ECE60146: Homework 7

Submission By: Joseph Wang
wang3450@purdue.edu

Prepared for Dr. Avinash Kak and Dr. Charles Bouman
GTA: Fangda Li and Qiuchen Zhai
Purdue University
April 5, 2023

Contents

List of Figures

1 Introduction

The main goal of this homework assignment is to create our own pizza-generating generative adversarial networks (GANs). The learning objectives are as follows:

1. Understanding how the generator and discriminator networks are trained to compete against each other in a min max game.
2. Experiment with two different GAN loss functions. Binary Cross-Entropy (BCE) and Wasserstein Distance.
3. Evaluate generated images qualitatively via. inspection and quantitatively using the Frechet Inception Distance (FID).

The data set we used was supplied by Fangda Li, which contains roughly $8,000$ training samples and exactly $1,000$ evaluation samples. Figure 1 below depicts a few images from our training set.

Figure 1: Sample Images from Pizza Set

2 Building Our GANs

BCE GAN was implemented with two networks: a generator and a discriminator. The generator can be thought of as a function $G(z, \theta_g)$ that maps random noise vectors to images that we want to

look like the images in our training data. On the other hand, the generator can be thought of as a function $D(x, \theta_d)$ that returns the probability that the input x is from the probability distribution that describes the training data.

In our implemenation, the generator accepts a $100 \times 1 \times 1$ noise vector and sequentially up samples it by a factor of 2 until we arrive as a $3 \times 64 \times 64$ image tensor. Our discriminator contrarily takes a $3 \times 64 \times 64$ image tensor and down samples it until we arrive at its latent space representation.

WGAN was also implemented with two networks: a generator and a critic. The generator in our WGAN is the same generator as found in BCE GAN. The critic implements a 1-Lipschitz function – because, unlike what was the case for the discriminator, the job of the critic is not to accept or reject what is produced by the Generator, but to become adept at estimating the Wasserstein distance between the distribution that corresponds to the training dataset and the distribution that has been learned by the generator so far.

Shown below in Figure 2 are block diagram representations of all three networks used in this assignment.

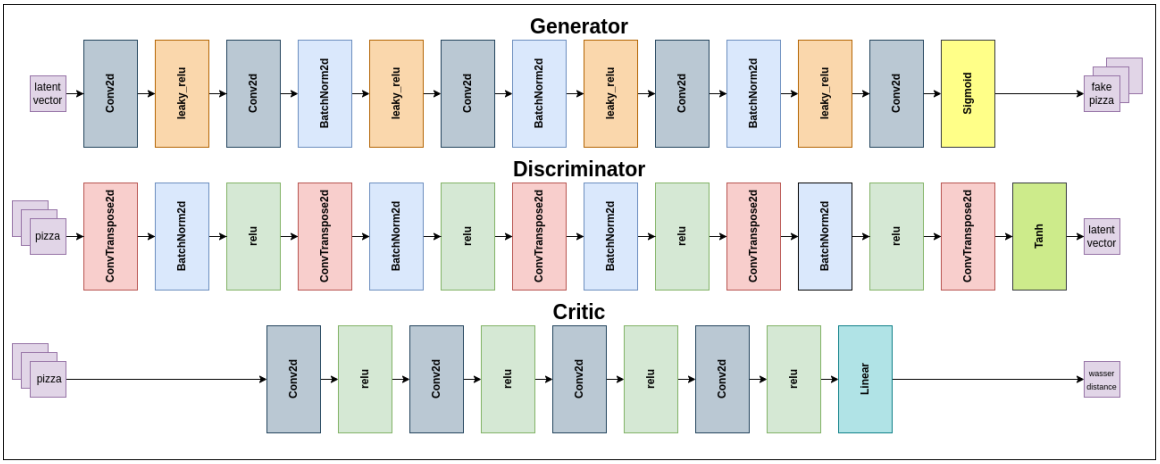


Figure 2: Generator, Discriminator, and Critic Block Diagrams

3 Training BCE GAN

Training BCE GAN boils down to solving the following min max equation (1):

$$\min_{\theta_g} \max_{\theta_d} [E_{x \sim p_{data}}[logD(x)] + E_{z \sim p_z}[log(1 - D(G(z))]] \tag{1}$$

The discriminator is trained to maximize the probability of assigning the correct label to an input image that looks like it came from the same distribution as the training data. Thus, for discriminator training, we want the parameters θ_d to maximize the following expectation (2):

$$\max_{\theta_d} E_{x \sim P_{data}}[logD(x))] \tag{2}$$

The generator is trained to transform a noise vector into an image that should look like a pizza from the training set. Thus, for generator training, we want the parameters θ_g to minimize the following expectation (3):

$$\min_{\theta_g} E_{z\sim p_z}[log(1 - D(G(z))] \tag{3}$$

Equations (2) and (3) can be written together to produce the combined optimization seen in equation (1).

4 Training WGAN

The main difference in training WGAN is the idea that we train a generator G that minimizes the Wasserstein distance between the true distribution P and its learned approximation Q. At the same time, the WGAN must also discover a critic C that seeks to maximize the same distance. An expression for the Wasserstein distance function is shown below in (4).

$$d_W(P_r, P_\theta) = \sup_{||f||_L \leq 1} [E_{x\sim P_r}[f_w(x)] - E_{x\sim P_z}[f_w(G_\theta(z))]] \tag{4}$$

What this equation says is that there exists a 1-Lipschitz function, $f()$, for which the maximization shown on the right in (4) yields the value for the Wasserstein distance. Thus the min max objective for our WGAN is:

$$\min_G \max_C [E_{x\sim P[C(x)] - E_{z\sim p_z}[C(G(z))]}] \tag{5}$$

In order to enforce the 1-Lipschitz constraint of $f()$ we further added a gradient penalty term to the critic loss shown below in (6).

$$C_{loss} = E_{z\sim p_z}[C(G(z))] - E_{x\sim p}[C(x)] + \lambda[||\nabla_{\hat{x}} C(\hat{x})||^2 - 1]^2 \tag{6}$$

Our WGAN training logic borrows Marvin Cao's source code for a PyTorch implementation of computing the gradient penalty.

