ECE 60146 Deep Learning Homework 7

Mehmet Berk Sahin, sahinm@purdue.edu, 34740048

April 6, 2023

1 Introduction

The purpose of this homework is to implement our own pizza-generating Generative Adversarial Networks (GAN). Main take aways from this homework are as follows: understanding how the generator and discriminator networks are trained to beat each other in a minimax game, experimenting with different GAN criteria such as Binary Cross-Entropy (BCE) and the Wasserstein distance, and evaluating the generated images qualitatively and quantitatively via Frechet Inception Distance (FID). Before getting into the dataset and solution of the homework, I want to explain GANs and WGANs briefly to refer them later.

1.1 Generative Adversarial Network [1]

GAN was first proposed in [1]. They consists of two networks, which are generator and discriminator. Discriminator's objective is classify a sample either as real or fake sample. And generator's objective is deceive the discriminator by generating fake images, which look as if they are from the training set. This competition between two networks was formulated as a minimax optimization problem as follows:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}\left[\log D(x)\right] + \mathbb{E}_{x\sim p_z(z)}\left[\log\left(1 - D(G(z))\right)\right]. \tag{1}$$

As can be seen in the above formulation, discriminator's job is learning the classification for real and fake images by minimizing the corresponding binary cross-entropy. As opposed to that, generator tries to learn to deceive the discriminator by maximizing its binary cross-entropy loss. It was shown, in [1], that equation (1) is equivalent to saying that generator tries to minimze the *Jensen-Shannon* divergence between two distributions. And *JS* divergence is summation of two *Kullback-Leibler* (*KL*) divergence terms. They are defined as follows:

- The *Jensen-Shannon* (JS) divergence:

$$JS(\mathbb{P}_r, \mathbb{P}_g) = KL(\mathbb{P}_r||\mathbb{P}_m) + KL(\mathbb{P}_g||\mathbb{P}_m) \tag{2}$$

  where $\mathbb{P}_m = \frac{\mathbb{P}_r + \mathbb{P}_g}{2}$.

- The *Kullback-Leibler* (KL) divergence:

$$KL(\mathbb{P}_r||\mathbb{P}_g) = \int \log\left(\frac{P_r(x)}{P_g(x)}\right) P_r(x) d\mu(x) \tag{3}$$

  where $0 \leq P_r(x)$ and $0 < P_g(x)$.

As you may notice above, although JS and KL divergences are measures of difference between two probability distributions, they are not sufficient because they do not measure the "distance" between two distributions. For example, if for a sample, one of the probabilities is approximately 0 and other is not zero, then this will not give any meaningful gradient. And this is very likely at the initialization of the distributions. Hence, to use the distance information between two distributions, [2] introduced Wasserstein distance into the training algorithm. It is given as:

- The *Wasserstein* distance:

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma} \left[\|x - y\|\right] \tag{4}$$

  where $\Pi(\mathbb{P}_r, \mathbb{P}_g)$ indicates the set of all join distributions $\gamma(x, y)$ whose marginals are $\mathbb{P}_r$ and $\mathbb{P}_g$ respectively.

This is also called *Earth-Mover* distance and it denotes the "cost" of the optimal transport plan [2].

1.2 Wasserstein GAN

As JS measure is insufficient to measure the difference between probability distributions, GAN models are highly unstable and they are likely to result in mode collapse. Meaning that discriminator learned more than the generator and this prevent generator from learning the distribution of the data. [2] introduced *Wasserstein* distance as measure of difference between fake and real image distributions and they proposed WGAN. It was shown that wasserstein distance is equivalent to the dual maximization problem:

$$W(\mathbb{P}_r, \mathbb{P}_\theta) = \sup_{\|f\|\leq 1} \mathbb{E}_{x\sim\mathbb{P}_r} \left[f(x)\right] - \mathbb{E}_{x\sim\mathbb{P}_\theta} \left[f(x)\right] \tag{5}$$

where the supremum is overall 1-Lipschitz functions *f*. In the training, to impose the lipschitz constraint, there are several methods. Some of them are gradient clipping, gradient penalty, and spectral norm. First one, restricts the gradient values between two bounds, which are pre-defined. Second one adds the constraint as a soft penalty term to the loss to penalize the high gradients. The third one normalizes the gradients via maximum singular values. In this homework, I reported the results of WGAN gradient penalty. Thus, the critic loss function I used for WGAN is as follows:

$$CriticLoss = \mathbb{E}_{x\sim\mathbb{P}_\theta} \left[C(G(x))\right] - \mathbb{E}_{x\sim\mathbb{P}_r} \left[C(x)\right] + \lambda \left[\|\nabla_{\hat{x}} C(\hat{x})\| - 1\right]^2, \tag{6}$$

where $\hat{x} = \epsilon x + (1 - \epsilon)z$ for $z \sim \mathbb{P}_\theta$. The interpolation between two distributions come from Proposition 1 in [3]. And generator's objective is maximizing the first term in that loss. Note that minimization of critic loss corresponds to maximization of Wasserstein distance and vice versa.

In this homework, I implemented DCGAN, which is a type of GAN consisting of convolutional layers [4], and WGAN with gradient clipping and gradient penalty. This homework report is structured as follows. In Section 2, I go through the dataset. In Section 3 and Section 4, I explained the implementation of GAN and WGAN architectures and their train algorithms respectively. I did not explain the code for FID because I just used the template given in the homework. In Section 5, I did quantitative and qualitative evaluation of the results and discussed them. In Section 6, I explained how to run the code and in Section 7, I discussed takeaways of this homework. Source code and additional images can be found in the Appendix. **Videos for the outputs of GAN and WGAN for each 500 iteration can be found in the submission folder!**

2 Dataset

Dataset required for this homework is simpler than previous homeworks so it required less work to do. Dataset is divided into two as train and test sets, which consist of pizza images. Train set consists of 8.2k pizza images and test set consists of 1k pizza images. Each image was resized to 64×64. To incorporate the dataset into PyTorch training loop, I implemented a custom PyTorch dataset in `dataset.py` module.

2.1 `dataset.py`

Although implementation of the custom dataset is simpler than previous homeworks, I wanted to explain it to show the reader the pre-processing steps that each image goes through. The name of

```python
class PizzaDataset(nn.Module):

    def __init__(self, path="pizza", train=True):
        super(PizzaDataset, self).__init__()
        self.data_dir = os.path.join(path, "train") if train else os.path.join(path, "eval")
        self.file_list = os.listdir(self.data_dir) # list of file names
        # copy image -> tensor -> normalize
        self.transform = tvt.Compose([tvt.Lambda(lambda img: img.copy()),
                                      tvt.ToTensor(),
                                      tvt.Normalize((0.5,0.5,0.5), (0.5,0.5,0.5))]) # normalization

    def __len__(self):
        return len(self.file_list) # data size

    def __getitem__(self, idx):
        file_name = os.path.join(self.data_dir, self.file_list[idx])
        I = io.imread(file_name)
        img = self.transform(I)
        return img, file_name
```

Figure 1: Implementation of the custom dataset

the custom dataset is `PizzaDataset` and its implementation is given in Figure 1. Hierarchy of the dataset folders as follows `pizzas` is the main folder, it has `train` and `eval` subfolders. They consists of train and test pizza images respectively. The location of the main folder should be the same as the program files. As seen in Figure 1, `PizzaDataset` stores file names and when a sample called by `__getitem__`, it reads the image file then passes it through consecutive transformations. First transformation copies the image NumPy object (I did that as a workaround of a bug I encountered.) Second transformation, converts NumPy array to PyTorch tensor and scales it to the range of $[0, 1]$. Third transformation, normalizes the data with respect to the given mean and standard deviation, which are 0.5. In addition to the image tensor, I returned file names as well because I used them later to calculate FID evaluation score. In the next section, I will discuss the implementation of GAN and WGAN models with their training loops.

