SPRING 2023 ECE 60146 – Homework 7

Sahithi Kodali – 34789866
kodali1@purdue.edu

3.1 Building and Training GAN

Two GAN networks – one based on Binary Cross-Entropy Loss (BCE) called as BCE-GAN and the other using Wasserstein distance called as W-GAN – have been built being inspired from Prof.Avinash Kak's dcgan_DG1 and wgan_CG1 (based on weight clipping for critic) networks. The model BCE-GAN is trained for 50 epochs and model W-GAN for 100 epochs, with the other parameters explained in the source code.

The plot of loss over training iterations for generator and discriminator in BCE-GAN and W-GAN can be seen in Figures 1, 2 respectively.

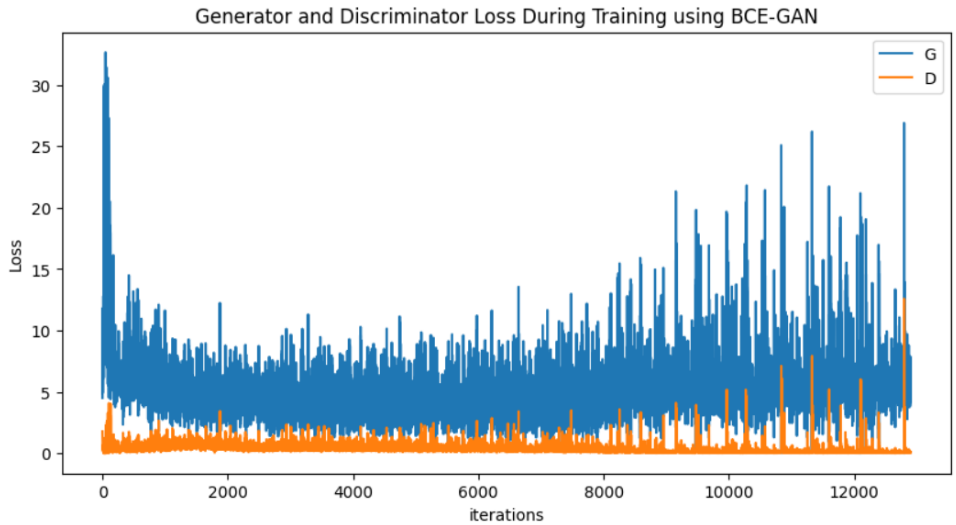

Figure 1: Plot of loss vs iterations for BCE-GAN

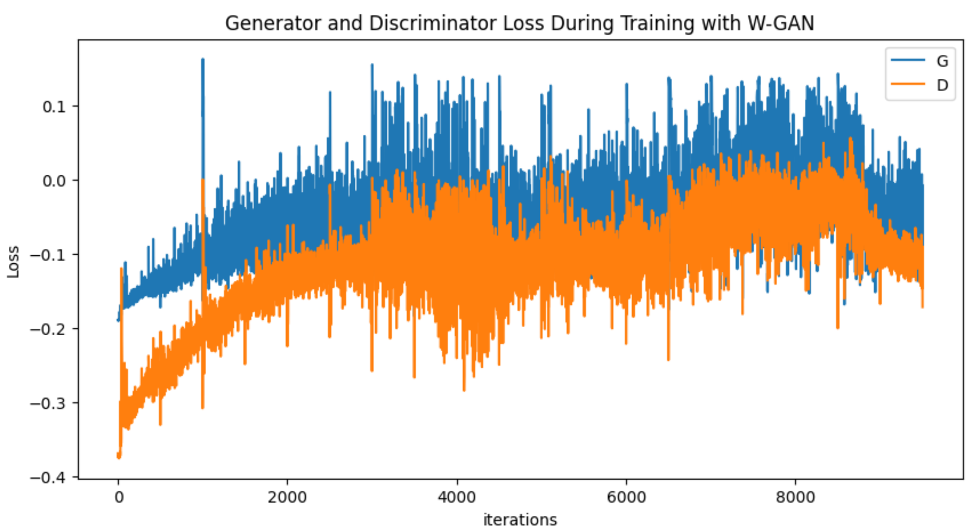

