ECE 60146 HW7 Report

Zhengxin Jiang
(jiang839@purdue.edu)

1 Training Losses vs. Iterations

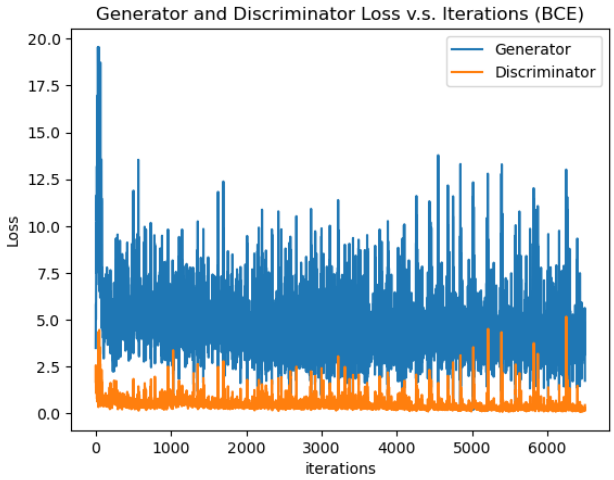

Figure 1: Plot of losses vs iterations for BCE-GAN

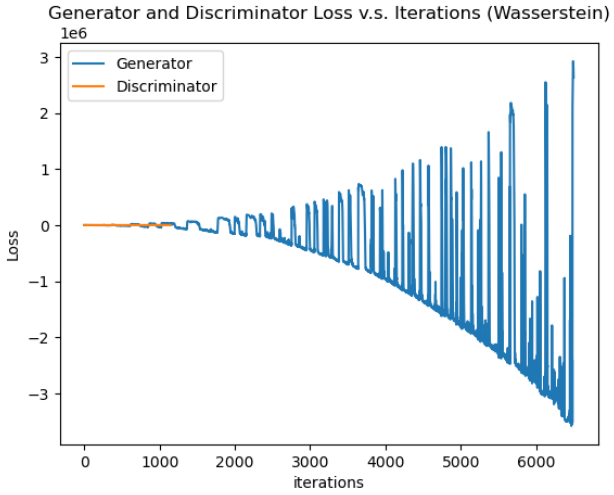

Figure 2: Plot of losses vs iterations for WGAN

Note that I use the implementation of WGAN suggested by our TA in the piazza post. The magnitude of loss does not matter here since the gradient targets of one and minus one are used.