5 Evaluation of Our GANs

We quantitatively evaluated the performance of both our GANs using the Frechet Inception Distance (FID). Originally proposed by Heusel et al, FID is a widely used metric for measuring both the quality and the diversity of GAN-generated images. More specifically, it does so by measuring how close the distribution of the fake images is to the distribution of the real images. To calculate the FID metric for both GANs, we first encoded both the set of real and fake images into feature vectors using a pretrained Inception Network. Then we modeled the resulting distribution of feature vectors with a multivariate Gaussian. The Frechet distance between the two multivariate Gaussians is our FID.

6 Results

Shown below in Figures 3 and 4 are the training losses for BCE GAN and WGAN respectively. An important note to consider when examining these loss plots is that the loss axis for BCE GAN has unit scale whereas the loss axis for WGAN has a scale of 1×10^8. In both plots, the generator's loss is the blue signal while the discriminator/critic's loss is the orange signal.

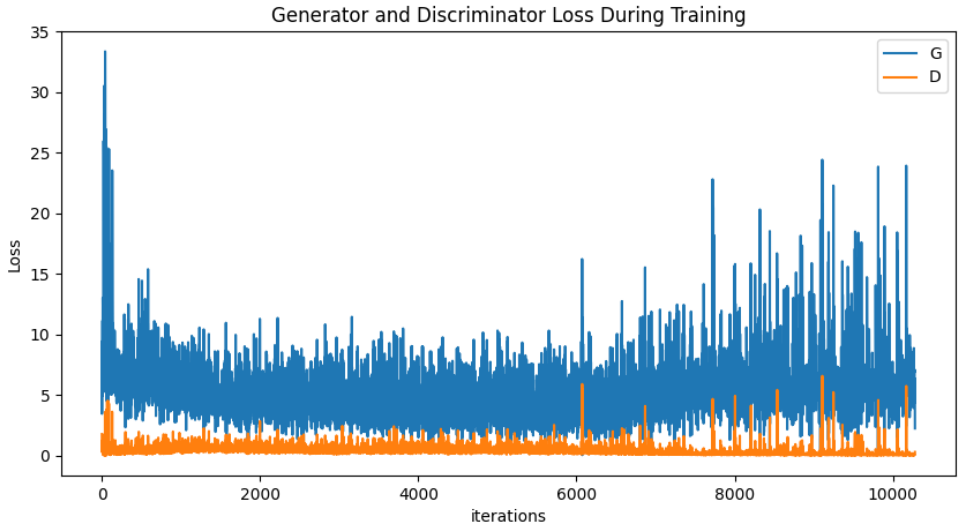


Figure 3: BCE GAN Training Loss

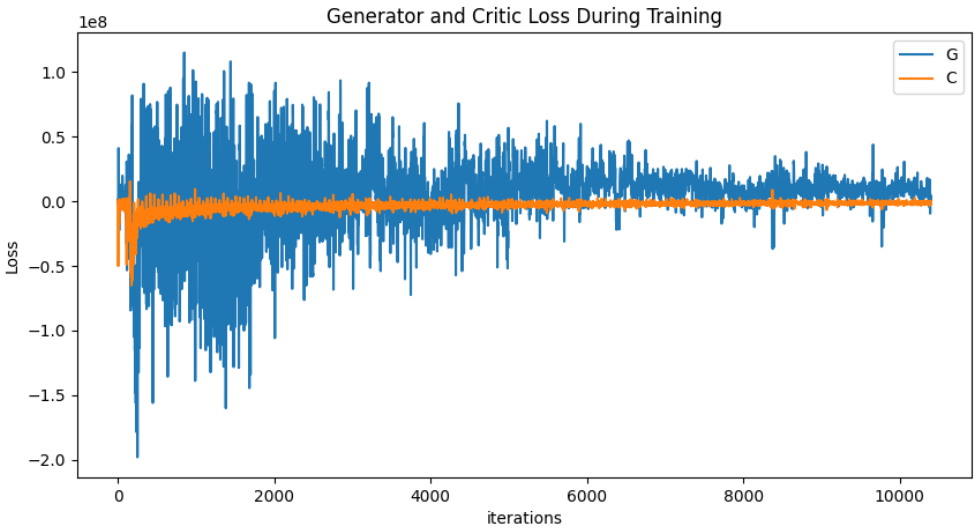


Figure 4: WGAN Training Loss

Shown below in Figures 5 and 6 are samples of fake images produced by our BCE GAN and WGAN respectively.

Figure 5: BCE GAN Fake Samples

Figure 6: WGAN Fake Samples

Shown below in Table 1 are the FID scores for our BCE GAN and WGAN.

Table 1: FID for BCE GAN and WGAN

GAN Type	FID
BCE GAN	136.75
WGAN	129.87

7 Discussion of Results

When qualitatively and quantitatively examining the performance of both GAN architectures, we come to the conclusion that on the pizza data set, both models perform roughly as well as the other. From a qualitative perspective, WGAN seems to slightly outperform BCE GAN. More specifically, BCE GAN generated fakes seem to contain more checkerboard artifacts than fakes generated by WGAN. This observation is puzzling from an intuition stand point since both generators have the same architecture and use `nn.ConvTranspose2d` to up sample the noise vector into a color image. The most likely source of these artifacts is in the adversarial training of the networks, however, at the moment we are unable to deterministically identify the source. Similarly from a quantitative perspective, WGAN slightly edges out BCE GAN as the FID score for fakes generated by WGAN have a smaller FID score compared to the control group of real images.