3 GAN

In the original GAN paper [1], authors used fully connected neural networks to implement generator and discriminator. This is not a good idea for image datasets because thanks to convolutional layers, network become scale invariant and number of parameters reduce drastically. To improve the performance, [4] introduced DCGAN architecture whose generator and discriminator consist of convolution and transpose convolution layers respectively. They are similar to encoder-decoder network architecture. Discriminator can be thought of as encoder and generator can be thought of as decoder whose input is a noise vector. Since its performance is promising, I have implemented the same architecute as DCGAN [4].

Model classes are implemented in `hw7_MehmetBerkSahin.py` module. Generator is implemented as `Generator` class, which is a subclass of `nn.Module`. Its implementation can be seen in Figure 14b. Generator expects a noise vector with the following shape `(batch_size, 100, 1, 1)`. Then,

```
class Generator(nn.Module):
    def __init__(self, chn=3, bias=False, in_dim=100):
        super(Generator, self).__init__()
        # filter sizes: [1024, 512, 256, 128]
        self.chn = chn
        self.up_convs = nn.Sequential(
            # first up layer
            nn.ConvTranspose2d(in_dim, 1024, 4, stride=1, padding=0, bias=bias),
            nn.BatchNorm2d(1024),
            nn.ReLU(True),
            # second up layer
            nn.ConvTranspose2d(1024, 512, 4, stride=2, padding=1, bias=bias),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            # third layer
            nn.ConvTranspose2d(512, 256, 4, stride=2, padding=1, bias=bias),
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            # fourth layer
            nn.ConvTranspose2d(256, 128, 4, stride=2, padding=1, bias=bias),
            nn.BatchNorm2d(128),
            nn.ReLU(True)
        )
        self.last_layer = nn.Sequential(nn.ConvTranspose2d(128, 3, 4, stride=2, padding=1, bias=bias),
                                        nn.Tanh())
    def forward(self, x):
        out = self.last_layer(self.up_convs(x))
        return out
```

Figure 2: `Generator` implementation

it passes this input through two-dimensional transpose convolutions and increase the height and width of the input to obtain an output having equal shape to the original images in the dataset. Doing so, in the training, generator learns the fine details in the dataset. Dimension of the batches increase as follows: $1 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64$. As the images are normalized at the beginning and as it was done by DCGAN paper, I added tanh activation function at the end of the architecture. Furthermore, to increase the performance, I used batch normalization at each layer. In the forward pass, I just passed the input through up convolutions and last layer with activation function. I used the same generator for GAN and WGAN as it will make the comparison of two algorithms easy and DCGAN generator is used for both approaches. Next, I will discuss the implementation of discriminator for GAN.

Discriminator's purpose is classifying images as fake or real so it's a classifier. As the dataset consists of images, it is a good idea to implement CNN for that purpose. I implemented discriminator of DCGAN, which performs well in [4]. Model architecture can be seen in Figure 3. It consists of convolutional layers which reduce the size of the images by `stride` $= 2$. To improve the performance I used batch normalization and different than generator, output of the last convolutional layer passes through sigmoid activation function. Because discriminator estimates the probability of image being fake or real. At the end of the training, if generator is trained well, discriminator should output 0.5 for any image. Meaning that generator generates very good fake samples so discriminator is not able to distinguish them. Next I will explain the GAN class.

Definition of GAN with its `__init__` is given in Figure 4. Initialization of parameters necessary for GAN training are initialized such as beta parameters of Adam optimizer, device (cpu/gpu), learning rate, batch size, length of the noise vector, generator and discriminator objects, and their corresponding Adam optimizer objects. Next, I will explain the `train()` function for GAN. As it is not short, I will explain its important parts for details, reader can look at the source code. `train()` function has two arguments: `num_epochs` and `trainloader`. First one is the number of epochs for training loop and the second one is dataloader for train set. In Figure 5, I do the same weight initialization as in DCGAN paper [4] because it provides better start to training. Then, I initialized dictionaries to save them and plot them later. Lastly, I initialized `fixed_noise` to save

```python
class Discriminator(nn.Module):
    def __init__(self, in_chn, neg_slope=0.2):
        super(Discriminator, self).__init__()
        # filter sizes: [128, 256, 512, 1024]

        self.in_chn = in_chn
        self.convs = nn.Sequential(
            # first down layer
            nn.Conv2d(in_chn, 128, 4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(neg_slope, inplace=True),
            # second down layer
            nn.Conv2d(128, 256, 4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(neg_slope, inplace=True),
            # third down layer
            nn.Conv2d(256, 512, 4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(neg_slope, inplace=True),
            # fourth down layer
            nn.Conv2d(512, 1024, 4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(1024),
            nn.LeakyReLU(neg_slope, inplace=True),
            # fifth layer
            nn.Conv2d(1024, 1, 4, stride=1, padding=0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        out = self.convs(x)
        return out
```

Figure 3: `Discriminator` implementation

```python
class GAN(object):
    def __init__(self, args):

        # set device
        self.device = f"cuda:{args.cuda_idx}" if torch.cuda.is_available() else "cpu"

        # hyperparamters
        self.beta1 = args.beta1
        self.beta2 = args.beta2
        self.batch_size = args.batch_size
        self.lr = args.lr
        self.zdim = args.noise_dim

        # generator & discriminator
        self.generator = Generator(in_dim=args.noise_dim)
        self.discriminator = Discriminator(in_chn=3)

        # train params
        self.opt_d = Adam(params=self.discriminator.parameters(),
                          lr=self.lr, betas=(self.beta1, self.beta2))
        self.opt_g = Adam(params=self.generator.parameters(),
                          lr=self.lr, betas=(self.beta1, self.beta2))
```

Figure 4: `__init__` method of `GAN`

the outputs of the same noise to see the progress of the generator qualitatively. Second part of the implementation can be seen in Figure 6. Train loop consist of two loops. Outer loop iterates over epochs and the inner loop iterates over batches. Firstly, discriminator's gradients are set to 0 in case there may be accumulated gradients from previous steps. Then, forward pass is done over the discriminator for real images. To calculate the first term of the binary cross-entropy loss in equation (1), I entered the predicted probabilites and labels 1 to `nn.BCELoss`. Then, I do back-propagation through the first term of BCE to calculate the gradients related to the first term. Then, to calculate the gradients for the second term in equation BCE, I first generate noise vectors for each sample in the batch, then I create fake images by passing the noise vector through the generator. I pass the fake images to the discriminator to get the estimates of their probabilities of being real. Then, I calculated the second term of BCE by passing the fake images with labels 0 to the `nn.BCELoss` criterion. I back-propagated the loss to calculate the gradients for the second term of BCE. Finally, I updated the discriminator weights with `step()` function. After those steps, discriminator is trained for one step. Then, we need to train the generator to make it generate more realistic fake images. To do so, I first set the accumulated gradients of the generator to 0.

```
# intialize weights
self.discriminator.apply(weight_inits)
self.generator.apply(weight_inits)

# BCE loss
criterion = nn.BCELoss()

target_real = torch.ones(self.batch_size, device=self.device)
target_fake = torch.zeros(self.batch_size, device=self.device)

losses = {"bce_loss" : [],
          "g_loss" : [],
          "d_loss" : []}

mean_probs = {"real_probs" : [],
              "fake_probs" : [],
              "fake_probs_g" : []}

# fixed noise
fixed_noise = torch.FloatTensor(self.batch_size, self.zdim, 1, 1).normal_(0, 1).to(self.device)

counter = 0
```

Figure 5: `GAN.train()` implementation part 1

```
for epoch in range(num_epochs):

    # for each batch
    for i, (imgs, _) in enumerate(train_loader, 1):

        if imgs.shape[0] != self.batch_size:
            continue

        counter += 1

        self.discriminator.zero_grad()
        # get batch
        imgs = imgs.to(self.device)

        # calculate discriminator loss
        pred_probs_r = self.discriminator(imgs).view(-1)
        real_loss = criterion(pred_probs_r, target_real)
        real_loss.backward()

        z = torch.randn(self.batch_size, self.zdim, 1, 1, device=self.device)
        fake_imgs = self.generator(z)
        pred_probs_f = self.discriminator(fake_imgs.detach()).view(-1)
        fake_loss = criterion(pred_probs_f, target_fake)
        fake_loss.backward()
        # total discriminator loss (BCE loss)
        bce_loss = real_loss.item() + fake_loss.item()
        # update weights
        self.opt_d.step()

        # train generator
        self.generator.zero_grad()
        pred_probs_f2 = self.discriminator(fake_imgs).view(-1)
        gen_loss = criterion(pred_probs_f2, target_real)
        gen_loss.backward()
        self.opt_g.step()
```

Figure 6: `GAN.train()` implementation part 2

Then, using the fake images generated previously, I do forward pass through the discriminator. Then, I minimize the second term of equation 1. Here, I want to clarify a possible confusion. In equation 1, optimization is over $-BCE$. So, minimizing the second term of equation 1 is equivalent to maximizing the second term of BCE. To do so, instead of entering fake labels, I enter real labels with fake probabilities to `nn.BCELoss`. Then, I calculate the gradients with `backward()` and update the generator weights. Rest of the lines, which do not appear in Figure 6, consists of saving the results, reporting them, and saving generated images for each 500 iteration.