Figure 2: Plot of loss vs iterations for W-GAN

3.2 Evaluating GAN

For evaluating the models, 1000 images and generated from noise using trained generators which are compared to the 1000 evaluation images provided. Qualitative comparison can be done from the display of 4x4 grid images of generated images as shown in Figure 3, 5 using BCE-GAN and W-GAN respectively. For quantitative comparison Fréchet Inception Distance (FID) is used as shown in Figures 4, 6 using BCE-GAN and W-GAN respectively.

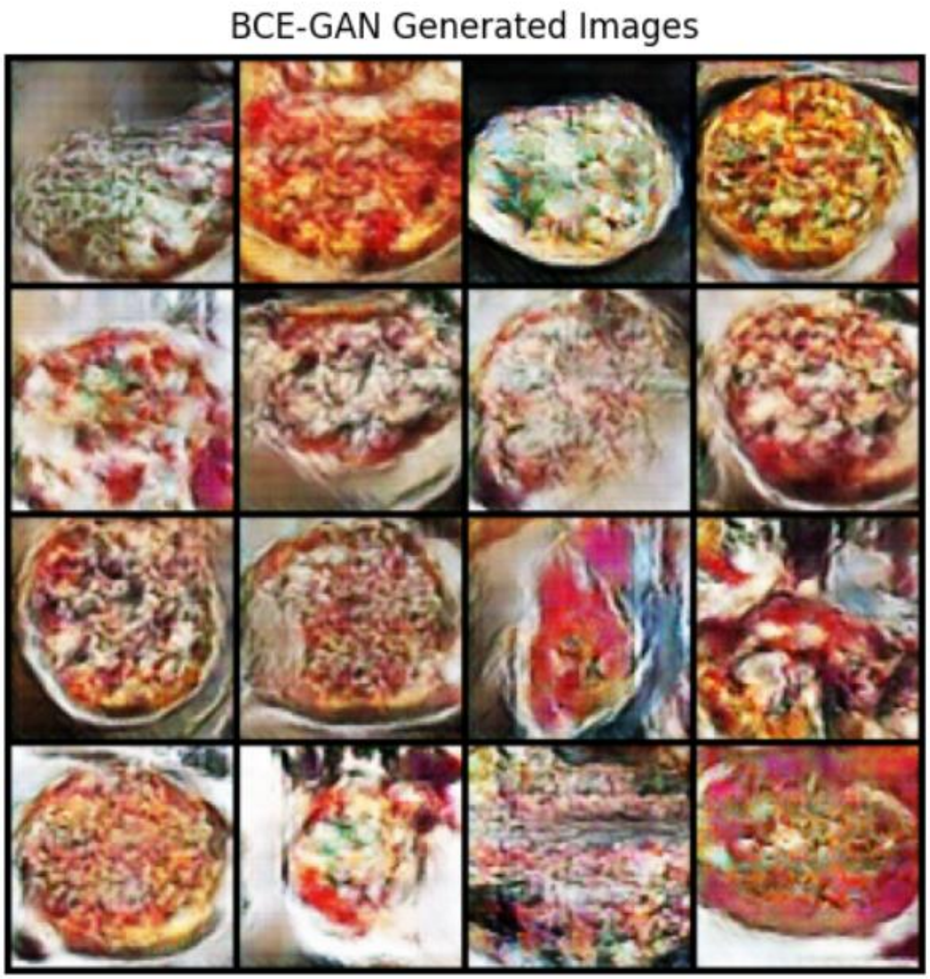

Figure 3: Sample of fake images generated by trained Generator for BCE-GAN

```
100%|██████████| 91.2M/91.2M [00:04<00:00, 21.0MB/s]
100%|██████████| 20/20 [00:25<00:00,  1.25s/it]
100%|██████████| 20/20 [00:04<00:00,  4.68it/s]
FID using BCE-GAN:  149.17
```

