# hw7_ChengjunGuo

Chengjun Guo

April 2023

## 1 Loss

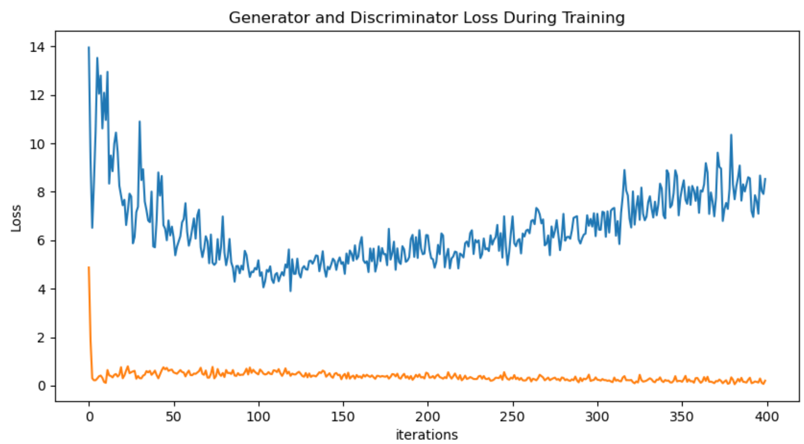


Figure 1: BCE loss

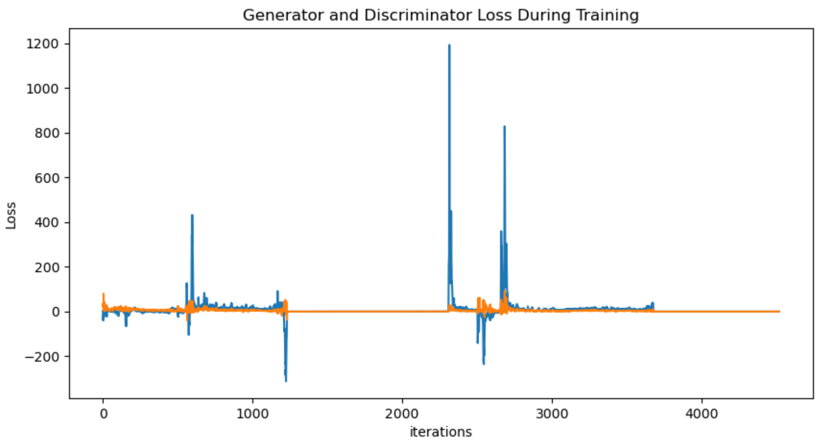


Figure 2: WGAN

# 2 real image vs fake

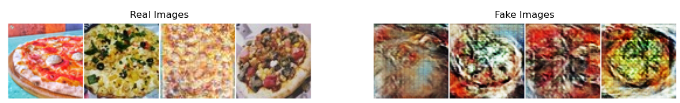


Figure 3: BCE real vs fake

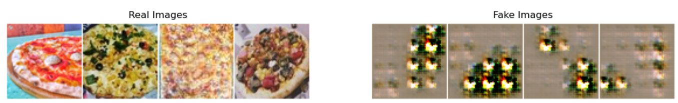


Figure 4: WGAN real vs fake

# 3 evaluation

Figure 5: BCE eval

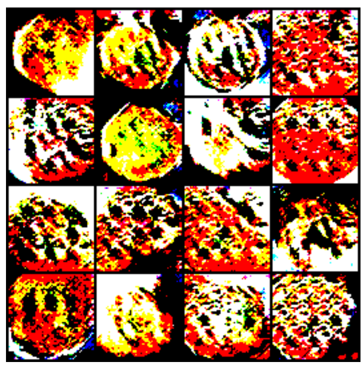

Figure 6: network

# 4 FID

| | FID |
|---|---|
| DCGAN | 207.65 |
| WGAN | 327.72 |

# 5 Discussion

WGAN uses the Wasserstein distance to measure the difference between the distributions of real and generated images. Based on my result, WGAN seems to have worse performance than DCGAN which use BCE for loss. DCGAN have lower FID score than WGAN which means the distribution of real images is closer to the distribution of generated images generated by DCGAN.

