky_relu(out)
        if self.downsample:
            out = self.downsampler2(out)
            identity = self.downsampler1(identity)
        if self.skip_connections:
            out += identity
        return out

# Critic for WGAN
class CriticCG(nn.Module):
    def __init__(self):
        super(CriticCG, self).__init__()
        self.conv_in = SkipBlock(3, 64, downsample=True, skip_connections=True)
        self.conv_in2 = SkipBlock(64, 128, downsample=True, skip_connections=False)
        self.conv_in3 = SkipBlock(128, 256, downsample=True, skip_connections=False)
        self.conv_in4 = SkipBlock(256, 512, downsample=True, skip_connections=False)
        self.bn1 = nn.BatchNorm2d(128)
        self.bn2 = nn.BatchNorm2d(256)
        self.bn3 = nn.BatchNorm2d(512)
        self.leaky_relu = nn.LeakyReLU(0.2, inplace=True)
        self.final = nn.Linear(512*4*4, 1)

    def forward(self, x):
        x = self.leaky_relu(self.conv_in(x))
        x = self.bn1(self.conv_in2(x))
        x = self.leaky_relu(x)
        x = self.bn2(self.conv_in3(x))
        x = self.leaky_relu(x)
        x = self.bn3(self.conv_in4(x))
        x = self.leaky_relu(x)
        x = x.flatten(1)
        x = self.final(x)
        x = x.mean(0)
        x = x.view(1)
        return x

# Generator for WGAN
class GeneratorCG(nn.Module):
    def __init__(self):
        super(GeneratorCG, 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.relu = nn.ReLU()
        self.tanh  = nn.Tanh()

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

# Initialize WGAN
netC = CriticCG()
netG = GeneratorCG()

netC.apply(weights_init)
netG.apply(weights_init)

# netG.load_state_dict(torch.load('/content/drive/MyDrive/Purdue/ECE60146/HW7/saved_models/HW7_WGAN_2_G.pt'))
# netC.load_state_dict(torch.load('/content/drive/MyDrive/Purdue/ECE60146/HW7/saved_models/HW7_WGAN_2_C.pt'))

epochs = 100
display_interval = 20

num_learnable_params_critic = sum(p.numel() for p in netC.parameters() if p.requires_grad)
print("\n\nThe number of learnable parameters in the Critic: %d\n" % num_learnable_params_critic)
num_learnable_params_gen = sum(p.numel() for p in netG.parameters() if p.requires_grad)
print("\nThe number of learnable parameters in the Generator: %d\n" % num_learnable_params_gen)
num_layers_critic = len(list(netC.parameters()))
print("\nThe number of layers in the Critic: %d\n" % num_layers_critic)
num_layers_gen = len(list(netG.parameters()))
print("\nThe number of layers in the Generator: %d\n\n" % num_layers_gen)

# Routine to train a WGAN
def train_wgan(device, critic, generator, data_loader, model_name,
               dir_name_for_results, epochs = 10, display_interval = 100):

    nz = 100

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

    batch_size = 64

    fixed_noise = torch.randn(batch_size, nz, 1, 1, device=device)

    one = torch.FloatTensor([1]).to(device)
    minus_one = torch.FloatTensor([-1]).to(device)

    #  Adam optimizers for the Critic and the Generator:
    optimizerC = optim.Adam(netC.parameters(), lr=0.0002, betas=(0.5, 0.999))
    optimizerG = optim.Adam(netG.parameters(), lr=0.0002, betas=(0.5, 0.999))

    img_list = []
    Gen_losses = []
    Cri_losses = []

    iters = 0
    gen_iterations = 0
    print("\n\nStarting Training Loop.......\n\n")
    clipping_thresh = 0.005
    # For each epoch
    for epoch in range(epochs):
        data_iter = iter(data_loader)
        i = 0
        ncritic = 5
        while i < len(data_loader):
            for p in netC.parameters():
```

```
            p.requires_grad = True
        if gen_iterations < 25 or gen_iterations % display_interval == 0:
            ncritic = 100
        ic = 0
        while ic < ncritic and i < len(data_loader):
            ic += 1
            for p in netC.parameters():
                p.data.clamp_(-clipping_thresh, clipping_thresh)