8 Appendix

What follows is the source code for replicating the results found in the writing. The model definitions can be found in Section 8.1. Utility classes and methods (i.e. dataset class, train methods, etc) can be found in Section 8.2. Finally, notebooks to drive both BCE GAN and WGAN pipelines can be found in the attached pdfs following Section 8.2 starting on page 19. All source code can also be found on github here.

8.1 Source Code For Network Models

```
import sys,os,os.path
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as tvt
import torchvision.transforms.functional as tvtF
import torch.optim as optim
import numpy as np
import math
import random
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import time
import glob
import imageio

class DS_SkipBlock(nn.Module):
    """Skip-Connection building block to be used in Discriminator.
    Input: in_ch and out_ch specify the input and output channels
    Output: Downsamples the input tensor by a factor of 2 using nn.Conv2d()
    """
    def __init__(self, in_ch, out_ch, skip_connections=True):
        super(DS_SkipBlock, self).__init__()
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch

        # performs downsampling here
        # input shape [in_ch, H_in, W_in]
        # output shape [out_ch, H_in / 2, W_in / 2]
        self.conv = nn.Conv2d(in_ch, out_ch, kernel_size=4, stride=2, padding=1)

        self.bn = nn.BatchNorm2d(out_ch)

        if out_ch == 1:
            self.flag = True
            self.sig = nn.Sigmoid()
            self.conv = nn.Conv2d(in_ch, out_ch, kernel_size=4, stride=2, padding=0)
        else:
            self.flag = False

    def forward(self, x):
        identity = x
        out = self.conv(x)
        identity = self.conv(identity)
        if not self.flag:
            out = self.bn(out)
            out = torch.nn.functional.leaky_relu(out, negative_slope=0.2, inplace=
    True)
        elif self.flag:
            out = self.sig(out)
        if self.skip_connections:
            out = torch.cat((out[:,:self.in_ch,:,:] + identity[:,:self.in_ch,:,:],
    out[:,self.in_ch:,:,:] + identity[:,self.in_ch:,:,:]), dim=1)
        return out

class SkipBlock_Discriminator(nn.Module):
```

```
57      """This is an implemenation of the DCGAN Discriminator. The discriminator takes a
         64 x 64 color image
58      and transforms it to latent space rep.of shape [1, 1, 1]
59      """
60      def __init__(self):
61          super(SkipBlock_Discriminator, self).__init__()
62          self.conv_in = nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1)
63          self.skip_64_128 = DS_SkipBlock(64, 128)
64          self.skip_128_256 = DS_SkipBlock(128, 256)
65          self.skip_256_512 = DS_SkipBlock(256, 512)
66          self.skip_512_1 = DS_SkipBlock(512, 1)
67
68      def forward(self,x):
69          # [b, 3, 64, 64] -> [b, 64, 32, 32]
70          x = torch.nn.functional.leaky_relu(self.conv_in(x), negative_slope=0.2,
        inplace=True)
71
72          # [b, 64, 32, 32] -> [b, 128, 16, 16]
73          x = self.skip_64_128(x)
74
75          # [b, 128, 16, 16] -> [b, 256, 8, 8]
76          x = self.skip_128_256(x)
77
78          # [b, 256, 8, 8] -> [b, 512, 4, 4]
79          x = self.skip_256_512(x)
80
81          # [b, 512, 4, 4] -> [b, 1, 1, 1]
82          x = self.skip_512_1(x)
83          return x
84
85  class US_SkipBlock(nn.Module):
86      """Skip-Connection building block to be used in Generator.
87      Input: in_ch and out_ch specify the input and output channels
88      Output: Upsamples the input tensor by a factor of 2 using nn.ConvTranspose2d()
89      """
90      def __init__(self, in_ch, out_ch, skip_connections=True):
91          super(US_SkipBlock, self).__init__()
92          self.skip_connections = skip_connections
93          self.in_ch = in_ch
94          self.out_ch = out_ch
95
96          self.conv = nn.ConvTranspose2d(in_ch, out_ch, kernel_size=4, stride=2,
        padding=1, bias=False)
97          self.bn = nn.BatchNorm2d(out_ch)
98
99          if in_ch == 100:
100             self.conv = nn.ConvTranspose2d(in_ch, out_ch, kernel_size=4, stride=1,
        padding=0, bias=False)
101         if out_ch == 3:
102             self.flag = True
103             self.tanh = nn.Tanh()
104         else:
105             self.flag = False
106
107     def forward(self,x):
108         identity = x
109         out = self.conv(x)
110         identity = self.conv(identity)
111         if not self.flag:
112             out = self.bn(out)
```

```
113             out = torch.nn.functional.relu(out)
114         elif self.flag:
115             out = self.tanh(out)
116         if self.skip_connections:
117             out = torch.cat((out[:,:self.in_ch,:,:] + identity[:,:self.in_ch,:,:],
    out[:,self.in_ch:,:,:] + identity[:,self.in_ch:,:,:]), dim=1)
118         return out
119
120 class SkipBlock_Generator(nn.Module):
121     """This is an implementation of the DCGAM Generator. The generator takes a 1x1
    noise vector with
122     100 channels, and upsamples it to a 64x64 color image
123     """
124     def __init__(self):
125         super(SkipBlock_Generator, self).__init__()
126         self.latent_to_image = US_SkipBlock(100, 512)
127         self.upsample1 = US_SkipBlock(512, 256)
128         self.upsample2 = US_SkipBlock(256, 128)
129         self.upsample3 = US_SkipBlock(128, 64)
130         self.upsample4 = US_SkipBlock(64, 3)
131
132     def forward(self,z):
133         # [b, 100, 1, 1] -> [b, 512, 4, 4]
134         z = self.latent_to_image(z)
135
136         # [b, 512, 4, 4] -> [b, 256, 8, 8]
137         z = self.upsample1(z)
138
139         # [b, 256, 8, 8] -> [b, 128, 16, 16]
140         z = self.upsample2(z)
141
142         # [b, 128, 16, 16] -> [b, 64, 32, 32]
143         z = self.upsample3(z)
144
145         # [b, 64, 32, 32] -> [b, 3, 64, 64]
146         z = self.upsample4(z)
147         return z
148
149
150 class Discriminator(nn.Module):
151     def __init__(self):
152         super(Discriminator, self).__init__()