2 Evaluation of GAN Networks

2.1 Qualitative Evaluation

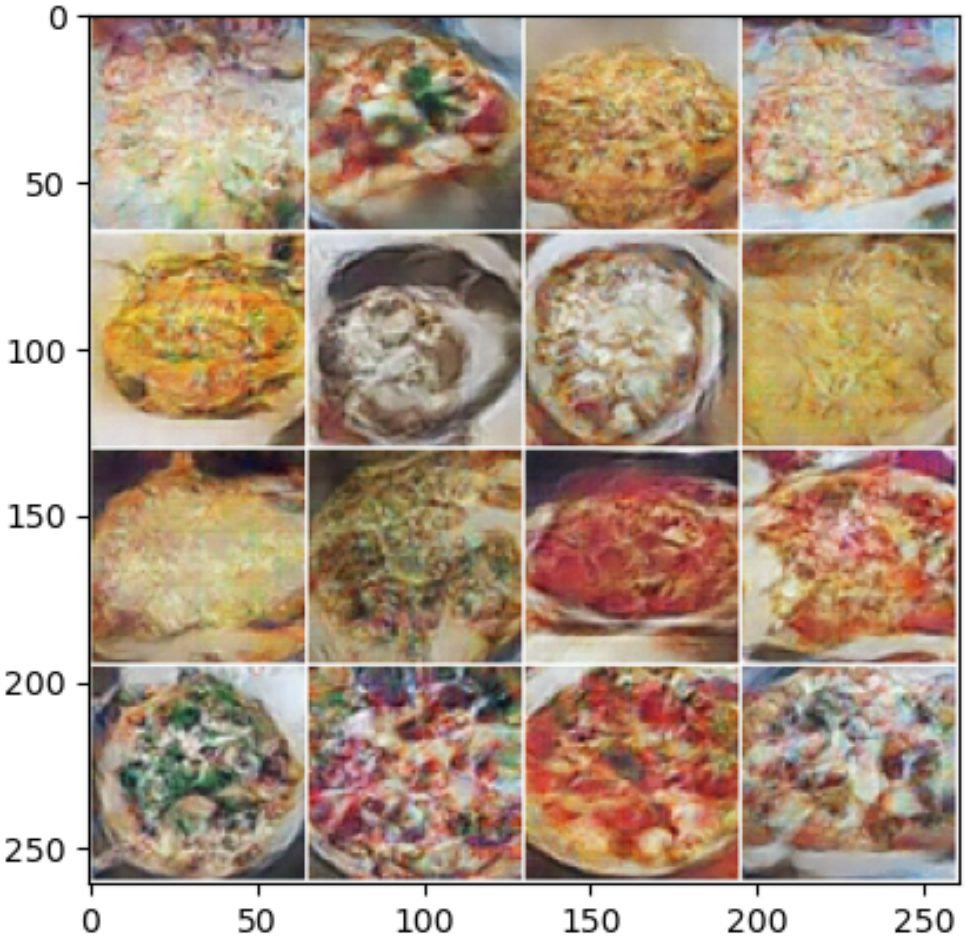

Figure 3: Images generated by BCE-GAN

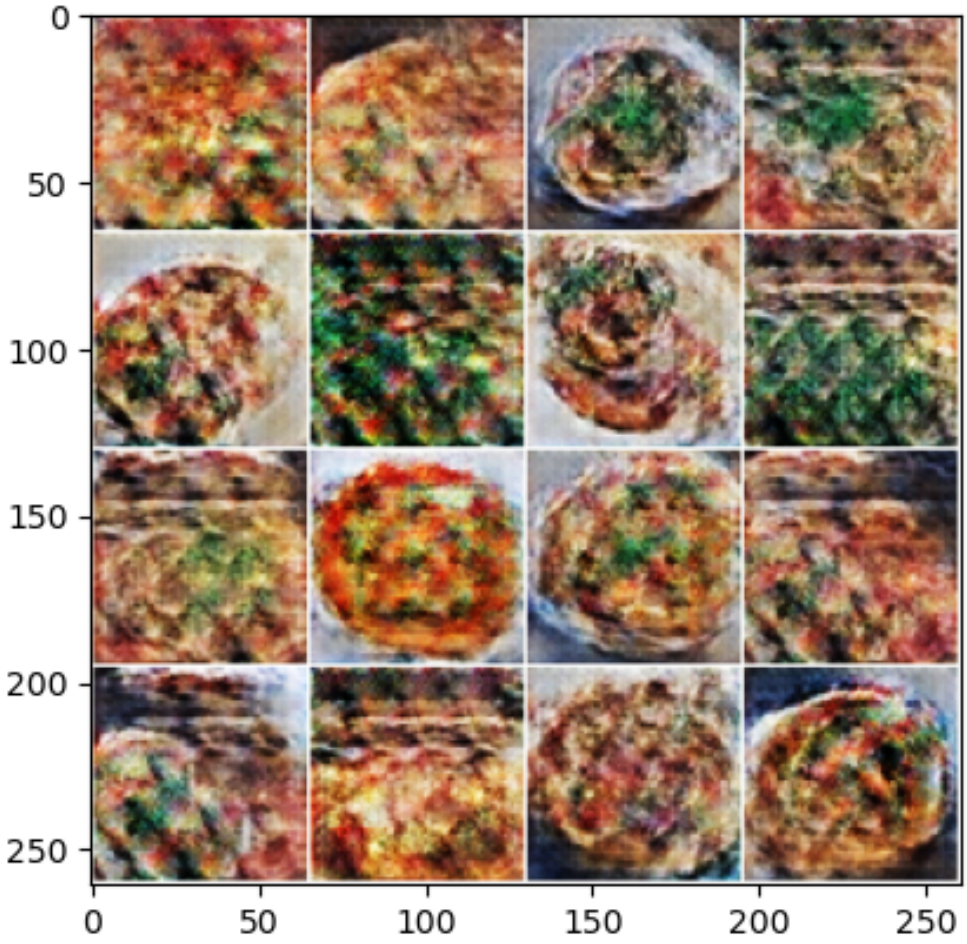

Figure 4: Images generated by WGAN

2.2 Quantitative Evaluation

```
(ece601_2) C:\Users\jzx>python -m pytorch_fid D:\coco\hw7\eval D:\coco\hw7\fake_bce
FID:  110.12495028070495
```