Figure 4: FID ~149 attained using BCE-GAN

Figure 5: Sample of fake images generated by trained Generator for W-GAN

```
100%|██████████| 91.2M/91.2M [00:01<00:00, 54.2MB/s]
100%|██████████| 20/20 [00:16<00:00,  1.18it/s]
100%|██████████| 20/20 [00:04<00:00,  4.85it/s]
FID using W-GAN:  226.94
```

Figure 6: FID ~226 attained using W-GAN

<u>Comparing BCE-GAN and W-GAN:</u>

To generate high quality images, W-GAN is usually preferred to BCE-GAN since W-GAN is more stable during training and avoids mode collapse seen in BCE-GAN. W-GAN is implemented using the 1-Lipschitz constraint by enforcing weight clipping/ gradient penalty. Though weight clipping has been used in the network implemented here, the results are not better than BCE-GAN mainly due to the scope of improvement in network for W-GAN implementation and not using the better performance scope with gradient penalty. Hence, in this case BCE-GAN generated better images than W-GAN, but with further improvement in networks and incorporating gradient penalty, W-GAN can give better quality outputs.

SOURCE CODE:

```
# -*- coding: utf-8 -*-
"""HW7_ECE60146ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1fIr83bAslAq36zUDsFGZMrIt76owy3HN
"""

# #extract data for adversarial learning
# !tar -xzf
/content/drive/MyDrive/Purdue/HW7/datasets_for_AdversarialNetworks.tar.gz
# !tar -xzf /content/drive/MyDrive/Purdue/HW7/dataGAN/PurdueShapes5GAN-
20000.tar.gz

# #unzip DLStudio and install
# !tar -xzf /content/drive/MyDrive/Purdue/HW7/DLStudio-2.2.5.tar.gz

# %cd /content/drive/MyDrive/Purdue/HW7/DLStudio-2.2.5

# !pip install .

# #execute the adversarial learning codes to check the working of the networks
# %cd /content/drive/MyDrive/Purdue/HW7/DLStudio-
2.2.5/ExamplesAdversarialLearning

# pip install pymsgbox

# %run 'dcgan_DG1.py'

# %run 'wgan_CG1.py'

# %run 'wgan_CG2.py'

# Commented out IPython magic to ensure Python compatibility.
#import libraries required
# %matplotlib inline
from pycocotools.coco import COCO
```

```python
import numpy as np
import matplotlib.pyplot as plt
import skimage.io as io
import random
import os
import zipfile
from shutil import copyfile

import torch
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from PIL import Image
import pandas as pd
import torchvision
import torchvision.transforms as tvt
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

#create dataset and dataloader
class PizzaDataset(Dataset):

    #intializations
    def __init__(self, root_dir, transform = None):
        super().__init__()
        self.root_dir = root_dir
        self.image_paths = []
        self.transform = transform

        #add image paths and corresponding class as a label
        for img_name in os.listdir(root_dir):
          self.image_paths.append(os.path.join(root_dir,img_name))

    #compute length of dataset
    def __len__(self):
        return len(self.image_paths)

    #apply transformations for the image chosen by index
    def __getitem__(self, index):
        img = Image.open(self.image_paths[index]).convert('RGB')

        if self.transform:
            img = self.transform(img)

        return img
```

```python
batch_size = 32
num_workers = 2

#load data (train and eval)
train_path = '/content/drive/MyDrive/Purdue/HW7/pizzas/train'
eval_path = '/content/drive/MyDrive/Purdue/HW7/pizzas/eval'

transform = tvt.Compose([tvt.ToTensor(), tvt.Normalize((0.5, 0.5, 0.5), (0.5,
0.5, 0.5))])

train_dataset = PizzaDataset(train_path, transform = transform)
eval_dataset = PizzaDataset(eval_path, transform = transform)