153         self.conv_in = nn.Conv2d(  3,    64,      kernel_size=4,      stride=2,
    padding=1)
154         self.conv_in2 = nn.Conv2d( 64,   128,     kernel_size=4,      stride=2,
    padding=1)
155         self.conv_in3 = nn.Conv2d( 128,  256,     kernel_size=4,      stride=2,
    padding=1)
156         self.conv_in4 = nn.Conv2d( 256,  512,     kernel_size=4,      stride=2,
    padding=1)
157         self.conv_in5 = nn.Conv2d( 512,  1,       kernel_size=4,      stride=1,
    padding=0)
158         self.bn1  = nn.BatchNorm2d(128)
159         self.bn2  = nn.BatchNorm2d(256)
160         self.bn3  = nn.BatchNorm2d(512)
161         self.sig = nn.Sigmoid()
162     def forward(self, x):
163         x = torch.nn.functional.leaky_relu(self.conv_in(x), negative_slope=0.2,
    inplace=True)
164         x = self.bn1(self.conv_in2(x))
```

```
165         x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
166         x = self.bn2(self.conv_in3(x))
167         x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
168         x = self.bn3(self.conv_in4(x))
169         x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
170         x = self.conv_in5(x)
171         x = self.sig(x)
172         return x
173
174 class Generator(nn.Module):
175     def __init__(self):
176         super(Generator, self).__init__()
177         self.latent_to_image = nn.ConvTranspose2d( 100,   512,  kernel_size=4, stride
    =1, padding=0, bias=False)
178         self.upsampler2 = nn.ConvTranspose2d( 512, 256, kernel_size=4, stride=2,
    padding=1, bias=False)
179         self.upsampler3 = nn.ConvTranspose2d (256, 128, kernel_size=4, stride=2,
    padding=1, bias=False)
180         self.upsampler4 = nn.ConvTranspose2d (128, 64,  kernel_size=4, stride=2,
    padding=1, bias=False)
181         self.upsampler5 = nn.ConvTranspose2d(  64,  3,  kernel_size=4, stride=2,
    padding=1, bias=False)
182         self.bn1 = nn.BatchNorm2d(512)
183         self.bn2 = nn.BatchNorm2d(256)
184         self.bn3 = nn.BatchNorm2d(128)
185         self.bn4 = nn.BatchNorm2d(64)
186         self.tanh  = nn.Tanh()
187
188     def forward(self, x):
189         x = self.latent_to_image(x)
190         x = torch.nn.functional.relu(self.bn1(x))
191         x = self.upsampler2(x)
192         x = torch.nn.functional.relu(self.bn2(x))
193         x = self.upsampler3(x)
194         x = torch.nn.functional.relu(self.bn3(x))
195         x = self.upsampler4(x)
196         x = torch.nn.functional.relu(self.bn4(x))
197         x = self.upsampler5(x)
198         x = self.tanh(x)
199         return x
200
201 class SkipBlockDN(nn.Module):
202     def __init__(self, in_ch, out_ch, downsample=False, skip_connections=True):
203         super(SkipBlockDN, self).__init__()
204         self.downsample = downsample
205         self.skip_connections = skip_connections
206         self.in_ch = in_ch
207         self.out_ch = out_ch
208         self.convo1 = nn.Conv2d(in_ch, out_ch, kernel_size=3, stride=1, padding=1)
209         self.convo2 = nn.Conv2d(in_ch, out_ch, kernel_size=3, stride=1, padding=1)
210         self.bn1 = nn.BatchNorm2d(out_ch)
211         self.bn2 = nn.BatchNorm2d(out_ch)
212         self.downsampler1 = nn.Conv2d(in_ch, out_ch, kernel_size=1, stride=2)
213         self.downsampler2 = nn.Conv2d(out_ch, out_ch, kernel_size=1, stride=2)
214     def forward(self, x):
215         identity = x
216         out = self.convo1(x)
217         out = self.bn1(out)
218         out = torch.nn.functional.relu(out)
219         if self.in_ch == self.out_ch:
```

```
            out = self.convo2(out)
            out = self.bn2(out)
            out = torch.nn.functional.leaky_relu(out, negative_slope=0.2)
        if self.downsample:
            out = self.downsampler2(out)
            identity = self.downsampler1(identity)
        if self.skip_connections:
            out += identity
        return out

class Critic(nn.Module):
    def __init__(self):
        super(Critic, self).__init__()
        self.conv_in = SkipBlockDN(3, 64, downsample=True, skip_connections=True)
        self.conv_in2 = SkipBlockDN( 64,   128,  downsample=True, skip_connections=False)
        self.conv_in3 = SkipBlockDN(128,   256,  downsample=True, skip_connections=False)
        self.conv_in4 = SkipBlockDN(256,   512,  downsample=True, skip_connections=False)
        self.conv_in5 = SkipBlockDN(512,   1,  downsample=False, skip_connections=False)
        self.bn1  = nn.BatchNorm2d(128)
        self.bn2  = nn.BatchNorm2d(256)
        self.bn3  = nn.BatchNorm2d(512)
        self.final = nn.Linear(512, 1)
        self.final2 = nn.Linear(336, 1)
    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))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.conv_in5(x)
        x = x.view(-1)
        if x.size(0) == 512:
            x = self.final(x)
        else:
            x = self.final2(x)
        x = x.mean(0)
        x = x.view(1)
        return x

class Critic1(nn.Module):
    def __init__(self):
        super(Critic1, self).__init__()
        self.DIM = 64
        self.sequence = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=self.DIM, kernel_size=5, stride=2, padding=2),
            nn.ReLU(True),
            nn.Conv2d(in_channels=self.DIM, out_channels=2*self.DIM, kernel_size=5, stride=2, padding=2),
            nn.ReLU(True),
            nn.Conv2d(in_channels=2*self.DIM, out_channels=4*self.DIM, kernel_size=5, stride=2, padding=2),
            nn.ReLU(True),
```

```
            nn.Conv2d(in_channels=4*self.DIM, out_channels=8*self.DIM, kernel_size=5,
     stride=2, padding=2),
            nn.ReLU(True)
        )

        self.output = nn.Linear(in_features=4*4*4*self.DIM, out_features=1)

    def forward(self, x):
        x = x.view(-1, 3, 64, 64)
        x = self.sequence(x)
        x = x.view(-1, 4*4*4*self.DIM)
        x = self.output(x)

        assert x.mean(dim=0) == x.mean(0), "Dimensions are not the same"
        x = x.mean(dim=0) # Return only one value as required by the expectation
    operator
        x = x.view(1)
        return x
```