Figure 5: The FID value between real images and images generated by BCE-GAN

```
(ece601_2) C:\Users\jzx>python -m pytorch_fid D:\coco\hw7\eval D:\coco\hw7\fake_w
FID:  177.55913854905492
```

Figure 6: The FID value between real images and images generated by WGAN

3 Discussion

From the results of qualitative evaluation, both BCE-GAN and WGAN generate pizza-like images. The quality of images generated from the BCE-GAN can be better.

From the results of quantitative evaluation, BCE-GAN has less FID value than WGAN. We can say that BCE-GAN performs better.

4 Source code

```
# ECE60146 HW7
# Zhengxin Jiang
# jiang839

import numpy as np
import os
import matplotlib.pyplot as plt
from PIL import Image
from pycocotools.coco import COCO
import seaborn as sn
import random
import json
import math

import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as tvt
from torch.utils.data import DataLoader

import cv2

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

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)


# The Dataset class for hw7
class hwDataset(torch.utils.data.Dataset):

    def __init__(self, root, tasktype):
        super().__init__()

        if tasktype == 'training':
            self.root = os.path.join(root, 'hw7', 'train').replace("\\","/")
            self.idx_offset = 1001
        if tasktype == 'validation':
            self.root = os.path.join(root, 'hw7', 'eval').replace("\\","/")
            self.idx_offset = 1

    def __len__(self):
        return len(os.listdir(self.root))

    def __getitem__(self, index):

        filename = str(index+self.idx_offset).zfill(5)+'.jpg'
        img = Image.open(os.path.join(self.root, filename).replace("\\","/"))
```

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

        img_tensor = tr(img)

        return img_tensor

# The generator network
class generator(nn.Module):

    def __init__(self):
        super().__init__()

        self.main = nn.Sequential(
            nn.ConvTranspose2d(100, 512, 4, 1, 0, bias=False),
            nn.BatchNorm2d(512),
            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, x):
        return self.main(x)

# The discriminator network
class discriminator(nn.Module):

    def __init__(self):
        super().__init__()

        self.main = 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, x):
        return self.main(x)

class critic(nn.Module):

    def __init__(self):
        super().__init__()

        self.main = 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, 512, 4, 1, 0, bias=False),
            nn.Flatten(),
            nn.Linear(512, 1)

        )

    def forward(self, x):
        x = self.main(x)
        x = x.mean(0)
        x = x.view(1)

        return x



# custom weights initialization called on netG and netD
# https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html
```

```
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)



######### MAIN ##########

root = 'D:/coco'

lr = 0.0001
batch_size = 128
epochs = 100

traindataset = hwDataset(root, 'training')
train_data_loader = DataLoader(traindataset, batch_size=batch_size, num_workers=0,
    shuffle=True)


##### Training BCE #####
netG = generator()
netG = netG.to(device)
netD = discriminator()
netD = netD.to(device)

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

criterion = nn.BCELoss()

optimizerD = torch.optim.Adam(netD.parameters(), lr=lr, betas=(0.5, 0.999))
optimizerG = torch.optim.Adam(netG.parameters(), lr=lr, betas=(0.5, 0.999))

G_losses = []
D_losses = []

for epoch in range(epochs):
    g_loss = 0.0
    d_loss = 0.0

    for i, data in enumerate(train_data_loader):

        bsize = data.size(0)
        inputs = data.to(device)
        label = torch.full((bsize,), 1, dtype=torch.float, device=device)

        ## update discriminator
        ## train with real images
        netD.zero_grad()
        output = netD(inputs)
```

```
        errD_real = criterion(output.view(-1), label)
        errD_real.backward()