#check for the data length
print(len(train_dataset))
print(len(eval_dataset))

#create dataloader
train_data_loader = DataLoader(train_dataset, batch_size = batch_size, shuffle =
True, num_workers=num_workers)
eval_data_loader = DataLoader(eval_dataset, batch_size = batch_size, shuffle =
True, num_workers=num_workers)

#plot images of a batch to check dataloader
images = next(iter(train_data_loader))
plt.figure(figsize=(10,10))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(torchvision.utils.make_grid(images[:], padding=2,
normalize=True),(1,2,0)));

#initialize weights
def weights_init(m):
    """
    Uses the DCGAN initializations for the weights
    """
    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)
```

```
#Generator and Discriminator networks for BCE-GAN (inspired from prof avinash kak
dcgan_DG1 model )

class GeneratorNet(nn.Module):
    def __init__(self):
        super(GeneratorNet, self).__init__()

        # self.fc = nn.Linear(100, 4 * 4 * 512)
        self.conv_transposeN = nn.ConvTranspose2d(100,   512,  kernel_size=4,
stride=1, padding=0, bias=False)
        self.conv_transpose1 = nn.ConvTranspose2d(512, 256, kernel_size=4,
stride=2, padding=1, bias=False)
        self.conv_transpose2 = nn.ConvTranspose2d(256, 128, kernel_size=4,
stride=2, padding=1, bias=False)
        self.conv_transpose3 = nn.ConvTranspose2d(128, 64, kernel_size=4,
stride=2, padding=1, bias=False)
        self.conv_transpose4 = nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2,
padding=1, bias=False)
        self.batch_normN = nn.BatchNorm2d(512)
        self.batch_norm1 = nn.BatchNorm2d(256)
        self.batch_norm2 = nn.BatchNorm2d(128)
        self.batch_norm3 = nn.BatchNorm2d(64)

        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, z):
        # x = self.fc(z)
        # x = x.view(-1, 512, 4, 4)#4x4 image

        #using only convolutional layers
        x = self.relu(self.batch_normN(self.conv_transposeN(z)))#4x4 image
        x = self.relu(self.batch_norm1(self.conv_transpose1(x)))#8x8 image
        x = self.relu(self.batch_norm2(self.conv_transpose2(x)))#16x16 image
        x = self.relu(self.batch_norm3(self.conv_transpose3(x)))#32x32 image
        x = self.tanh(self.conv_transpose4(x))#64x64 image

        return x


class DiscriminatorNet(nn.Module):
    def __init__(self):
        super(DiscriminatorNet, self).__init__()

        self.conv1 = nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1)
```

```
        self.conv2 = nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1)
        self.conv3 = nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1)
        self.conv4 = nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1)
        self.conv5 = nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=0)

        self.batch_norm2 = nn.BatchNorm2d(128)
        self.batch_norm3 = nn.BatchNorm2d(256)
        self.batch_norm4 = nn.BatchNorm2d(512)

        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = torch.nn.functional.leaky_relu(self.conv1(x), negative_slope=0.2,
inplace=True)
        x = self.batch_norm2(self.conv2(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.batch_norm3(self.conv3(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.batch_norm4(self.conv4(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.conv5(x)
        x = self.sigmoid(x)
        return x

#initialize network and check stats
GNet = GeneratorNet()
number_of_learnable_params = sum(p.numel() for p in GNet.parameters() if
p.requires_grad)
num_layers = len(list(GNet.parameters()))