models.py

8.2 Source Code For Utils

```
import sys,os,os.path
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as tvt
import torchvision.transforms.functional as tvtF
import torch.optim as optim
import numpy as np
import math
import random
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import time
import glob
import imageio
from PIL import Image

LAMBDA = 10

class MyDataset(torch.utils.data.Dataset):
    """This is a custom dataset class that overwrites the ___len___ and __getitem__
    methods to load our pizza train/val dataset"""
    def __init__(self, root, train_or_val):
        super().__init__()
        self.root = root
        self.train_or_val = train_or_val
        self.all_images = list()
        self.transforms = tvt.Compose([
            tvt.PILToTensor(),
            tvt.ConvertImageDtype(torch.float),
            tvt.Normalize([0.5,0.5,0.5], [0.5,0.5,0.5])
        ])

        self.path_2_data = os.path.join(self.root, self.train_or_val)
```

```
36
37          self.build_img_list()
38
39      def build_img_list(self):
40          for filename in os.listdir(self.path_2_data):
41              filepath = os.path.join(self.path_2_data, filename)
42              self.all_images.append(filepath)
43
44
45      def __len__(self):
46          return len(self.all_images)
47
48      def __getitem__(self,idx):
49          img = Image.open(self.all_images[idx])
50          img = self.transforms(img)
51          return img
52
53  def weights_init(m):
54      """
55      method adopted directly from DLStudio's Adversarial Learning Class
56      """
57      classname = m.__class__.__name__
58      if classname.find('Conv') != -1:
59          nn.init.normal_(m.weight.data, 0.0, 0.02)
60      elif classname.find('BatchNorm') != -1:
61          nn.init.normal_(m.weight.data, 1.0, 0.02)
62          nn.init.constant_(m.bias.data, 0)
63
64  def train_gan(device, epochs, when_display, generator, discriminator, dataloader,
        save_path):
65      # set the number of channels for the 1x1 input noise vectors for the generator
66      nz = 100
67      netD = discriminator.to(device)
68      netG = generator.to(device)
69
70      # initialize the parameters of netD and netG
71      netD.apply(weights_init)
72      netG.apply(weights_init)
73
74      # use the same noise batch to check on netG progress
75      fixed_noise = torch.randn(32, nz, 1, 1, device=device)
76      real_label = 1
77      fake_label = 0
78
79      # define ADAM optimizers for netD and netG
80      optimizerD = optim.Adam(netD.parameters(), lr=2e-4, betas=(0.5, 0.999))
81      optimizerG = optim.Adam(netG.parameters(), lr=2e-4, betas=(0.5, 0.999))
82
83      # define the criterion for measuring loss at output of Discriminator
84      criterion = nn.BCELoss()
85
86      # use the following lists to store results accumulated during training
87      img_list = list()
88      G_losses = list()
89      D_losses = list()
90      iters = 0
91
92      print(f'Starting Training Loop')
93
94      for epoch in range(epochs):
```

```
        g_losses_per_print_cycle = list()
        d_losses_per_print_cycle = list()

        for i, data in enumerate(dataloader):
            '''Applying discriminator to training images'''
            netD.zero_grad()
            real_images_in_batch = data.to(device)

            # need to know how many images we pulled
            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()

            '''Applying discriminator to training 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_fakes + lossD_for_reals
            d_losses_per_print_cycle.append(lossD)
            optimizerD.step()

            '''Applying generator to output of 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 % when_display == when_display-1:
                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]      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 (iters % 500 == 0) or ((epoch == epoch-1) and (i == len(dataloader)
    -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
    print(f'Finished Training, Preparing Visuals')
    plt.figure(figsize=(10,5))
    plt.title("Generator and Discriminator Loss During Training")
    plt.plot(G_losses,label="G")
```

```
151     plt.plot(D_losses,label="D")
152     plt.xlabel("iterations")
153     plt.ylabel("Loss")
154     plt.legend()
155     plt.savefig(save_path + "/gen_and_disc_loss_training.png")
156 
157     #  Make an animated gif from the Generator output images stored in img_list:
158     images = []
159     for i,imgobj in enumerate(img_list):
160         img = tvtF.to_pil_image(imgobj)
161         images.append(img)
162         if i == len(img_list) -1:
163             for j in range(10):
164                 images.append(img)
165     imageio.mimsave(save_path + "/generation_animation.gif", images, fps=5)
166     #  Make a side-by-side comparison of a batch-size sampling of real images drawn
        from the
167     #  training data and what the Generator is capable of producing at the end of
        training:
168     real_batch = next(iter(dataloader))
169     real_batch = real_batch[0]
170     plt.figure(figsize=(15,15))
171     plt.subplot(1,2,1)
172     plt.axis("off")
173     plt.title("Real Images")
174     plt.imshow(np.transpose(torchvision.utils.make_grid(real_batch.to(device),
        padding=1, pad_value=1, normalize=True).cpu(),(1,2,0)))
175     plt.subplot(1,2,2)
176     plt.axis("off")
177     plt.title("Fake Images")
178     plt.imshow(np.transpose(img_list[-1],(1,2,0)))
179     plt.savefig(save_path + "/real_vs_fake_images.png")
180 
181 def calc_gradient_penalty(netC, real_images_in_batch, fake_images):
182     epsilon = torch.rand(1).cuda()
183     interpolates = epsilon * real_images_in_batch + ((1 - epsilon) * fake_images)
184     interpolates = interpolates.requires_grad_(True).cuda()
185     critic_interpolates = netC(interpolates)
186     gradients = torch.autograd.grad(outputs=critic_interpolates, inputs=interpolates,
187                                     grad_outputs=torch.ones(critic_interpolates.size()).
        cuda(),
188                                     create_graph=True, retain_graph=True, only_inputs=
        True)[0]
189     gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean() * LAMBDA
190     return gradient_penalty
191 
192 def train_wgan(dataloader, device, netC, netG, epochs):
193     nz = 100
194     netC.to(device)
195     netC.apply(weights_init)
196 
197     netG = netG.to(device)
198     netG.apply(weights_init)
199 
200     optimizerG = torch.optim.Adam(netG.parameters(), lr=1e-3, betas=(0.5, 0.999)) #
        Adam Optimizer for Generator
201     optimizerC = torch.optim.Adam(netC.parameters(), lr=1e-3, betas=(0.5, 0.999))
202 
203     fixed_noise = torch.randn(32, nz, 1, 1, device=device)
204 
```

```
    # To train the Critic, label=1 is assigned to the actual training images and label=-1 assigned to the fake images produced by the Generator
    one = torch.tensor([1], dtype=torch.float).to(device)
    minus_one = torch.tensor([-1], dtype=torch.float).to(device)

    image_list = []
    lossesG = []
    lossesC = []
    iters = 0
    gen_iterations = 0

    print(f'Starting Training')

    for epoch in range(epochs):
        data_iter = iter(dataloader)
        batch_idx, ncritic = 0, 5

        while(batch_idx < len(dataloader)):
            for p in netC.parameters():
                p.requires_grad = True
            ic = 0

            while (ic < ncritic and batch_idx < len(dataloader)):
                ic += 1

                netC.zero_grad()
                real_images_in_batch = next(data_iter)
                batch_idx += 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)

                noise = torch.randn(b_size, nz, 1, 1, device=device)
                fake_images = netG(noise)

                critic_for_fakes_mean = netC(fake_images.detach())
                critic_for_fakes_mean.backward(one)

                gradient_penalty = calc_gradient_penalty(netC, real_images_in_batch, fake_images)
                gradient_penalty.backward()
                wasser_dist = critic_for_reals_mean - critic_for_fakes_mean
                loss_critic = -wasser_dist + gradient_penalty

                optimizerC.step()

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

            noise = torch.randn(b_size, nz, 1, 1, device=device)
            fake_images = netG(noise)
            critic_for_fakes_mean = netC(fake_images)
            loss_gen = critic_for_fakes_mean
            loss_gen.backward(minus_one)

            optimizerG.step()
            gen_iterations += 1