4 WGAN

GANs are able to generate fake images when they are tuned well to the dataset, model architecture and problem formulation. Doing so is not easy task and requires lot of experiments. And it is very likely to encounter mode collapse problem, which prevent generator from learning the underlying distribution. To mitigate these problems, [2] was introduced. Now, instead of discriminator, we have a critic function and it is constrained to be 1-Lipschitz. I tried weight clipping and gradient

penalty approaches to enforce that constraint [2,3]. The latter performs better as pointed out in [3] so I only reported WGAN with gradient penalty results. WGAN model is implemented as `WGAN` class in `hw7_MehmetBerkSahin.py`. I will explain its implementation step by step. Similar to GAN,

```
class WGAN(object):
    def __init__(self, args):

        # set device
        self.device = f"cuda:{args.cuda_idx}" if torch.cuda.is_available() else "cpu"

        # hyperparamters
        self.beta1 = args.beta1
        self.beta2 = args.beta2
        self.batch_size = args.batch_size
        self.lr = args.lr
        self.gp = args.gp
        self.penalty = args.penalty if self.gp else None
        self.clip = args.clip if not self.gp else None
        self.zdim = args.noise_dim

        # generator & critic
        self.generator = Generator(bias=args.bias, in_dim=args.noise_dim)
        self.critic = Critic(bias=args.bias)

        # train params
        self.opt_c = Adam(params=self.critic.parameters(),
                          lr=self.lr, betas=(self.beta1, self.beta2))
        self.opt_g = Adam(params=self.generator.parameters(),
                          lr=self.lr, betas=(self.beta1, self.beta2))
        self.c_iter = args.c_iter
        self.cl_iter = args.cl_iter
```

Figure 7: `__init__` method of `WGAN`

I initialize the device, optimizer, and training hyperparameters. Different than GAN, there are penalty term, `c_iter`, `cl_iter`, and `clip`. I will elaborate on them further in next paragraphs.

There are common points in WGAN and GAN train loops. I will omit them to avoid repetition, instead, I will explain the different points step by step. `train()` function implementation can be seen in Figure 8. I initialize dictionaries to keep different type of losses. I put if-else statement for

```
def train(self, train_loader, pre_trained=False, num_epochs=30):
    # that's where images will be saved
    if not os.path.exists("wgan_fake_samples"):
        os.mkdir("wgan_fake_samples")
    print(f"training is running on {self.device}!")

    # histories
    metrics = {
        "wasserstein" : [],
        "critic loss" : [],
        "gen loss" : []
    }

    # put the models to device
    self.critic = self.critic.to(self.device)
    self.generator = self.generator.to(self.device)

    if pre_trained == False:
        self.critic.apply(weight_inits)
        self.generator.apply(weight_inits)
    else:
        self.critic.load_state_dict(torch.load("best_wgan_critic"))
        self.generator.load_state_dict(torch.load("best_wgan_generator"))

    # fixed noise
    fixed_noise = torch.FloatTensor(self.batch_size, self.zdim, 1, 1).normal_(0, 1).to(self.device)
    # targets
    one = torch.FloatTensor([1]).to(device=self.device)
    mone = torch.FloatTensor([-1]).to(device=self.device)
    # loss variables
    gen_loss = 0
    critic_loss = 0
    wass_dist = 0
```

Figure 8: `WGAN.train()` implementation part 1

training from pre-trained model or training from scratch (I did not use any pre-trained model in my results.) Furthermore, there is `one` and `mone` tensors, which are important and will be used in back-propagation. Train loop consists of three loops. The first and second loop are the same as

```
# number of generator iterations
gen_iter = 0
for epoch in range(num_epochs):
    data_iter = iter(train_loader)
    idx = 0 # number of iteration in current epoch
    while idx < len(train_loader):
        # activate weights of the critic
        for param in self.critic.parameters():
            param.requires_grad = True

        # that was 500
        c_iter = self.cl_iter if gen_iter < 25 or gen_iter % 500 == 0 else self.c_iter
        c_idx = 0 # number of critic iteration
        # train critic
        while c_idx < c_iter and idx < len(train_loader):
            # update counters
            c_idx += 1
            idx += 1
            # if not doing gradient penalty, then do gradient clipping
            if not self.gp:
                for param in self.critic.parameters():
                    param.data.clamp_(-self.clip, self.clip)

            # calculate real loss
            imgs, _ = next(data_iter)
            imgs = imgs.to(self.device)
```

Figure 9: `WGAN.train()` implementation part 2

GAN loops. Different than that, here I train critics more than once to prevent mode collapse so there is a third loop. It iterates `c_iter` times, which is pre-defined and 5. There is also `cl_iter` and it denotes large number of critic iterations. Meaning that, at first 25 step, critic is trained more than 5 iterations to prevent mode collapse. In my experiments, although this worked for weight clipping, it did not work for gradient penalty so I made it 5 too. Stopping criteria for the inner loop is either finishing one epoch or number of iterations for critic. Before the critic training, I activate the gradient calculation for critic weights to backprop them later. Inside the loop, if weight clipping is on, I clip the weights and extract the next batch. Using the real images

```
            # calculate real loss
            imgs, _ = next(data_iter)
            imgs = imgs.to(self.device)

            self.critic.zero_grad()
            # calculate gradients for real part
            real_loss = self.critic(imgs).mean(dim=0).view(1)
            real_loss.backward(mone)

            # calculate gradients for fake part
            z = torch.randn(imgs.shape[0], self.zdim, 1, 1, device=self.device)
            fake_imgs = self.generator(z)
            fake_loss = self.critic(fake_imgs).mean(dim=0).view(1)
            fake_loss.backward(one)

            # gradient penalty term
            if self.gp:
                ratio = torch.FloatTensor(imgs.shape[0], 1, 1, 1).uniform_(0, 1).to(self.device)
                # interpolated distribution
                int_dist = ratio * imgs + (1 - ratio) * fake_imgs.detach()
                int_dist.requires_grad = True
                c_out = self.critic(int_dist)
                # calculate the gradient for soft constraint
                c_grads = torch.autograd.grad(c_out, int_dist, torch.ones(c_out.size(), device=self.device),
                                              create_graph=True, retain_graph=True)[0]
                #print("LİMBASS CENNETİ:", c_grads.shape)
                c_grads = c_grads.view(c_grads.shape[0], -1)
                penalty = self.penalty * ((c_grads.norm(2, dim=1) - 1) ** 2).mean()
                penalty.backward()

            wass_dist = real_loss - fake_loss
            loss_critic = wass_dist
            self.opt_c.step()

        # deactivate the critic weights
        for param in self.critic.parameters():
            param.requires_grad = False
```