print("\nThe number of layers in G: %d" % num_layers)
print("\nThe number of learnable parameters in G: %d" %
number_of_learnable_params)

#initialize network and check stats
DNet = DiscriminatorNet()
number_of_learnable_params = sum(p.numel() for p in DNet.parameters() if
p.requires_grad)
num_layers = len(list(DNet.parameters()))
print("\nThe number of layers in D: %d" % num_layers)
print("\nThe number of learnable parameters in D: %d" %
number_of_learnable_params)

# Set device to GPU if available, otherwise use CPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
```

```python
#check model summary
from torchsummary import summary
DNet.to(device)

input_tensor = torch.randn(3, 64, 64).to(device)
summary(DNet, input_tensor.shape)

#Training

# Set hyperparameters
lr = 0.0002
epochs = 50
nz_dim = 100

# Initialize generator and discriminator networks
GNet = GNet.to(device)
DNet = DNet.to(device)

DNet.apply(weights_init)
GNet.apply(weights_init)

# Define binary cross-entropy loss function
bce_loss = nn.BCELoss()

# Define optimizer for generator and discriminator
optimizer_g = optim.Adam(GNet.parameters(), lr=lr, betas=(0.5, 0.999))
optimizer_d = optim.Adam(DNet.parameters(), lr=lr, betas=(0.5, 0.999))

#fixed noise
fixed_noise = torch.randn(batch_size, nz_dim, 1, 1, device=device)

#  Establish convention for real and fake labels during training
real_label = 1
fake_label = 0

#initialize lists to store results
img_list = []
G_losses = []
D_losses = []
iters = 0
print("\n\nStarting Training Loop...\n\n")

# Train the GAN
for epoch in range(epochs):
```

```
  g_losses_per_print_cycle = []
  d_losses_per_print_cycle = []

  for i, data in enumerate(train_data_loader):

    # Train discriminator with real images
    DNet.zero_grad()
    real_images_in_batch = data.to(device)
    b_size = real_images_in_batch.size(0)
    # print(b_size)
    label = torch.full((b_size,), real_label, dtype=torch.float, device=device)

    output = DNet(real_images_in_batch).view(-1)
    lossD_for_reals = bce_loss(output,
label)
    lossD_for_reals.backward()

    # Train discriminator with fake images generated by generator
    noise = torch.randn(b_size, nz_dim, 1, 1, device=device)
    fakes = GNet(noise)
    label.fill_(fake_label)

    output = DNet(fakes.detach()).view(-1)
    lossD_for_fakes = bce_loss(output, label)
    lossD_for_fakes.backward()

    lossD = lossD_for_reals + lossD_for_fakes
    d_losses_per_print_cycle.append(lossD)

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


    # Train generator to fool discriminator
    GNet.zero_grad()
    label.fill_(real_label)
    output = DNet(fakes).view(-1)
    lossG = bce_loss(output, label)
    g_losses_per_print_cycle.append(lossG)
    lossG.backward()

    # Update generator parameters
    optimizer_g.step()

    # Print losses at end of each epoch and append losses
```

```
    if (i+1) % 100 == 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))
      print ("[ epoch : %d/%d, iter : %5d] 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())

    #append fake images obtained by generator
    if (i == len(train_data_loader)-1):
      with torch.no_grad():
        fake = GNet(fixed_noise).detach().cpu()
      img_list.append(torchvision.utils.make_grid(fake, padding=1, pad_value=1,
normalize=True))
    iters += 1


# Save generator and discriminator models
torch.save(GNet.state_dict(),
"/content/drive/MyDrive/Purdue/HW7/generator_bce_50epoch.pth")
torch.save(DNet.state_dict(),
"/content/drive/MyDrive/Purdue/HW7/discriminator_bce_50epoch.pth")