## 6 Code

hw7_net:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class HW7Generator(nn.Module):
    def __init__(self):
        super(HW7Generator, self).__init__()
        model = [nn.ConvTranspose2d(100, 64 * 8,
            kernel_size=4, stride=1, padding=0),
                 nn.LeakyReLU(0.2,inplace=True),
                 nn.ConvTranspose2d(64 * 8, 64 * 4,
                     kernel_size=4, stride=2, padding=1),
                 nn.BatchNorm2d(64 * 4),
                 nn.LeakyReLU(0.2, inplace=True),
                 nn.ConvTranspose2d(64 * 4, 64 * 2,
                     kernel_size=4, stride=2, padding=1),
                 nn.BatchNorm2d(64 * 2),
                 nn.LeakyReLU(0.2, inplace=True),
                 nn.ConvTranspose2d(64 * 2, 64,
                     kernel_size=4, stride=2, padding=1),
                 nn.BatchNorm2d(64),
                 nn.LeakyReLU(0.2, inplace=True),
                 nn.ConvTranspose2d(64, 3, kernel_size=4,
                      stride=2, padding=1),
                 ]

        self.model = nn.Sequential(*model)
        head = [nn.Tanh()]
        self.head = nn.Sequential(*head)

    def forward(self, input):
        ft = self.model(input)
        ft = self.head(ft)
        return ft


class HW7Discriminator(nn.Module):
    def __init__(self):
        super(HW7Discriminator, self).__init__()
```

```
        model = [nn.Conv2d(3, 64, kernel_size=4, stride
            =2, padding=1),
                 nn.LeakyReLU(0.2,inplace=True),
                 nn.Conv2d(64, 64 * 2, kernel_size=4,
                     stride=2, padding=1),
                 nn.BatchNorm2d(64 * 2),
                 nn.LeakyReLU(0.2, inplace=True),
                 nn.Conv2d(64 * 2, 64 * 4, kernel_size=4,
                      stride=2, padding=1),
                 nn.BatchNorm2d(64 * 4),
                 nn.LeakyReLU(0.2, inplace=True),
                 nn.Conv2d(64 * 4, 64 * 8, kernel_size=4,
                      stride=2, padding=1),
                 nn.BatchNorm2d(64 * 8),
                 nn.LeakyReLU(0.2, inplace=True),
                 nn.Conv2d(64 * 8, 1, kernel_size=4,
                     stride=1, padding=0)
                 ]

        self.model = nn.Sequential(*model)
        head = [nn.Sigmoid()]
        self.head = nn.Sequential(*head)

    def forward(self, input):
        # print(input.shape)
        ft = self.model(input)
        ft = self.head(ft)
        return ft

class HW7Critic(nn.Module):
    def __init__(self):
        super(HW7Critic, self).__init__()
        self.DIM = 64
        model = [
            nn.Conv2d(3, self.DIM, 5, stride=2, padding
                =2),
            nn.ReLU(True),
            nn.Conv2d(self.DIM, 2 * self.DIM, 5, stride
                =2, padding=2),
            nn.ReLU(True),
            nn.Conv2d(2 * self.DIM, 4 * self.DIM, 5,
                stride=2, padding=2),
            nn.ReLU(True),
        ]
        self.model = nn.Sequential(*model)
        self.output = nn.Linear(4 * 4 * 4 * self.DIM, 1)
```

```
    def forward(self,input):
        input = input.view(-1, 3, 64, 64)
        out = self.model(input)
        out = out.view(-1, 4 * 4 * 4 * self.DIM)
        out = self.output(out)
        out = out.mean(0)
        out = out.view(1)
        return out
```

hw7:

```
import os
os.environ['KMP_DUPLICATE_LIB_OK'] = 'True'
import torch
import torch.nn as nn
import torch.nn.functional as F
from PIL import Image
import torchvision
import torchvision.transforms as tvt
from torchvision.io import read_image
from torch.utils.data import DataLoader, Dataset
import copy
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
from hw7_net import HW7Generator,HW7Discriminator,
    HW7Critic
import time
import pickle
from statistics import mean
import cv2
from skimage import color, io
import numpy as np
import pandas as pd
from pytorch_fid.fid_score import
    calculate_activation_statistics,
    calculate_frechet_distance
from pytorch_fid.inception import InceptionV3
```