Figure 10: `WGAN.train()` implementation part 3

in the batch, algorithm performs forward pass through the critic. Since the second term in critic loss (6) has a minus sign I add `mone` inside the `backward()` to multiply the gradients by -1. Then, gradients for the second term in equation (6) are calculated. Afterwards, from a noise vector, which is sampled from a normal distribution, generator generates fake images. These images are passed to critic and empirical mean at the first term of critic loss (6) is calculated. Since it is positive

in the critic loss, I passed `one` to the `backward()` function. Then, gradients for the first term are calculated. Lastly, if the gradient penalty mode is on, algorithm calculates the penalty term as follows. First, for each image, it samples a number from uniform distribution between 0 and 1. Then, real and fake samples are interpolated by the sampled epsilons, which is `ratio` in the code. Then, I activate the gradient calculations with respect to interpolated image and they are passed to the critic for forward pass. They gradient of the critic with respect to its input is calculated by using `torch.autograd.grad()` function. Its L2 norm is calculated and it gets multiplied by a scalar, which varies depending on how much we want the function to be smooth. Then, algorithm back-propagates through the penalty term. Lastly, all necessary gradients are calculated and critic weights are updated using those gradients by calling `step()` on `self.opt_c`. Losses are saved and gradient calculation for critic parameters are disabled for generator training. Part 4 of the WGAN `train()` can be seen in Figure 11. First, accumulated gradients of generator from previous steps

```
# train generator
self.generator.zero_grad()
z = torch.randn(imgs.shape[0], self.zdim, 1, 1, device=self.device)
fake_imgs = self.generator(z)
loss_gen = self.critic(fake_imgs).mean().view(1)
loss_gen.backward(mone)
gen_loss = -loss_gen
self.opt_g.step()
gen_iter += 1

# save losses
print(f"[{epoch+1}/{num_epochs}][{idx}/{len(train_loader)}][{gen_iter}] loss_C: {wass_dist.item():.4f}, l
metrics["wasserstein"].append(wass_dist.item())
metrics["critic loss"].append(loss_critic.item())
metrics["gen loss"].append(gen_loss.item())

# save the results per 500 generator iteration
if gen_iter % 500 == 0:
    with torch.no_grad():
        fake = self.generator(fixed_noise).detach().cpu()
        fake = fake.mul(0.5).add(0.5)
        save_image(fake, fp=f"wgan_fake_samples/{gen_iter//500}.jpg", nrow=int(self.batch_size**0.5))
```

Figure 11: `WGAN.train()` implementation part 4

are set to 0. Then, noise vectors from normal distribution are sampled for each sample in batch. Then, fake image is constructed by the generator. That is forward passed to the critic. Then, results are back-propagated through the networks. As the generator tries to maximize the first term in critic loss 6, this is equivalent to minimizing its minus. So, I multiplied the gradients with -1 by passing `mone` inside `backward()`. Then, generator weights are updates by calling `step()` on `self.opt_g`. Other lines consist of saving the losses and saving the generated images at each 500 iteration. Next, I will discuss the experiments and results. For the rest of the code, reader can take a look at the source code. It is well-commented.

5 Results & Discussion

I conducted three experiments for GAN, WGAN with weight clipping and WGAN with gradient penalty. I observed that gradient penalty is better at enforcing the 1-Lipschitz constraint than gradient clipping so I report the results for WGAN with gradient penalty. First, I plot the adversarial losses over training iterations for both generator and discriminator/critic and for both BCE-GAN and WGAN. Then, I evaluate the performance of the algorithms both quantitatively and qualitatively. Then, I present sample images from the generators on 4×4 grid. All experiments were run in NVIDIA A100 GPU. I do not guarantee training for GANs will run in CPU without any error.

5.1 BCE-GAN Results

I used the following hyperparameters to train DCGAN with BCE loss: Adam optimizer is used with $(\beta_1, \beta_2) = (0.5, 0.999)$, learning rate is $1e-4$, batch size is 64, dimensionality of the noise vector is 100, and number of epochs is 125. I tried to run the algorithm for 750 epochs as I did for WGAN but GAN started to diverge after 125^{th} epoch. Generator's and discriminator's adversarial

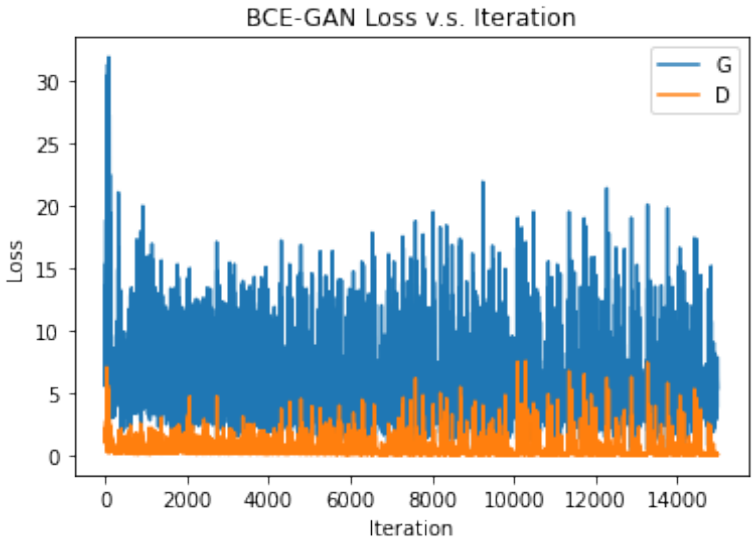


Figure 12: BCE-GAN loss of generator and discriminator

losses can be seen in Figure 12. G and D denote the generator and discriminator losses respectively. They are noisy because as opposed to previous tasks, objective of this task is adversarial. So, one's performance affect another and since both sides are being trained, training procedure is noisy. For example, discriminator's loss can be 0 but this may be due to the fact that generator may not generate realistic samples. It may not be because discriminator is trained very well. Due to such uncertainities, loss plot is noisy. However, it is obvious that both model started from a high loss and decreased rapidly at the beginning. To understand if the generator learns the underlying distribution of the dataset, we need to make quantitative and qualitative assesments.

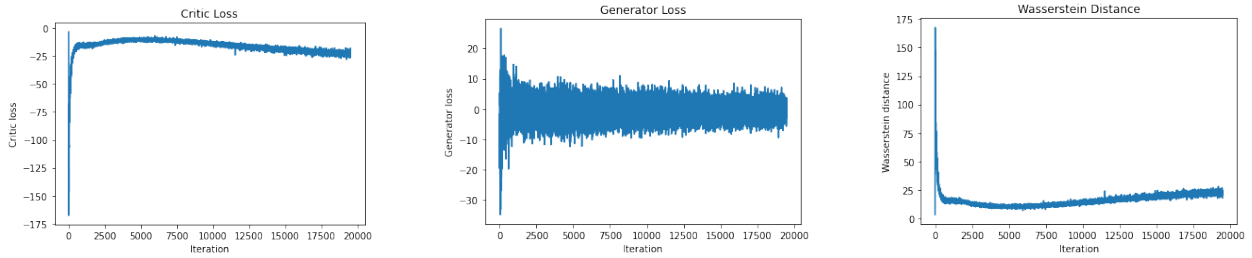
I make a gif of the generated images at each 500 iteration. You can find the video in the homework submission folder! In addition to that I added the 4×4 grid to the report as well for completeness. Sample fake pizzas which are generated by BCE-GAN generator can be found in Figure 13. As in the assignment, we are asked to display 4×4 grid, I did not add more but larger grid can be found in the Appendix section. As it can be seen in the figure, most of the pizzas' shapes are generated successfully, albeit with small errors. Furthermore, colors and textures inside pizzas resemble real pizzas. For example, image at second row and third column look as if it is a real pizza. Although there are some images without any blur or deformations, few samples include them. Moreover, pizzas generated in Figure 13 are not the same type. Model generated various types of pizzas. This implies that model learned the dataset distribution to some extent. Qualitative evaluations show that model learns the data distribution to some extent. To evaluate the performance quantitatively, I used the code template given in the assignment to calculate the FID score of fake images generated by BCE-GAN. **I chose 1k samples from real images and generated 1k images randomly. FID score was found to be 110.2647 for BCE-GAN**.