```

```
263             # if batch_idx % (ncritic * 20) == 0:
264                 # print("[epoch=%d/%d   batch_idx=%4d]      loss_critic=%7.4f
    loss_gen=%7.4f    Wasserstein_dist=%7.4f" % (epoch+1,epochs,batch_idx,loss_critic
    .data[0], loss_gen.data[0], wasser_dist.data[0]))
265
266             lossesG.append(loss_gen.data[0].item())
267             lossesC.append(loss_critic.data[0].item())
268
269             if((iters % 500 == 0) or ((epoch == epochs) and (batch_idx == len(
    dataloader)-1))):
270                 with torch.no_grad():
271                     fake = netG(fixed_noise).detach().cpu()
272                 image_list.append(torchvision.utils.make_grid(fake, padding=1,
    pad_value=1, nrow=4, normalize=True))
273             iters += 1
274
275     print(f'Finished Training, Preparing Visuals')
276     plt.figure(figsize=(10,5))
277     plt.title("Generator and Critic Loss During Training")
278     plt.plot(lossesG,label="G")
279     plt.plot(lossesC,label="C")
280     plt.xlabel("iterations")
281     plt.ylabel("Loss")
282     plt.legend()
283     plt.savefig("/mnt/cloudNAS4/jo/ECE60146/HW07/results/wgan_train_loss.png")
284     plt.show()
285
286     images = []
287     for i, imgobj in enumerate(image_list):
288         img = tvtF.to_pil_image(imgobj)
289         images.append(img)
290         if i == len(image_list) -1:
291             for j in range(10):
292                 images.append(img)
293     imageio.mimsave("/mnt/cloudNAS4/jo/ECE60146/HW07/results/wgan_animation.gif",
    images, fps=5)
294     # plt.savefig(dir_name_for_results + "/gen_and_critic_loss_training.png")
295
296     real_batch = next(iter(dataloader))
297     real_batch = real_batch[0]
298     plt.figure(figsize=(15,15))
299     plt.subplot(1,2,1)
300     plt.axis("off")
301     plt.title("Real Images")
302     plt.imshow(np.transpose(torchvision.utils.make_grid(real_batch.to(device),
    padding=1, pad_value=1, normalize=True).cpu(),(1,2,0)))
303     plt.subplot(1,2,2)
304     plt.axis("off")
305     plt.title("Fake Images")
306     plt.imshow(np.transpose(image_list[-1],(1,2,0)))
307     plt.savefig("/mnt/cloudNAS4/jo/ECE60146/HW07/results/wgan_fakes.png")
```