#plotting loss vs iterations for BCE-GAN
plt.figure(figsize=(10,5))

plt.title("Generator and Discriminator Loss During Training using BCE-GAN")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.savefig('/content/drive/MyDrive/Purdue/HW7/gen_and_disc_loss_training_50epoch
_bce.png')
plt.show()

#plotting real vs fake images
real_batch = next(iter(train_data_loader))
```

```
# real_batch = real_batch[0]
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('/content/drive/MyDrive/Purdue/HW7/real_vs_fake_images.png')

plt.show()

!pip install pytorch-fid

from torchvision.utils import save_image

#Evaluation for BCE-GAN

# Generate 1,000 fake pizza images
num_images = 1000

with torch.no_grad():
    noise = torch.randn(num_images, nz_dim, 1, 1, device=device)
    fake_images = GNet(noise)

#save fake images
print(fake_images.size())
fake_piz_list = []

for i in range(fake_images.shape[0]):
  img = fake_images[i].detach().cpu()
  fake_piz_list.append(img)
  save_image(img,f'/content/drive/MyDrive/Purdue/HW7/fakes_piz/fake_img_{i}.png')

#get paths for fake images generated and evaluation images
fake_piz_paths = []
```

```
fake_piz_imgs_path = '/content/drive/MyDrive/Purdue/HW7/fakes_piz'
for img_name in os.listdir(fake_piz_imgs_path):
  fake_piz_paths.append(os.path.join(fake_piz_imgs_path,img_name))
print(fake_piz_paths[:5])

real_piz_paths = []

eval_piz_imgs_path = '/content/drive/MyDrive/Purdue/HW7/pizzas/eval'
for img_name in os.listdir(eval_piz_imgs_path):
  real_piz_paths.append(os.path.join(eval_piz_imgs_path,img_name))
print(real_piz_paths[:5])

from pytorch_fid.fid_score import calculate_activation_statistics,
calculate_frechet_distance
from pytorch_fid.inception import InceptionV3

#compute FID score
real_paths = real_piz_paths
fake_paths = fake_piz_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 using BCE-GAN: { fid_value : .2f}')

#plot a sample of 16 fake images generated using BCE-GAN
plot_imgs = fake_images[16:32].detach().cpu()
fig = plt.figure(figsize=(6,6))
plt.axis("off")
plt.title("BCE-GAN Generated Images")
plt.imshow(np.transpose(torchvision.utils.make_grid(plot_imgs, padding=2,
normalize=True, nrow = 4), (1,2,0)))
plt.show()

"""**W-GAN Begins....**"""

#Generator and Discriminator networks (improvised BCE-GAN above to suit scalar
output)
```

```python
class GeneratorNetW(nn.Module):
    def __init__(self):
        super(GeneratorNetW, self).__init__()

        self.conv_transposeN = nn.ConvTranspose2d(100,    512,  kernel_size=4,
stride=1, padding=0, bias=False)
        self.conv_transpose1 = nn.ConvTranspose2d(512, 256, kernel_size=4,
stride=2, padding=1, bias=False)
        self.conv_transpose2 = nn.ConvTranspose2d(256, 128, kernel_size=4,
stride=2, padding=1, bias=False)
        self.conv_transpose3 = nn.ConvTranspose2d(128, 64, kernel_size=4,
stride=2, padding=1, bias=False)
        self.conv_transpose4 = nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2,
padding=1, bias=False)
        self.batch_normN = nn.BatchNorm2d(512)
        self.batch_norm1 = nn.BatchNorm2d(256)
        self.batch_norm2 = nn.BatchNorm2d(128)
        self.batch_norm3 = nn.BatchNorm2d(64)

        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, z):

        x = self.relu(self.batch_normN(self.conv_transposeN(z)))#4x4 image
        x = self.relu(self.batch_norm1(self.conv_transpose1(x)))#8x8 image
        x = self.relu(self.batch_norm2(self.conv_transpose2(x)))#16x16 image
        x = self.relu(self.batch_norm3(self.conv_transpose3(x)))#32x32 image
        x = self.tanh(self.conv_transpose4(x))#64x64 image