(a) Real images from train dataset (b) Fake images generated by BCE-GAN

Figure 13: Real images v.s. BCE-GAN fake images

5.2 WGAN with Gradient Penalty

I used the following hyperparameters to train DCGAN with Wasserstein distance loss. Adam optimizer is used with $(\beta_1, \beta_2) = (0.5, 0.999)$, `c_iter` is 5, `cl_iter` is 5, penalty weight is 10, learning rate is $1e{-}4$, batch size is 64, and dimension of the noise vector is 100. As this model did not diverge and more stable than BCE-GAN, I run it for 750 epochs. Generator and discriminator adversarial losses with wasserstein distance over iterations can be seen in Figure 14. First observation is that



(a) Critic loss v.s. iteration (b) Generator loss v.s. iteration (c) Wasserstein distance v.s. iteration

Figure 14: Different metrics for WGAN GP at each iteration

wasserstein distance decreased tremendously at the beginning of the training and then it increased slightly. Although it increased, this neither indicates algorithm diverges or lack of learning because wasserstein distance in the training is only empirical estimate of the real wasserstein distance. For it to be 100% accurate, samples should be distributed uniformly and critic should be 100% accurate, which are not the case obviously. So, it is more reliable to asses the performance of the algorithm over FID metric and by visualizing samples. Moreover, the noise in the generator loss plot is due to the fact that wasserstein estimation is noisy because critic's performance is uncertain. And we can that critic's loss increased in Figure 14a. That implies generator learned to generate realistic

fake images. I plotted these in one plot and it can be seen in Figure 15. In Figure 15, G, W,

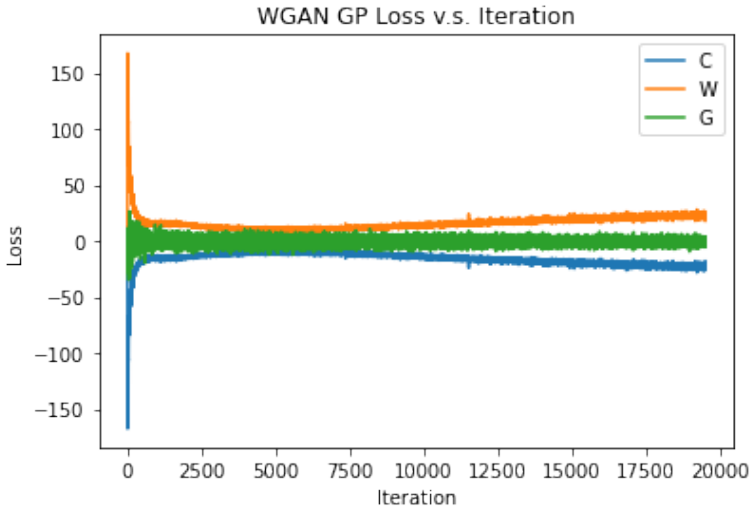


Figure 15: WGAN GP loss of generator, critic, and wasserstein distance

and C denote generator loss, wasserstein distance and critic loss respectively. One observation for WGAN losses is that although it is noisy, it is not as noisy as GAN losses. I think this supports the fact that WGANs are more stable than GANs, especially when 1-Lipschitzness constraint is enforced. **I made a gif of the generated images at each 500 iteration. You can find the video in the homework folder!** Additionally, I added 8×8 grid and 4×4 grid to the report for completeness. First one is in the Appendix and the second one is in Figure 16. As in Figure 16, fake pizzas generated by WGAN GP are more realistic than fake images generated by BCE GAN in Figure 13. In Figure 16, there are various type of pizzas from different perspectives. For example, in first row second column, there is a mixed pizza. In second row, fourth column there is a very good margherita from top and close view. Similar margherita can be seen at fourth row and fourth column but from a different perspective (far away). Although there are some distortions in some images, quality of the fake samples are sufficiently good for the purpose of this homework. To evaluate the performance of WGAN GP, **I chose 1k samples from real images and generated 1k images randomly. FID score was found to be 91.9196 for WGAN GP**, which is better than GAN. This was expected because instead of JS, we minimized Wasserstein distance by incorporating 1-Lipschitzness of the critic function.

To sum, both BCE-GAN and WGAN GP models work fine. Their loss function imply generator learns the distribution of the data. And the loss functions are very similar to the corresponding loss functions in Prof. Kak's slides so that is another factor verifying the fact that my implementations are correct. Fake images seem realistic and they look like real pizzas, albeit with small errors. To increase the performance of the model and learn the data distribution better, one may implement spectral norm to enforce the 1-Lipschitzness constraint on the objective. It is not a soft constraint like gradient penalty and it normalizes the gradients with the largest singular value. In addition to that, I did not test other types of architectures such as skip-connections or different convolutional layers (different kernel sizes, stride, padding, etc.) because I think current models' performances are good enough for this homework.

(a) Real images from train dataset (b) Fake images generated by WGAN GP

Figure 16: Real images v.s. WGAN GP fake images

WGAN's performance is better than BCE-GAN. This can be seen from both quantitative and qualitative analysis. FID of WGAN is less than GAN's FID. And if we compare the fake images that are generated by those models, images generated by WGAN have better resolution, texture and shape therefore they are more realistic. Some of the GAN's fake images look as if they are drawn with brush and they have some blurs. However, in WGAN's images, they are sharper and there are not too much blurs. The distinction is clearer if the reader look at the 8×8 grids in the Appendix. Furthermore, WGAN's speed of learning is much faster than GAN's learning speed. If you look at the gif videos that I put in homework file, the first outputs of WGAN make much more sense and look realistic than GAN's first outputs. Lastly, as WGAN is more stable than GAN, I can run it for 750 epochs but I could run GAN for only 125 epochs. After that it started to diverge.

6 How to run the code?

To run the code properly, one needs to put the pizza dataset under the same location as the homework program files. And folder hierarchies should follow the convention explained in Section 2. To run the training of the BCE-GAN with recommended parameters, one can run the following command:

```
python hw7_MehmetBerkSahin --beta1 0.5 --beta2 0.999 --lr 1e-4 --batch_size 64
            --noise_dim 100 --num_epochs 125 --train_wgan False
```

To run the training of the WGAN GP, one can run the following command in terminal:

```
python hw7_MehmetBerkSahin --beta1 0.5 --beta2 0.999 --lr 1e-4 --batch_size 64
--noise_dim 100 --num_epochs 750 --train_wgan True --c_iter 5 --cl_iter 5 --gp True
                                --penalty 10
```

All loss plots and figures including fake samples will be saved to the local disk.

7 Lessons Learned

In this homework, I learned how the generator, discriminator and the critic are trained to beat each other in minimax game, I experimented with different GAN types including GAN with BCE, WGAN with weight clipping, and WGAN with gradient penalty. I observed the evolution of the generated samples and saved them as gif files. If reader wants, s/he can check the homework file that I submitted. I experimented how different normalizations such as `InstanceNorm` and `BatchNorm` affect the performance of GANs. I observed how model collapse occur for GANs with wrong hyperparameters and learned that they should be tuned properly.