```
## ref: avi kak tutorial:https://engineering.purdue.edu/
    kak/distDLS/AdversarialLearning-2.2.5_CodeOnly.html


class MyDataset(torch.utils.data.Dataset):
    def __init__(self,istrain):
        super().__init__()
        self.istrain = istrain
        self.root_path = 'E:\ECE60146DL\hw7_new/'
        self.train_path = 'E:\ECE60146DL\hw7_new/train/'
        self.val_path = 'E:\ECE60146DL\hw7_new/eval/'
        self.data = []
        self.transform = tvt.Compose([tvt.ToTensor(), tvt
            .Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])])
        if self.istrain:
            path = self.train_path
        else:
            path = self.val_path
        for root, dirs, files in os.walk(path, topdown=
            False):
            for name in files:
                self.data.append(os.path.join(root, name)
                    )



    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        img = cv2.imread(self.data[index])
        if img.ndim == 2 or img.shape[-1]<3:
            img = color.gray2rgb(img)
        img = cv2.resize(img, (64, 64))
        image = self.transform(img)
        return image.type(torch.float)


def run_training_BCE(data_loader):
    nz = 100
    batch_size = 4
    discriminator = HW7Discriminator()
    generator = HW7Generator()
    netD = discriminator.to(device)
    netG = generator.to(device)
```

```
netD.apply(weights_init)
netG.apply(weights_init)
fixed_noise = torch.randn(batch_size, nz, 1, 1,
    device=device)
real_label = 1
fake_label = 0
optimizerD = torch.optim.Adam(netD.parameters(), lr=1
    e-3, betas=(0.9, 0.999))
optimizerG = torch.optim.Adam(netG.parameters(), lr=1
    e-3, betas=(0.9, 0.999))
criterion = nn.BCELoss()
epochs = 20
loss_g = []
loss_d = []
img_list = []
start_time = time.time()
for epoch in range(epochs):
    running_loss_generator = 0.0
    running_loss_discriminator = 0.0
    for i, data in enumerate(data_loader):
        netD.zero_grad()
        real_images_in_batch = data.to(device)
        # print(data.shape)
        label = torch.full((data.shape[0],),
            real_label, dtype=torch.float, device=
            device)
        output = netD(real_images_in_batch).view(-1)
        # print(output.shape,label.shape,i)
        lossD_for_reals = criterion(output, label)
        lossD_for_reals.backward()

        noise = torch.randn(data.shape[0], 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
        running_loss_discriminator += lossD.item()

        optimizerD.step()
        netG.zero_grad()
        label.fill_(real_label)
        output = netD(fakes).view(-1)
        lossG = criterion(output, label)
```

```
            running_loss_generator += lossG.item()
            lossG.backward()
            optimizerG.step()
            if i % 100 == 99:
                elapsed_time = time.time() - start_time
                start_time = time.time()
                print("[epoch=%d/%d    iter=%4d
                    elapsed_time=%5d secs]      mean_D_loss
                    =%7.4f      mean_G_loss=%7.4f" %((epoch
                    + 1), epochs, (i + 1), elapsed_time,
                    running_loss_discriminator/100,
                    running_loss_generator/100))
                loss_g.append(running_loss_generator/100)
                loss_d.append(running_loss_discriminator
                    /100)
                running_loss_generator = 0.0
                running_loss_discriminator = 0.0

            if (i % 500 == 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))

    torch.save(netD,"netDBCE.pth")
    torch.save(netG,"netGBCE.pth")
    return loss_d, loss_g, img_list



def run_training_W(data_loader):
    nz = 100
    batch_size = 4
    critic = HW7Critic()
    generator = HW7Generator()
    netC = critic.to(device)
    netG = generator.to(device)
    netC.apply(weights_init)
    netG.apply(weights_init)
    fixed_noise = torch.randn(batch_size, nz, 1, 1,
        device=device)
    one = torch.FloatTensor([1]).to(device)
    minus_one = torch.FloatTensor([-1]).to(device)
```