            ## Training the Critic with real images
            netC.zero_grad()
            real_images_in_batch = next(data_iter)
            i += 1
            real_images_in_batch = real_images_in_batch.to(device)
            b_size = real_images_in_batch.size(0)
            critic_for_reals_mean = netC(real_images_in_batch)
            critic_for_reals_mean.backward(minus_one)

            ## Training the Critic with fake images
            noise = torch.randn(b_size, nz, 1, 1, device=device)
            fakes = netG(noise)
            critic_for_fakes_mean = netC(fakes)
            critic_for_fakes_mean.backward(one)
            wasser_dist = critic_for_reals_mean - critic_for_fakes_mean
            loss_critic = critic_for_fakes_mean - critic_for_reals_mean

            #  Update the Critic
            optimizerC.step()

        ## Training the Generator
        for p in netC.parameters():
            p.requires_grad = False
        netG.zero_grad()

        noise = torch.randn(b_size, 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) == 0:
            print("[epoch=%d/%d   i=%4d]     loss_critic=%7.4f   loss_gen=%7.4f   Wasserstein_dist=%7.4f" %  \
                    (epoch,epochs,i,loss_critic.data[0], loss_gen.data[0], wasser_dist.data[0]))

        Gen_losses.append(loss_gen.data[0].item())
        Cri_losses.append(loss_critic.data[0].item())

        if (iters % display_interval == 0) or ((epoch == epochs-1) and (i == len(data_loader)-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 Gen_losses and Cri_losses:
plt.figure(figsize=(10,5))
plt.title("Generator and Critic Loss During Training")
plt.plot(Gen_losses,label="G")
plt.plot(Cri_losses,label="C")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.savefig(dir_name_for_results + "gen_and_critic_loss_training.png")
plt.show()

#  Create GIF
images = []
for imgobj in img_list:
    img = tvt.ToPILImage()(imgobj)
    images.append(img)
imageio.mimsave(dir_name_for_results + "generation_animation.gif", images, fps=5)

# Compare Real and Fake Images
real_batch = next(iter(data_loader))
```

```
    real_batch = real_batch
    plt.figure(figsize=(15,15))
    plt.subplot(1,2,1)
    plt.axis("off")
    plt.title("Real Images")
    plt.imshow(np.transpose(torchvision.utils.make_grid(real_batch.to(device),
                                        padding=1, pad_value=1, normalize=True).cpu(),(1,2,0)))
    plt.subplot(1,2,2)
    plt.axis("off")
    plt.title("Fake Images")
    plt.imshow(np.transpose(img_list[-1],(1,2,0)))
    plt.savefig(dir_name_for_results + "real_vs_fake_images.png")
    plt.show()

    # Save model weights
    checkpoint_path_G = os.path.join('/content/drive/MyDrive/Purdue/ECE60146/HW7/saved_models',
                        f'{model_name}_G.pt')
    torch.save(netG.state_dict(), checkpoint_path_G)
    checkpoint_path_C = os.path.join('/content/drive/MyDrive/Purdue/ECE60146/HW7/saved_models',
                                f'{model_name}_C.pt')
    torch.save(netC.state_dict(), checkpoint_path_C)

    return Cri_losses, Gen_losses, img_list

# Training WGAN
net2_losses = train_wgan(device, netC, netG, data_loader = train_data_loader,
                         model_name = 'HW7_WGAN', dir_name_for_results =
                          '/content/drive/MyDrive/Purdue/ECE60146/HW7/data/wgan',
                         epochs=epochs, display_interval = display_interval)

# Validate WGAN
save_path = '/content/drive/MyDrive/Purdue/ECE60146/HW7/saved_models/HW7_WGAN_G.pt'
validate_gan(device, netG,'/content/drive/MyDrive/Purdue/ECE60146/HW7/data/eval', \
               '/content/drive/MyDrive/Purdue/ECE60146/HW7/data/wgan/images',
             model_name = 'WGAN', model_path = save_path)

# Different Routine to train a WGAN
def train_wgan_2(device, critic, generator, data_loader, model_name,
                dir_name_for_results, epochs = 10, display_interval = 100):

    nz = 100

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

    batch_size = 64

    fixed_noise = torch.randn(batch_size, nz, 1, 1, device=device)