8 Appendix

8.1 Sample Real Images

Figure 17: Real images from train dataset

8.2 Sample Fake Images

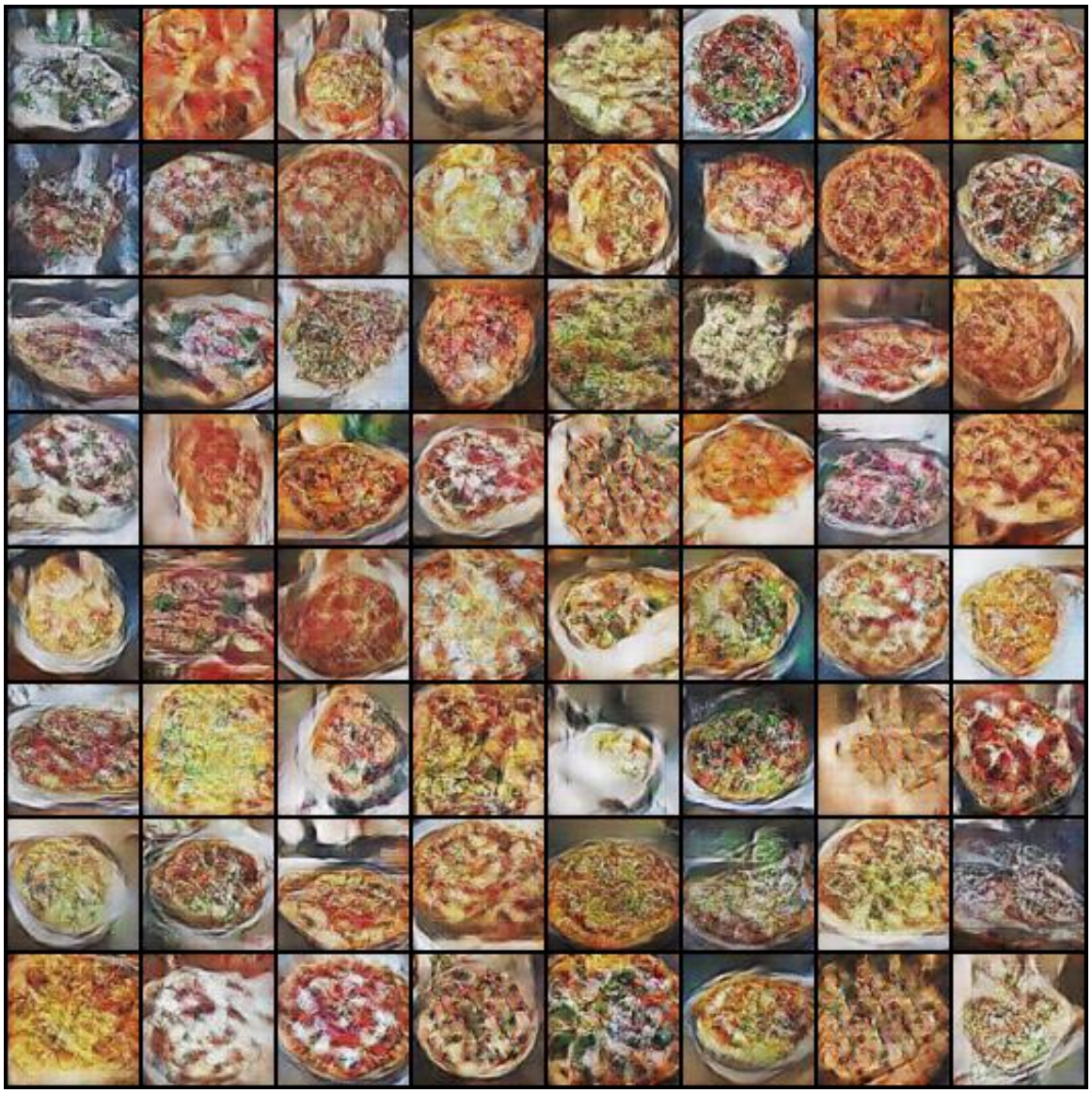


Figure 18: Fake images generated by BCE-GAN

Figure 19: Fake images generated by WGAN GP

8.3 Source Code

8.3.1 dataset.py

```
import torch.nn as nn
import os
from skimage import io
import torchvision.transforms as tvt

class PizzaDataset(nn.Module):

    def __init__(self, path="pizza", train=True):
        super(PizzaDataset, self).__init__()
        self.data_dir = os.path.join(path, "train") if train else os.path.join(path, "eval")
        self.file_list = os.listdir(self.data_dir) # list of file names
        # copy image -> tensor -> normalize
        self.transform = tvt.Compose([tvt.Lambda(lambda img: img.copy()),
                                      tvt.ToTensor(),
                                      tvt.Normalize((0.5,0.5,0.5), (0.5,0.5,0.5))]) # normalization

    def __len__(self):
        return len(self.file_list) # data size
```

```
    def __getitem__(self, idx):
        file_name = os.path.join(self.data_dir, self.file_list[idx])
        I = io.imread(file_name)
        img = self.transform(I)
        return img, file_name

# test code
if __name__ == "__main__":

    print("dataset is being generated...")
    train_data = PizzaDataset(path="pizzas", train=True)
    test_data = PizzaDataset(path="pizzas", train=False)
    print("train and test data were constructed!")
    print(f"train length: {len(train_data)}")
    print(f"test length: {len(test_data)}")

    a = 5
    print(len(train_data[0]))
```

8.3.2 utils.py

```
from pytorch_fid.fid_score import calculate_activation_statistics, calculate_frechet_distance
from pytorch_fid.inception import InceptionV3
import torch
from torchvision.utils import save_image
import os


def frechet_value(real_paths, fake_paths, device, dims=2048):
    # load InceptionV3 model
    block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
    model = InceptionV3([block_idx]).to(device)
    # calculate statistics for both batch
    m1, s1 = calculate_activation_statistics(real_paths, model, device=device)
    m2, s2 = calculate_activation_statistics(fake_paths, model, device=device)
    # calculate frechet value
    fid_value = calculate_frechet_distance(m1, s1, m2, s2)
    return fid_value

def calc_frechet(model, dataset, size=64):
    device = "cuda" if torch.cuda.is_available() else "cpu"
    # sample real images randomly
    perm = torch.randperm(len(dataset))
    idxs = perm[:size]
    # img path names
    real_paths = [dataset[i][1] for i in idxs]
    # create fake images
    z = torch.randn(size, 100, 1, 1)
    fake_imgs = model.generator(z).mul(0.5).add(0.5)
    # create directory for fake images
    folder_name = "fake_images"
    if not os.path.exists(folder_name):
        os.mkdir(folder_name)
    # save fake images
    fake_paths = []
    for idx in range(fake_imgs.shape[0]):
        f = os.path.join(folder_name, f"fake_img{idx}.jpg")
        save_image(fake_imgs[idx], fp=f)
        fake_paths.append(f)
    # calculate frechet value
    fid = frechet_value(real_paths, fake_paths, device)
    return fid


# test code
if __name__ == "__main__":
    real_paths = ["pizzas/train/01001.jpg", "pizzas/train/01002.jpg"]
    fake_paths = ["pizzas/train/01003.jpg", "pizzas/train/01004.jpg"]

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

    fid_val = frechet_value(real_paths, fake_paths, device, dims=2048)
    print(f"frechet value:", fid_val)
```