        return x


class DiscriminatorNetW(nn.Module):
    def __init__(self):
        super(DiscriminatorNetW, self).__init__()

        self.conv1 = nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1)
        self.conv3 = nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1)
        self.conv4 = nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1)
        # self.conv5 = nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=0)

        self.batch_norm2 = nn.BatchNorm2d(128)
```

```
        self.batch_norm3 = nn.BatchNorm2d(256)
        self.batch_norm4 = nn.BatchNorm2d(512)

        self.fc_final = nn.Linear(512*4*4,1)

    def forward(self, x):
        x = torch.nn.functional.leaky_relu(self.conv1(x), negative_slope=0.2,
inplace=True)
        x = self.batch_norm2(self.conv2(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.batch_norm3(self.conv3(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.conv4(x)

        # x = self.conv5(x)
        # print(x.shape)

        x = x.view(-1, 512*4*4)
        x = self.fc_final(x)
        x = x.mean(0)
        x = x.view(1)
        # print(x.shape)
        return x

#initialize network and check stats
WGNet = GeneratorNetW()
number_of_learnable_params = sum(p.numel() for p in WGNet.parameters() if
p.requires_grad)
num_layers = len(list(WGNet.parameters()))
print("\nThe number of layers in G: %d" % num_layers)
print("\nThe number of learnable parameters in G: %d" %
number_of_learnable_params)

#initialize network and check stats
WDNet = DiscriminatorNetW()
number_of_learnable_params = sum(p.numel() for p in WDNet.parameters() if
p.requires_grad)
num_layers = len(list(WDNet.parameters()))
print("\nThe number of layers in D: %d" % num_layers)
print("\nThe number of learnable parameters in D: %d" %
number_of_learnable_params)

#check model summary
from torchsummary import summary
WDNet.to(device)
```

```python
input_tensor = torch.randn(3, 64, 64).to(device)
summary(WDNet, input_tensor.shape)

#Training with W-Net

# Set hyperparameters
lr = 0.0002
epochs = 100
nz_dim = 100
clip_thresh = 0.005

# Initialize generator and discriminator networks
WGNet = WGNet.to(device)
WDNet = WDNet.to(device)

WDNet.apply(weights_init)
WGNet.apply(weights_init)

# Define optimizer for generator and discriminator
optimizer_gW = optim.Adam(WGNet.parameters(), lr=lr, betas=(0.5, 0.999))
optimizer_dW = optim.Adam(WDNet.parameters(), lr=lr, betas=(0.5, 0.999))

#fixed noise
fixed_noise = torch.randn(batch_size, nz_dim, 1, 1, device=device)

#  Establish convention for real and fake labels during training
one = torch.FloatTensor([1]).to(device)
minus_one = torch.FloatTensor([-1]).to(device)

#initialize lists to store results
Wimg_list = []
WG_losses = []
WD_losses = []
iters = 0
gen_iters = 0
print("\n\nStarting Training Loop...\n\n")

# Train the GAN
for epoch in range(epochs):
  data_iter = iter(train_data_loader)

  i = 0
  ncritic = 5
```

```
  while i < len(train_data_loader):
    for p in WDNet.parameters():
      p.requires_grad = True
    if gen_iters < 25 or gen_iters % 500 == 0:     # the choices 25 and 500 are
from WGAN
      ncritic = 50
    ic = 0

    #inner while loop
    while ic < ncritic and i < len(train_data_loader):
      ic += 1
      for p in WDNet.parameters():
        p.data.clamp_(-clip_thresh, clip_thresh)

      # Train discriminator with real images with traget -1
      WDNet.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)
      # print(real_images_in_batch.size())

      critic_for_reals_mean = WDNet(real_images_in_batch)