        ## train with fake images
        noise = torch.randn(bsize, 100, 1, 1, device=device)
        fake = netG(noise)
        label.fill_(0)

        output = netD(fake.detach())
        errD_fake = criterion(output.view(-1), label)
        errD_fake.backward()

        errD = errD_real + errD_fake
        optimizerD.step()

        ## update generator
        netG.zero_grad()
        label.fill_(1)

        output = netD(fake) # another forward pass in D
        errG = criterion(output.view(-1), label)
        errG.backward()
        optimizerG.step()

        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D:␣%.4f\tLoss_G:␣%.4f'
                  % (epoch, epochs, i, len(train_data_loader),
                     errD.item(), errG.item()))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())


##### Training WGAN #####
netG = generator()
netG = netG.to(device)
netD = critic()
netD = netD.to(device)

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

# criterion = nn.BCELoss()

optimizerD = torch.optim.Adam(netD.parameters(), lr=lr, betas=(0.5, 0.999))
optimizerG = torch.optim.Adam(netG.parameters(), lr=lr, betas=(0.5, 0.999))

G_losses = []
D_losses = []

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

```
for epoch in range(epochs):
    g_loss = 0.0
    d_loss = 0.0

    for i, data in enumerate(train_data_loader):

        bsize = data.size(0)
        inputs = data.to(device)
        label = torch.full((bsize,), 1, dtype=torch.float, device=device)

        ## update discriminator
        ## train with real images
        netD.zero_grad()
        output = netD(inputs)

        errD_real = output
        errD_real.backward(minus_one)

        ## train with fake images
        noise = torch.randn(bsize, 100, 1, 1, device=device)
        fake = netG(noise)
        label.fill_(0)

        output = netD(fake.detach())
        errD_fake = output
        errD_fake.backward(one)

        errD = errD_real + errD_fake
        optimizerD.step()

        ## update generator
        netG.zero_grad()
        label.fill_(1)

        output = netD(fake) # another forward pass in D
        errG = output
        errG.backward(minus_one)
        optimizerG.step()

        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D:␣%.4f\tLoss_G:␣%.4f'
                  % (epoch, epochs, i, len(train_data_loader),
                     errD.item(), errG.item()))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())


## save models
# torch.save(netG.state_dict(), 'G_BCE.pth')
# torch.save(netD.state_dict(), 'D_BCE.pth')
torch.save(netG.state_dict(), 'G_W.pth')
```

```
torch.save(netD.state_dict(), 'D_W.pth')

## plot the loss
plt.figure()
plt.title("Generator and Discriminator Loss v.s. Iterations (Wasserstein)")
plt.plot(G_losses,label="Generator")
# plt.plot(D_losses,label="Discriminator")
plt.plot(C_losses,label="Discriminator")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()

## load models
netG = generator()
netG = netG.to(device)
netD = discriminator()
netD = netD.to(device)

netG.load_state_dict(torch.load('G_BCE.pth', map_location=torch.device(device)))
netD.load_state_dict(torch.load('D_BCE.pth', map_location=torch.device(device)))

##### Evaluation #####
fake_num = 1000
# fake_folder = 'fake_bce'
fake_folder = 'fake_w'

## generate 1000 fake images
with torch.no_grad():
    noise = torch.randn(fake_num, 100, 1, 1, device=device)
    fake = netG(noise)
    fake = fake.view(fake_num, 3, 64, 64)

img_grid = np.transpose(torchvision.utils.make_grid(fake[:16], nrow = 4,
                                                    padding=1, pad_value=1, normalize=True).
                                                        cpu(),(1,2,0))

transform = tvt.Compose([
               tvt.Normalize((0,0,0),(2,2,2)),
               tvt.Normalize((-0.5,-0.5,-0.5,),(1,1,1)),
               tvt.ToPILImage()
           ])

for i in range(fake_num):
    img = transform(fake[i])
    filename = str(i).zfill(5)+'.jpg'
    img.save(os.path.join(root, 'hw7', fake_folder, filename).replace("\\","/"))

plt.imshow(img_grid)
```