8.3.3 hw7_MehmetBerkSahin.py

```
import torch
import torch.nn as nn
import pickle
from torch.optim import Adam, RMSprop
import time
from torch.utils.data import Dataset, DataLoader
from torchvision.utils import make_grid, save_image
import torchvision.transforms as tvt
```

```
from dataset import PizzaDataset
import torch.nn.functional as F
import os
import argparse
import random
import numpy as np
from utils import calc_frechet

class Generator(nn.Module):
    def __init__(self, chn=3, bias=False, in_dim=100):
        super(Generator, self).__init__()
        # filter sizes: [1024, 512, 256, 128]
        self.chn = chn
        self.up_convs = nn.Sequential(
            # first up layer
            nn.ConvTranspose2d(in_dim, 1024, 4, stride=1, padding=0, bias=bias),
            nn.BatchNorm2d(1024),
            nn.ReLU(True),
            # second up layer
            nn.ConvTranspose2d(1024, 512, 4, stride=2, padding=1, bias=bias),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            # third layer
            nn.ConvTranspose2d(512, 256, 4, stride=2, padding=1, bias=bias),
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            # fourth layer
            nn.ConvTranspose2d(256, 128, 4, stride=2, padding=1, bias=bias),
            nn.BatchNorm2d(128),
            nn.ReLU(True)
        )
        self.last_layer = nn.Sequential(nn.ConvTranspose2d(128, 3, 4, stride=2, padding=1, bias=bias),
                                        nn.Tanh())
    def forward(self, x):
        out = self.last_layer(self.up_convs(x))
        return out

class Discriminator(nn.Module):
    def __init__(self, in_chn, neg_slope=0.2):
        super(Discriminator, self).__init__()
        # filter sizes: [128, 256, 512, 1024]

        self.in_chn = in_chn
        self.convs = nn.Sequential(
            # first down layer
            nn.Conv2d(in_chn, 128, 4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(neg_slope, inplace=True),
            # second down layer
            nn.Conv2d(128, 256, 4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(neg_slope, inplace=True),
            # third down layer
            nn.Conv2d(256, 512, 4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(neg_slope, inplace=True),
            # fourth down layer
            nn.Conv2d(512, 1024, 4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(1024),
            nn.LeakyReLU(neg_slope, inplace=True),
            # fifth layer
            nn.Conv2d(1024, 1, 4, stride=1, padding=0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        out = self.convs(x)
        return out

class Critic(nn.Module):
    def __init__(self, neg_slope=0.2, bias=False):
        super(Critic, self).__init__()
        # filter sizes: [128, 256, 512, 1024]
        self.convs = nn.Sequential(
            # first down layer
            nn.Conv2d(3, 128, 4, stride=2, padding=1, bias=bias),
            #nn.BatchNorm2d(128),
            nn.LeakyReLU(neg_slope, inplace=True),
            # second down layer
            nn.Conv2d(128, 256, 4, stride=2, padding=1, bias=bias),
            #nn.BatchNorm2d(256),
            nn.LeakyReLU(neg_slope, inplace=True),
            # third down layer
            nn.Conv2d(256, 512, 4, stride=2, padding=1, bias=bias),
            #nn.BatchNorm2d(512),
            nn.LeakyReLU(neg_slope, inplace=True),
            # fourth down layer
            nn.Conv2d(512, 1024, 4, stride=2, padding=1, bias=bias),
            #nn.BatchNorm2d(1024),
            nn.LeakyReLU(neg_slope, inplace=True),
            # fifth layer
            nn.Conv2d(1024, 1, 4, stride=1, padding=0, bias=bias),
```

```
        )

    def forward(self, x):
        out = self.convs(x)
        return out

class WGAN(object):
    def __init__(self, args):

        # set device
        self.device = f"cuda:{args.cuda_idx}" if torch.cuda.is_available() else "cpu"

        # hyperparamters
        self.beta1 = args.beta1
        self.beta2 = args.beta2
        self.batch_size = args.batch_size
        self.lr = args.lr
        self.gp = args.gp
        self.penalty = args.penalty if self.gp else None
        self.clip = args.clip if not self.gp else None
        self.zdim = args.noise_dim

        # generator & critic
        self.generator = Generator(bias=args.bias, in_dim=args.noise_dim)
        self.critic = Critic(bias=args.bias)

        # train params
        self.opt_c = Adam(params=self.critic.parameters(),
                          lr=self.lr, betas=(self.beta1, self.beta2))
        self.opt_g = Adam(params=self.generator.parameters(),
                          lr=self.lr, betas=(self.beta1, self.beta2))
        self.c_iter = args.c_iter
        self.cl_iter = args.cl_iter

    def train(self, train_loader, pre_trained=False, num_epochs=30):
        # that's where images will be saved
        if not os.path.exists("wgan_fake_samples"):
            os.mkdir("wgan_fake_samples")
        print(f"training is running on {self.device}!")

        # histories
        metrics = {
            "wasserstein" : [],
            "critic loss" : [],
            "gen loss" : []
        }

        # put the models to device
        self.critic = self.critic.to(self.device)
        self.generator = self.generator.to(self.device)

        if pre_trained == False:
            self.critic.apply(weight_inits)
            self.generator.apply(weight_inits)
        else:
            self.critic.load_state_dict(torch.load("best_wgan_critic"))
            self.generator.load_state_dict(torch.load("best_wgan_generator"))

        # fixed noise
        fixed_noise = torch.FloatTensor(self.batch_size, self.zdim, 1, 1).normal_(0, 1).to(self.device)
        # targets
        one = torch.FloatTensor([1]).to(device=self.device)
        mone = torch.FloatTensor([-1]).to(device=self.device)
        # loss variables
        gen_loss = 0
        critic_loss = 0
        wass_dist = 0
        # number of generator iterations
        gen_iter = 0
        for epoch in range(num_epochs):
            data_iter = iter(train_loader)
            idx = 0 # number of iteration in current epoch
            while idx < len(train_loader):
                # activate weights of the critic
                for param in self.critic.parameters():
                    param.requires_grad = True

                # that was 500
                c_iter = self.cl_iter if gen_iter < 25 or gen_iter % 500 == 0 else self.c_iter
                c_idx = 0 # number of critic iteration
                # train critic
                while c_idx < c_iter and idx < len(train_loader):
                    # update counters
                    c_idx += 1
                    idx += 1
                    # if not doing gradient penalty, then do gradient clipping
                    if not self.gp:
                        for param in self.critic.parameters():
                            param.data.clamp_(-self.clip, self.clip)

                    # calculate real loss
                    imgs, _ = next(data_iter)
```

```
                imgs = imgs.to(self.device)

                self.critic.zero_grad()
                # calculate gradients for real part
                real_loss = self.critic(imgs).mean(dim=0).view(1)
                real_loss.backward(mone)

                # calculate gradients for fake part
                z = torch.randn(imgs.shape[0], self.zdim, 1, 1, device=self.device)
                fake_imgs = self.generator(z)
                fake_loss = self.critic(fake_imgs).mean(dim=0).view(1)
                fake_loss.backward(one)

                # gradient penalty term
                if self.gp:
                    ratio = torch.FloatTensor(imgs.shape[0], 1, 1, 1).uniform_(0, 1).to(self.device)
                    # interpolated distribution
                    int_dist = ratio * imgs + (1 - ratio) * fake_imgs.detach()
                    int_dist.requires_grad = True
                    c_out = self.critic(int_dist)
                    # calculate the gradient for soft constraint
                    c_grads = torch.autograd.grad(c_out, int_dist, torch.ones(c_out.size(), device=self.device),
                                                  create_graph=True, retain_graph=True)[0]
                    #print("LIMBASS CENNETI:", c_grads.shape)
                    c_grads = c_grads.view(c_grads.shape[0], -1)
                    penalty = self.penalty * ((c_grads.norm(2, dim=1) - 1) ** 2).mean()
                    penalty.backward()

                wass_dist = real_loss - fake_loss
                loss_critic = -wass_dist
                self.opt_c.step()

            # deactivate the critic weights
            for param in self.critic.parameters():
                param.requires_grad = False

            # train generator
            self.generator.zero_grad()
            z = torch.randn(imgs.shape[0], self.zdim, 1, 1, device=self.device)
            fake_imgs = self.generator(z)
            loss_gen = self.critic(fake_imgs).mean().view(1)
            loss_gen.backward(mone)
            gen_loss = -loss_gen
            self.opt_g.step()
            gen_iter += 1

            # save losses
            print(f"[{epoch+1}/{num_epochs}][{idx}/{len(train_loader)}][{gen_iter}] loss_C: {wass_dist.item():.4f}, loss_G: {gen_loss.item():.4f} wass dist: {wass_dis
            metrics["wasserstein"].append(wass_dist.item())
            metrics["critic loss"].append(loss_critic.item())