```
optimizerC = torch.optim.Adam(netC.parameters(), lr=1
    e-3, betas=(0.9, 0.999))
optimizerG = torch.optim.Adam(netG.parameters(), lr=1
    e-3, betas=(0.9, 0.999))
epochs = 20
loss_g = []
loss_c = []
img_list = []
gen_iterations = 0
start_time = time.time()
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 %
            500 == 0:  # the choices 25 and 500 are
            from WGAN
            ncritic = 100
        ic = 0
        while ic < ncritic and i < len(data_loader):
            ic += 1
            for p in netC.parameters():
                p.data.clamp_(-0.005, +0.005)
            netC.zero_grad()
            real_images_in_batch = next(data_iter)
            i += 1
            real_images_in_batch =
                real_images_in_batch.to(device)
            b_size = len(real_images_in_batch)
            critic_for_reals_mean = netC(
                real_images_in_batch)
            critic_for_reals_mean.backward(one)

            noise = torch.randn(b_size, nz, 1, 1,
                device=device)
            fakes = netG(noise)
            critic_for_fakes_mean = netC(fakes.detach
                ())
            critic_for_fakes_mean.backward(minus_one)
            wasser_dist = critic_for_reals_mean -
                critic_for_fakes_mean
            loss_critic = critic_for_fakes_mean -
                critic_for_reals_mean
```

```
                optimizerC.step()

            for p in netC.parameters():
                p.requires_grad = False
            # noise = torch.randn(b_size, nz, 1, 1,
               device=device)
            netG.zero_grad()
            # fakes = netG(noise)
            critic_for_fakes_mean = netC(fakes)
            loss_gen = critic_for_fakes_mean
            critic_for_fakes_mean.backward(one)
            # Update the Generator
            optimizerG.step()
            gen_iterations += 1

            if i % (ncritic * 20) == 0:
                elapsed_time = time.time() - start_time
                start_time = time.time()
                print("[epoch=%d/%d    i=%4d    el_time=%5d
                     secs]      loss_critic=%7.4f
                    loss_gen=%7.4f    Wasserstein_dist=%7.4
                    f" % (epoch, epochs, i, elapsed_time,
                    loss_critic.data[0], loss_gen.data[0],
                    wasser_dist.data[0]))
            loss_g.append(loss_gen.data[0].item())
            loss_c.append(loss_critic.data[0].item())
            #  Get G's output on fixed_noise for the GIF
               animation:
            if (i % 500 == 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))

    torch.save(netC, "netCW.pth")
    torch.save(netG, "netGW.pth")
    return loss_c, loss_g, img_list




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)

if __name__ == '__main__':
    if torch.cuda.is_available() == True:
        device = torch.device("cuda:0")
    else:
        device = torch.device("cpu")

    # train_dataset = MyDataset(True)

    # train_data_loader = torch.utils.data.DataLoader(
        dataset=train_dataset, batch_size=4, shuffle=False
        , num_workers=4)
    # loss_d, loss_g, img_list = run_training_BCE(
        train_data_loader)
    # with open("test", "wb") as fp:
    #     pickle.dump(loss_d, fp)
    #     pickle.dump(loss_g, fp)
    #     pickle.dump(img_list, fp)
    # with open("test", "rb") as fp:
    #     loss_d = pickle.load(fp)
    #     loss_g = pickle.load(fp)
    #     img_list = pickle.load(fp)
    # plt.figure(figsize=(10, 5))
    # # df = pd.DataFrame(loss_g)
    # # print(df.head())
    # plt.title("Generator and Discriminator Loss During
        Training")
    # plt.plot(loss_g, label="G", linestyle='-')
    # plt.plot(loss_d, label="D", linestyle='-')
    # plt.xlabel("iterations")
    # plt.ylabel("Loss")
    # # plt.legend()
    # plt.savefig("loss_BCE.png")
    # plt.show()

    # real_batch = next(iter(train_data_loader))
    # plt.figure(figsize=(15, 15))
    # plt.subplot(1, 2, 1)
    # plt.axis("off")
    # plt.title("Real Images")
    # x = np.transpose(torchvision.utils.make_grid(
```

```
    real_batch.to(device),padding=1, pad_value=1,
    normalize=True).cpu(), (1, 2, 0)).numpy()
# img = cv2.cvtColor(x, cv2.COLOR_BGR2RGB)
# plt.imshow(img)
# plt.subplot(1, 2, 2)
# plt.axis("off")
# plt.title("Fake Images")
# img = cv2.cvtColor(np.transpose(img_list[-1], (1,
    2, 0)).numpy(),cv2.COLOR_BGR2RGB)
# plt.imshow(img)
# plt.savefig("real_vs_fake_BCE.png")
# plt.show()