util.py

Pizza BCE-GAN

Import Statements

In [1]:
```
'''External and system supplied classes and methods'''
import sys,os,os.path
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as tvt
import torchvision.transforms.functional as tvtF
import torch.optim as optim
import numpy as np
import math
import random
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import time
import glob
import imageio
import pickle
from torch.utils.data import DataLoader

'''User defined classes and methods'''
from models import *
from util import *

'''specify device for training and evaluation'''
device = 'cuda' if torch.cuda.is_available() == True else 'cpu'
print(device)
```

```
cuda
```

Instantiate Dataloader Instances For Train/Val Set

In [2]:
```
train_set = MyDataset(root='/mnt/cloudNAS4/jo/pizza/', train_or_val='train')
val_set = MyDataset(root='/mnt/cloudNAS4/jo/pizza/', train_or_val='eval')

train_loader = DataLoader(train_set, batch_size=32, num_workers=2, shuffle=T
val_loader = DataLoader(val_set, batch_size=2, num_workers=2, shuffle=True)
```

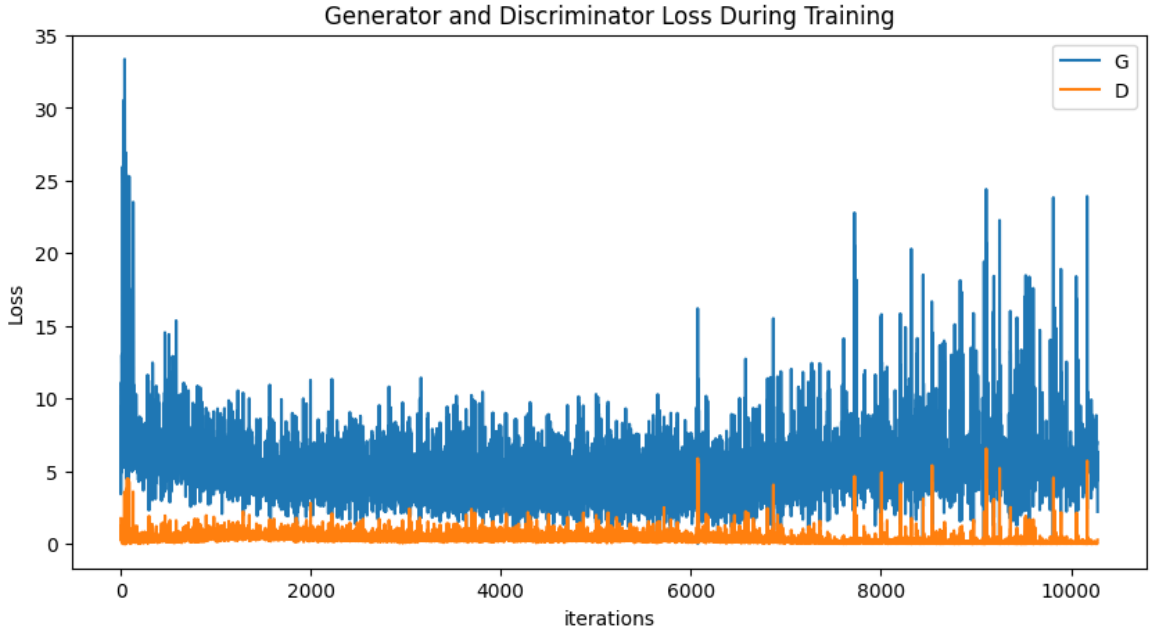
Instantiate Instances of Generator and Discriminator

In [3]:
```
generator = Generator()
discriminator = Discriminator()
```

Train the GAN

In [5]:
```
train_gan(device=device, epochs=40, when_display=100, generator=generator, d
```

```
Starting Training Loop
Finished Training, Preparing Visuals
```



Generate 1000 fakes

In [32]:
```
import cv2
fake_pizza_root = "/mnt/cloudNAS4/jo/fake_pizza"
noise = torch.randn(1000,100, 1, 1)
noise = noise.to(device)
img = generator(noise)

for i in range(1000):
    current_img = img[i]
    current_img = current_img / 2 + 0.5
    current_img = current_img.detach().cpu().numpy()
    current_img = np.transpose(current_img, (1,2,0))
```

```
    filename = os.path.join(fake_pizza_root, "fake"+str(i+1)+".jpg")
    plt.imsave(filename, current_img)
```

Compute FID Score of Generated Fakes

In [36]:
```
from pytorch_fid.fid_score import calculate_activation_statistics, calculate
from pytorch_fid.inception import InceptionV3

path_2_eval = "/mnt/cloudNAS4/jo/pizza/eval/"
path_2_fakes = "/mnt/cloudNAS4/jo/fake_pizza/"

real_paths = list()
fake_paths = list()

for filename in os.listdir(path_2_eval):
        filepath = os.path.join(path_2_eval, filename)
        real_paths.append(filepath)
for filename in os.listdir(path_2_fakes):
        filepath = os.path.join(path_2_fakes, filename)
        fake_paths.append(filepath)

assert(len(real_paths) == len(fake_paths))
assert(len(real_paths) == 1000)

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}')
```

```
Downloading: "https://github.com/mseitzer/pytorch-fid/releases/download/fid
_weights/pt_inception-2015-12-05-6726825d.pth" to /home/ubuntu/.cache/torc
h/hub/checkpoints/pt_inception-2015-12-05-6726825d.pth
  0%|          | 0.00/91.2M [00:00<?, ?B/s]
100%|██████████| 20/20 [00:02<00:00,  7.62it/s]
100%|██████████| 20/20 [00:01<00:00, 11.76it/s]
FID: 136.75
```