            metrics["gen loss"].append(gen_loss.item())

            # save the results per 500 generator iteration
            if gen_iter % 500 == 0:
                with torch.no_grad():
                    fake = self.generator(fixed_noise).detach().cpu()
                    fake = fake.mul(0.5).add(0.5)
                    save_image(fake, fp=f"wgan_fake_samples/{gen_iter//500}.jpg", nrow=int(self.batch_size**0.5))

        # save the results
        pickle.dump(metrics, open("wgan_results.pkl", "wb"))
        torch.save(self.generator.state_dict(), "best_wgan_generator")
        torch.save(self.critic.state_dict(), "best_wgan_critic")
        print("results and generator model are saved!")
        return metrics


class GAN(object):
    def __init__(self, args):

        # set device
        self.device = f"cuda:{args.cuda_idx}" if torch.cuda.is_available() else "cpu"

        # hyperparamters
        self.beta1 = args.beta1
        self.beta2 = args.beta2
        self.batch_size = args.batch_size
        self.lr = args.lr
        self.zdim = args.noise_dim

        # generator & discriminator
        self.generator = Generator(in_dim=args.noise_dim)
        self.discriminator = Discriminator(in_chn=3)

        # train params
        self.opt_d = Adam(params=self.discriminator.parameters(),
                          lr=self.lr, betas=(self.beta1, self.beta2))
        self.opt_g = Adam(params=self.generator.parameters(),
                          lr=self.lr, betas=(self.beta1, self.beta2))

    def train(self, train_loader, num_epochs):
```

```
# that's where images will be saved
if not os.path.exists("gan_fake_samples"):
    os.mkdir("gan_fake_samples")
print(f"training is running on {self.device}!")

# put the models to device
self.discriminator.to(self.device)
self.generator.to(self.device)

# intialize weights
self.discriminator.apply(weight_inits)
self.generator.apply(weight_inits)

# BCE loss
criterion = nn.BCELoss()

target_real = torch.ones(self.batch_size, device=self.device)
target_fake = torch.zeros(self.batch_size, device=self.device)

losses = {"bce_loss" : [],
          "g_loss" : [],
          "d_loss" : []}

mean_probs = {"real_probs" : [],
              "fake_probs" : [],
              "fake_probs_g" : []}

# fixed noise
fixed_noise = torch.FloatTensor(self.batch_size, self.zdim, 1, 1).normal_(0, 1).to(self.device)

counter = 0

for epoch in range(num_epochs):

    # for each batch
    for i, (imgs, _) in enumerate(train_loader, 1):

        if imgs.shape[0] != self.batch_size:
            continue

        counter += 1

        self.discriminator.zero_grad()
        # get batch
        imgs = imgs.to(self.device)

        # calculate discriminator loss
        pred_probs_r = self.discriminator(imgs).view(-1)
        real_loss = criterion(pred_probs_r, target_real)
        real_loss.backward()

        z = torch.randn(self.batch_size, self.zdim, 1, 1, device=self.device)
        fake_imgs = self.generator(z)
        pred_probs_f = self.discriminator(fake_imgs.detach()).view(-1)
        fake_loss = criterion(pred_probs_f, target_fake)
        fake_loss.backward()
        # total discriminator loss (BCE loss)
        bce_loss = real_loss.item() + fake_loss.item()
        # update weights
        self.opt_d.step()

        # train generator
        self.generator.zero_grad()
        pred_probs_f2 = self.discriminator(fake_imgs).view(-1)
        gen_loss = criterion(pred_probs_f2, target_real)
        gen_loss.backward()
        self.opt_g.step()

        # report and save the losses
        print(f"[{i + epoch * len(train_loader)}/{len(train_loader) * num_epochs} ITER] discriminator loss: {bce_loss:.4f}, generator loss: {gen_loss.item():.4f}"

        # save the losses to history
        losses["d_loss"].append(bce_loss)
        losses["g_loss"].append(gen_loss.item())
        # save mean probabilities
        mean_probs["real_probs"].append(pred_probs_r.mean().item())
        mean_probs["fake_probs"].append(pred_probs_f.mean().item())
        mean_probs["fake_probs_g"].append(pred_probs_f2.mean().item())

        if counter % 500 == 0:
            with torch.no_grad():
                fake = self.generator(fixed_noise).detach().cpu()
                fake = fake.mul(0.5).add(0.5)
                save_image(fake, fp=f"gan_fake_samples/{counter//500}.jpg", nrow=int(self.batch_size**0.5))

# save the results to local disk
results = {"losses" : losses,
           "probs" : mean_probs}

pickle.dump(results, open("gan_results.pkl", "wb"))
torch.save(self.generator.state_dict(), "best_gan_generator")
torch.save(self.discriminator.state_dict(), "best_gan_discriminator")
```

```
        print("results and generator model are saved!")
        return results

# generic functions

def weight_inits(model):
    class_name = model.__class__.__name__
    if class_name.find('Conv') != -1:
        nn.init.normal_(model.weight.data, 0.0, 0.02)
    elif class_name.find('BatchNorm') != -1:
        nn.init.normal_(model.weight.data, 1.0, 0.02)
        nn.init.constant_(model.bias.data, 0)

def save_imgs(generator, z, file_name="images.jpg", best_gen="best_generator"):
    # load best generator
    generator.load_state_dict(torch.load(best_gen))
    generator = generator.to('cpu')
    generator.eval()
    imgs = generator(z)
    imgs = imgs.mul(0.5).add(0.5)
    #grid = make_grid(imgs, nrow=8)
    save_image(imgs, file_name, nrow=8)
    print("generated images are saved")

# test code
if __name__ == "__main__":

    parser = argparse.ArgumentParser()
    # common parameters in train
    parser.add_argument("--batch_size", default=64, type=int, help="batch size for SGD")
    parser.add_argument("--lr", default=1e-4, type=float, help="learning rate")
    parser.add_argument("--cuda_idx", default=0, type=int, help="cuda id")
    parser.add_argument("--num_epochs", default=30, type=int, help="number of epochs for GAN training")
    parser.add_argument("--pre_trained", default=False, type=eval, help="start from pre-trained weights")
    parser.add_argument("--bias", default=False, type=eval, help="activate bias in CNN layers")
    # optimizer
    parser.add_argument("--beta1", default=0.5, type=float, help="beta 1 of ADAM")
    parser.add_argument("--beta2", default=0.999, type=float, help="beta 2 of ADAM")
    parser.add_argument("--c_iter", default=5, type=int, help="num. of iteration for critic")
    parser.add_argument("--cl_iter", default=5, type=int, help="large number for iteration at the beginning")
    # (W)GAN train parameters
    parser.add_argument("--gp", default=False, type=eval, help="activate gradient penalty")
    parser.add_argument("--noise_dim", default=100, type=int, help="number of dimension of noise vector")
    parser.add_argument("--train_wgan", default=True, type=eval, help="train wgan or gan")
    # WGAN_C parameter
    parser.add_argument("--clip", default=0.005, type=float, help="clipping bound")
    # WGAN_GP parameter
    parser.add_argument("--penalty", default=10, type=float, help="gradient penalty for WGAN")

    args = parser.parse_args()

    """
    Recommended GAN parameters
    beta1           : 0.5
    beta2           : 0.999
    learning rate   : 1e-4
    batch size      : 64
    noise dim       : 100
    num_epochs      : 125
    """

    """
    Recommended WGAN GP parameters
    beta1           : 0.5
    beta2           : 0.999
    c_iter          : 5
    cl_iter         : 5
    penalty         : 10
    learning rate   : 1e-4
    batch size      : 64
    noise dim       : 100
    num_epochs      : 750
    """

    # for reproducible results
    seed = 101
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    np.random.seed(seed)
    torch.backends.cudnn.deterministic=True
    torch.backends.cudnn.benchmarks=False
    torch.autograd.set_detect_anomaly(True)

    if args.gp:
        print("gradient penalty is activated!")
    else:
        print("gradient clipping is activated!")
    # get the data
    train_data = PizzaDataset(path="pizzas", train=True)
    test_data = PizzaDataset(path="pizzas", train=False)
    print("data is loaded!")
```

```
# get dataloaders
train_loader = DataLoader(train_data, batch_size=args.batch_size, shuffle=True)


if not args.train_wgan:
    # train and test GAN
    model = GAN(args)
    print("start training...")
    results = model.train(train_loader, args.num_epochs)
    print("training is successful!")
    # save the losses
    pickle.dump(results, open("gan_results.pkl", "wb"))
    print("results were saved!")
    z = torch.randn(16, 100, 1, 1)
    save_imgs(model.generator, z, file_name="GAN_imgs.jpg", best_gen="best_gan_generator")
else:
    # train and test WGAN
    model = WGAN(args)
    print("start training...")
    #results = model.train(train_loader, num_epochs=args.num_epochs)
    print("training is successful!")
    # save the results
    pickle.dump(results, open("wgan_results.pkl", "wb"))
    z = torch.randn(64, 100, 1, 1)
    save_imgs(model.generator, z, file_name="WGAN_imgs.jpg", best_gen="best_wgan_generator")


# calculate FID
model.generator.load_state_dict(torch.load("best_wgan_generator"))
fid = calc_frechet(model, train_data, size=1000)
print("frechet value:", fid)
```

References

[1] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[2] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan, 2017.

[3] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein gans, 2017.

[4] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2016.