      critic_for_reals_mean.backward(minus_one)

      # Train discriminator with fake images generated by generator with target 1
      noise = torch.randn(b_size, nz_dim, 1, 1, device=device)
      fakes = WGNet(noise)
      critic_for_fakes_mean = WDNet(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

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


    # Train generator to fool discriminator
    for p in WDNet.parameters():
      p.requires_grad = False
    WGNet.zero_grad()
    noise = torch.randn(b_size, nz_dim, 1, 1, device=device)
    fakes = WGNet(noise)
```

```
    critic_for_fakes_mean = WDNet(fakes)
    loss_gen = critic_for_fakes_mean
    critic_for_fakes_mean.backward(minus_one)

    # Update generator parameters
    optimizer_gW.step()


    gen_iters += 1


    # Print losses at end of each epoch and append losses
    if i % (ncritic * 20) == 0:
      print ("[ epoch : %d/%d, iter : %5d] loss_critic : %7.4f  loss_gen :
%7.4f  wasser_dist : %7.4f" % ((epoch), epochs, (i), loss_critic.data[0],
loss_gen.data[0], wasser_dist.data[0]))

    WG_losses.append(loss_gen.data[0].item())

    WD_losses.append(loss_critic.data[0].item())

    #append fake images obtained by generator
    if (iters % 100 == 0) or ((epoch == epochs-1) and (i ==
len(train_data_loader)-1)):
      with torch.no_grad():
        fake = WGNet(fixed_noise).detach().cpu()
      Wimg_list.append(torchvision.utils.make_grid(fake, padding=1, pad_value=1,
normalize=True))
    iters += 1


# Save generator and discriminator models
torch.save(WGNet.state_dict(),
"/content/drive/MyDrive/Purdue/HW7/generator_wass.pth")
torch.save(WDNet.state_dict(),
"/content/drive/MyDrive/Purdue/HW7/discriminator_wass.pth")

#plotting loss vs iterations for W-GAN
plt.figure(figsize=(10,5))

plt.title("Generator and Discriminator Loss During Training with W-GAN")
plt.plot(WG_losses,label="G")
plt.plot(WD_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
```

```
plt.legend()
#
plt.savefig('/content/drive/MyDrive/Purdue/HW7/gen_and_disc_loss_training_WGAN.pn
g')
plt.show()

#Evaluation for W-GAN

# Generate 1,000 fake pizza images
num_images = 1000

with torch.no_grad():
    noise = torch.randn(num_images, nz_dim, 1, 1, device=device)
    fake_images_WGAN = WGNet(noise)

#save fake images
print(fake_images_WGAN.size())
fake_piz_list_WGAN = []

for i in range(fake_images_WGAN.shape[0]):
  img = fake_images_WGAN[i].detach().cpu()
  fake_piz_list_WGAN.append(img)
  save_image(img,f'/content/drive/MyDrive/Purdue/HW7/fake_pizzs_WGAN/fake_img_{i}
.png')

#get paths for fake images generated and evaluation images
fake_piz_paths = []

fake_piz_imgs_path = '/content/drive/MyDrive/Purdue/HW7/fake_pizzs_WGAN'
for img_name in os.listdir(fake_piz_imgs_path):
  fake_piz_paths.append(os.path.join(fake_piz_imgs_path,img_name))
print(fake_piz_paths[:5])

real_piz_paths = []

eval_piz_imgs_path = '/content/drive/MyDrive/Purdue/HW7/pizzas/eval'
for img_name in os.listdir(eval_piz_imgs_path):
  real_piz_paths.append(os.path.join(eval_piz_imgs_path,img_name))
print(real_piz_paths[:5])

#compute FID score
real_paths = real_piz_paths
fake_paths = fake_piz_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 using W-GAN: { fid_value : .2f}')

#plot a sample of 16 fake images generated using W-GAN
plot_imgs = fake_images_WGAN[32+16:32+16+16].detach().cpu()
fig = plt.figure(figsize=(6,6))
plt.axis("off")
plt.title("W-GAN Generated Images")
plt.imshow(np.transpose(torchvision.utils.make_grid(plot_imgs, padding=2,
normalize=True, nrow = 4), (1,2,0)))
plt.show()
```

******