    #  Adam optimizers for the Critic and the Generator:
    optimizerC = optim.Adam(netC.parameters(), lr=0.0002, betas=(0.5, 0.999))
    optimizerG = optim.Adam(netG.parameters(), lr=0.0002, betas=(0.5, 0.999))

    img_list = []
    Gen_losses = []
    Cri_losses = []

    ncritic = 5
    iters = 0
    gen_iterations = 0
    print("\n\nStarting Training Loop.......\n\n")
    clipping_thresh = 0.02
    # For each epoch
    for epoch in range(epochs):

        for i, data in enumerate(data_loader):
            b_size = data.shape[0]
            data = data.to(device)

            z = torch.randn(b_size, nz, 1, 1).to(device)
            fake = netG(z)

            #Train critic function
            for p in netC.parameters():
                p.requires_grad = True
            netC.zero_grad()
```

```
        real_critic = fwx = netC(data)
        fake_critic = fwg = netC(fake.detach())

        loss_fw = fwg - fwx
        loss_fw.backward()
        optimizerC.step()

        #weight clipping
        for p in netC.parameters():
            p.data.clamp_(-clipping_thresh, clipping_thresh)

        #Train G
        if (i+1) % ncritic==0:
            for p in netC.parameters():
                p.requires_grad = False
            netG.zero_grad()
            fake = netG(z)
            fwg = netC(fake)

            loss_G = fwg
            loss_G.backward()
            optimizerG.step()

        if (i+1)%display_interval == 0:
            print('Epoch [{}/{}], Step [{}/{}], critic_loss: {:.4f}, g_loss: {:.4f}, fw(x): {:.4f}, fw(G(z)): {:.4f}'
                    .format(epoch, epochs, i+1, len(data_loader), loss_fw.item(),  fake_critic.item(),
                            real_critic.item(), fake_critic.item()))
            Gen_losses.append(fake_critic.item())
            Cri_losses.append(loss_fw.item())
            with torch.no_grad():
                netG.eval()
                fake = netG(fixed_noise).detach().cpu()
                img_list.append(torchvision.utils.make_grid(fake, padding=1, pad_value=1, normalize=True))
                netG.train()

#  At the end of training, make plots from the data in Gen_losses and Cri_losses:
plt.figure(figsize=(10,5))
plt.title("Generator and Critic Loss During Training")
plt.plot(Gen_losses,label="G")
plt.plot(Cri_losses,label="C")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.savefig(dir_name_for_results + "gen_and_critic_loss_training.png")
plt.show()

#  Create GIF
images = []
for imgobj in img_list:
    img = tvt.ToPILImage()(imgobj)
    images.append(img)
imageio.mimsave(dir_name_for_results + "generation_animation.gif", images, fps=5)

# Compare Real and Fake Images
real_batch = next(iter(data_loader))
real_batch = real_batch
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(torchvision.utils.make_grid(real_batch.to(device),
                                padding=1, pad_value=1, normalize=True).cpu(),(1,2,0)))
plt.subplot(1,2,2)
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(img_list[-1],(1,2,0)))
plt.savefig(dir_name_for_results + "real_vs_fake_images.png")
plt.show()

# Save model weights
checkpoint_path_G = os.path.join('/content/drive/MyDrive/Purdue/ECE60146/HW7/saved_models',
                  f'{model_name}_G.pt')
torch.save(netG.state_dict(), checkpoint_path_G)
checkpoint_path_C = os.path.join('/content/drive/MyDrive/Purdue/ECE60146/HW7/saved_models',
                            f'{model_name}_C.pt')
torch.save(netC.state_dict(), checkpoint_path_C)
```

```
    return Cri_losses, Gen_losses, img_list

# Train a WGAN using Method 2
net3_losses = train_wgan_2(device, netC, netG, data_loader = train_data_loader,
                        model_name = 'HW7_WGAN_2', dir_name_for_results =
                         '/content/drive/MyDrive/Purdue/ECE60146/HW7/data/wgan/',
                        epochs=epochs, display_interval = display_interval)

# Validate WGAN with Method 2
save_path = '/content/drive/MyDrive/Purdue/ECE60146/HW7/saved_models/HW7_WGAN_2_G.pt'
validate_gan(device, netG,'/content/drive/MyDrive/Purdue/ECE60146/HW7/data/eval', \
                '/content/drive/MyDrive/Purdue/ECE60146/HW7/data/wgan/images',
             model_name = 'WGAN', model_path = save_path)
```