Pizza WGAN

Import Statements

In [1]:
```
'''External and system supplied classes and methods'''
import sys,os,os.path
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as tvt
import torchvision.transforms.functional as tvtF
import torch.optim as optim
import numpy as np
import math
import random
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import time
import glob
import imageio
import pickle
from torch.utils.data import DataLoader

'''User defined classes and methods'''
from models import *
from util import *

'''specify device for training and evaluation'''
device = 'cuda' if torch.cuda.is_available() == True else 'cpu'
print(device)
```

```
cuda
```

Instantitate Dataloader Instances For Train/Val set

In [2]:
```
train_set = MyDataset(root='/mnt/cloudNAS4/jo/pizza/', train_or_val='train')
val_set = MyDataset(root='/mnt/cloudNAS4/jo/pizza/', train_or_val='eval')

train_loader = DataLoader(train_set, batch_size=32, num_workers=0, shuffle=T
val_loader = DataLoader(val_set, batch_size=2, num_workers=2, shuffle=True)
```

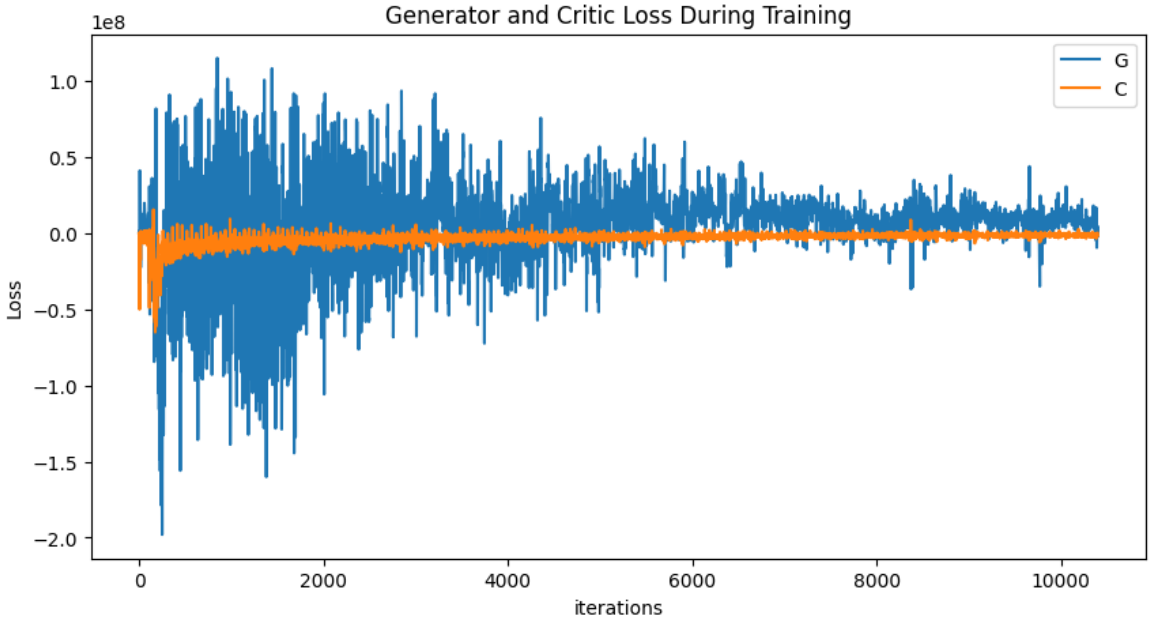
Instantitate Instances of Generator and Critic

In [3]:
```
generator = Generator()
critic = Critic1()
```

Train the WGAN

In [4]:
```
train_wgan(dataloader=train_loader, device=device, netC=critic, netG=generat
```

```
Starting Training
Finished Training, Preparing Visuals
```



Generate 1000 fakes

In [5]:
```
fake_pizza_root = "/mnt/cloudNAS4/jo/fake_pizza_wgan"
noise = torch.randn(1000,100, 1, 1)
noise = noise.to(device)
img = generator(noise)

for i in range(1000):
    current_img = img[i]
    current_img = current_img / 2 + 0.5
    current_img = current_img.detach().cpu().numpy()
    current_img = np.transpose(current_img, (1,2,0))
    filename = os.path.join(fake_pizza_root, "fake"+str(i+1)+".jpg")
    plt.imsave(filename, current_img)
```

In [6]:
```
from pytorch_fid.fid_score import calculate_activation_statistics, calculate
from pytorch_fid.inception import InceptionV3

path_2_eval = "/mnt/cloudNAS4/jo/pizza/eval/"
path_2_fakes = "/mnt/cloudNAS4/jo/fake_pizza_wgan//"
```

```
real_paths = list()
fake_paths = list()

for filename in os.listdir(path_2_eval):
        filepath = os.path.join(path_2_eval, filename)
        real_paths.append(filepath)
for filename in os.listdir(path_2_fakes):
        filepath = os.path.join(path_2_fakes, filename)
        fake_paths.append(filepath)

assert(len(real_paths) == len(fake_paths))
assert(len(real_paths) == 1000)

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}')
```

```
100%|██████████| 20/20 [00:02<00:00,  9.90it/s]
100%|██████████| 20/20 [00:02<00:00,  8.93it/s]
FID: 129.87
```