# loss_c, loss_g, img_list = run_training_W(
    train_data_loader)
# with open("test2", "wb") as fp:
#     pickle.dump(loss_c, fp)
#     pickle.dump(loss_g, fp)
#     pickle.dump(img_list, fp)
# with open("test2", "rb") as fp:
#     loss_c = pickle.load(fp)
#     loss_g = pickle.load(fp)
#     img_list = pickle.load(fp)
# plt.figure(figsize=(10, 5))
# # df = pd.DataFrame(loss_g)
# # print(df.head())
# plt.title("Generator and Discriminator Loss During
    Training")
# plt.plot(loss_g, label="G", linestyle='-')
# plt.plot(loss_c, label="C", linestyle='-')
# plt.xlabel("iterations")
# plt.ylabel("Loss")
# # plt.legend()
# plt.savefig("loss_W.png")
# plt.show()
#
# real_batch = next(iter(train_data_loader))
# plt.figure(figsize=(15, 15))
# plt.subplot(1, 2, 1)
# plt.axis("off")
# plt.title("Real Images")
# x = np.transpose(torchvision.utils.make_grid(
    real_batch.to(device),padding=1, pad_value=1,
    normalize=True).cpu(), (1, 2, 0)).numpy()
# img = cv2.cvtColor(x, cv2.COLOR_BGR2RGB)
# plt.imshow(img)
```

```
# plt.subplot(1, 2, 2)
# plt.axis("off")
# plt.title("Fake Images")
# img = cv2.cvtColor(np.transpose(img_list[-1], (1,
    2, 0)).numpy(),cv2.COLOR_BGR2RGB)
# plt.imshow(img)
# plt.savefig("real_vs_fake_W.png")
# plt.show()

eval_dataset = MyDataset(False)

eval_data_loader = torch.utils.data.DataLoader(
    dataset=eval_dataset, batch_size=4, shuffle=False,
     num_workers=4)
# real_paths = eval_dataset.data
# fake_paths = []
# # netDBCE = torch.load('netDBCE.pth')
# netGBCE = torch.load('netGBCE.pth')
# for i, data in enumerate(eval_data_loader):
#     with torch.no_grad():
#         fake = netGBCE(torch.randn(4, 100, 1, 1,
    device=device)).detach().cpu()
#
#         for j in range(len(fake)):
#             id = 4 * i + j
#             fake_paths.append('E:\ECE60146DL\
    hw7_new/fakeBCE/'+str(int(id))+'.png')
#             torchvision.utils.save_image(fake[j
    ][[2,1,0],:], 'E:\ECE60146DL\hw7_new/fakeBCE/'+str(
    int(id))+'.png')
#
#
# 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'BCE FID:{fid_value:.2f}')


# netCW = torch.load('netCW.pth')
netGW = torch.load('netGW.pth')
```

```
    real_paths = eval_dataset.data
    fake_paths = []
    for i, data in enumerate(eval_data_loader):
        with torch.no_grad():
            fake = netGW(torch.randn(4, 100, 1, 1, device
                =device)).detach().cpu()

            for j in range(len(fake)):
                id = 4 * i + j
                fake_paths.append('E:\ECE60146DL\hw7_new/
                    fakeW/'+str(int(id))+'.png')
                torchvision.utils.save_image(fake[j
                    ][[2,1,0],:], 'E:\ECE60146DL\hw7_new/
                    fakeW/'+str(int(id))+'.png')
    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'W FID:{fid_value:.2f}')

# BCE FID:207.65

#  FID:327.72


    # img_grid = torch.zeros([16,3,64,64])
    # for i in range(16):
    #     img = torchvision.io.read_image('E:\ECE60146DL\
        hw7_new/fakeW/'+str(int(i))+'.png')
    #     img_grid[i] = img
    # torchvision.utils.save_image(img_grid,'Weval.png',
        nrow=4)

    # img_grid = torch.zeros([16,3,64,64])
    # for i in range(16):
    #     img = torchvision.io.read_image('E:\ECE60146DL\
        hw7_new/fakeBCE/'+str(int(i))+'.png')
    #     img_grid[i] = img
    # torchvision.utils.save_image(img_grid,'BCEeval.png
        ',nrow=4)
```