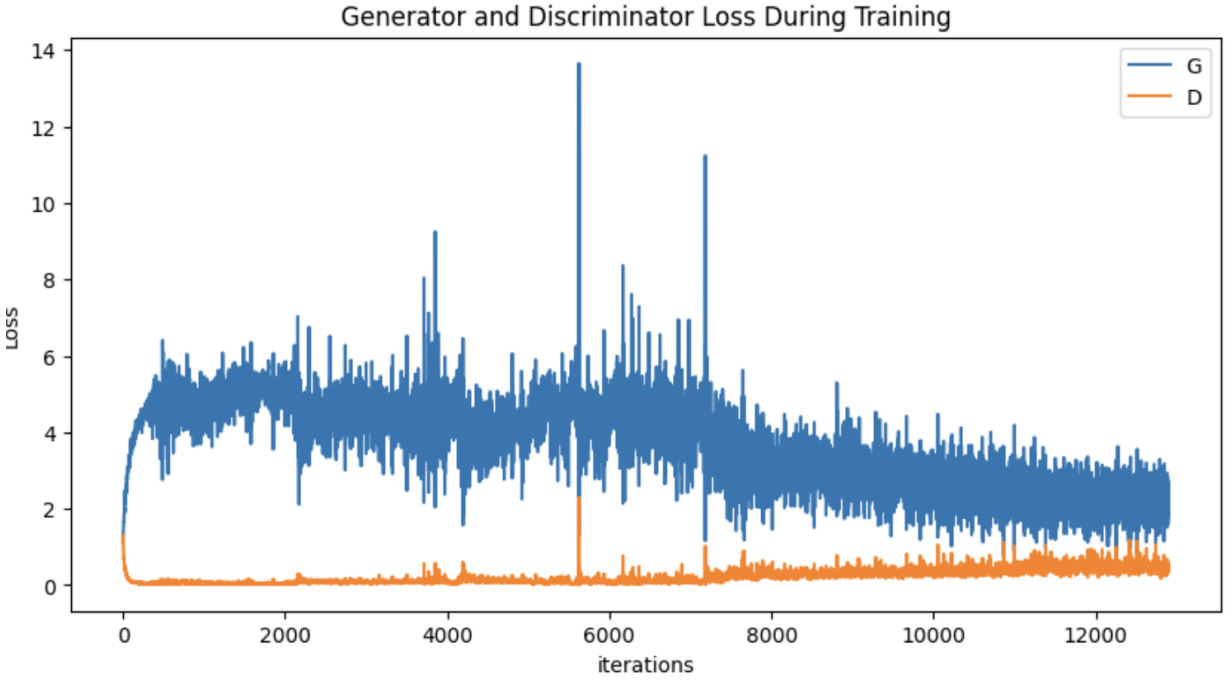
April 7, 2023

This assignment focuses on building two different GAN networks. The first is a BCE-GAN, which uses Binary Cross Entropy loss in the discriminator while the second is a W-GAN, which uses Wasserstein distance to evaluate loss (and gauge how far the generated distribution is from the training distribution). After training for 100 epochs on each network, the respective trained generators were used to generate 1,000 new “fake” images. Using these 1,000 images and the 1,000 evaluation images given to us, the FID scores were computed. Lastly, 16 fake images from each network are presented. At the end of the report, the code for this assignment is presented.

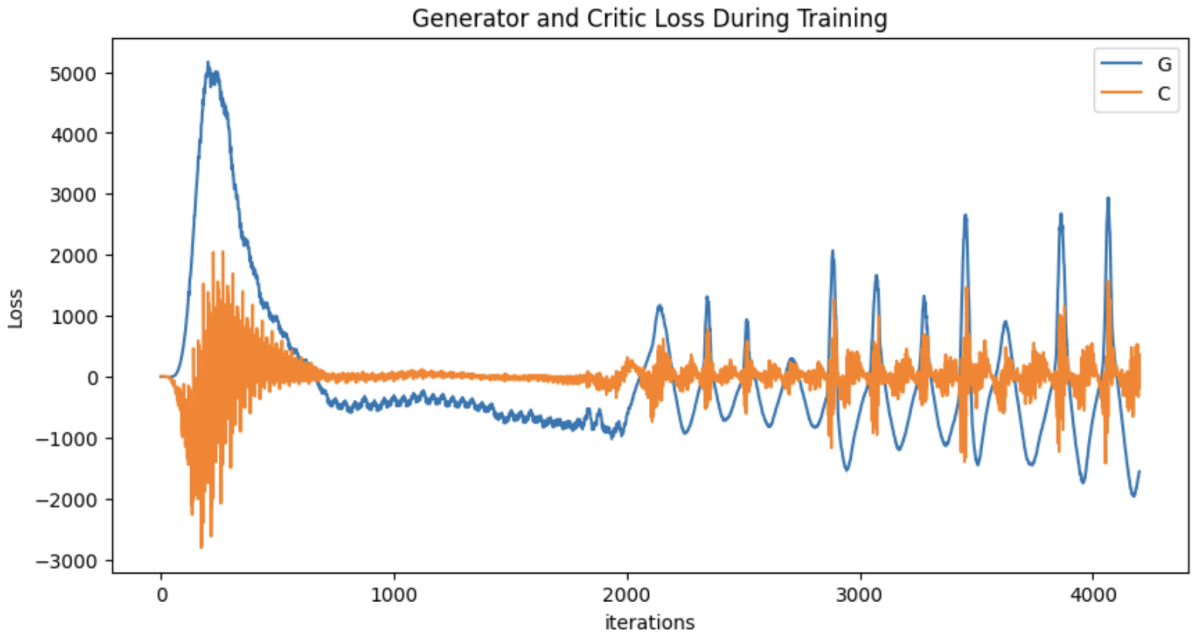
Plot of Adversarial Losses over Training Iterations:

BCE-GAN



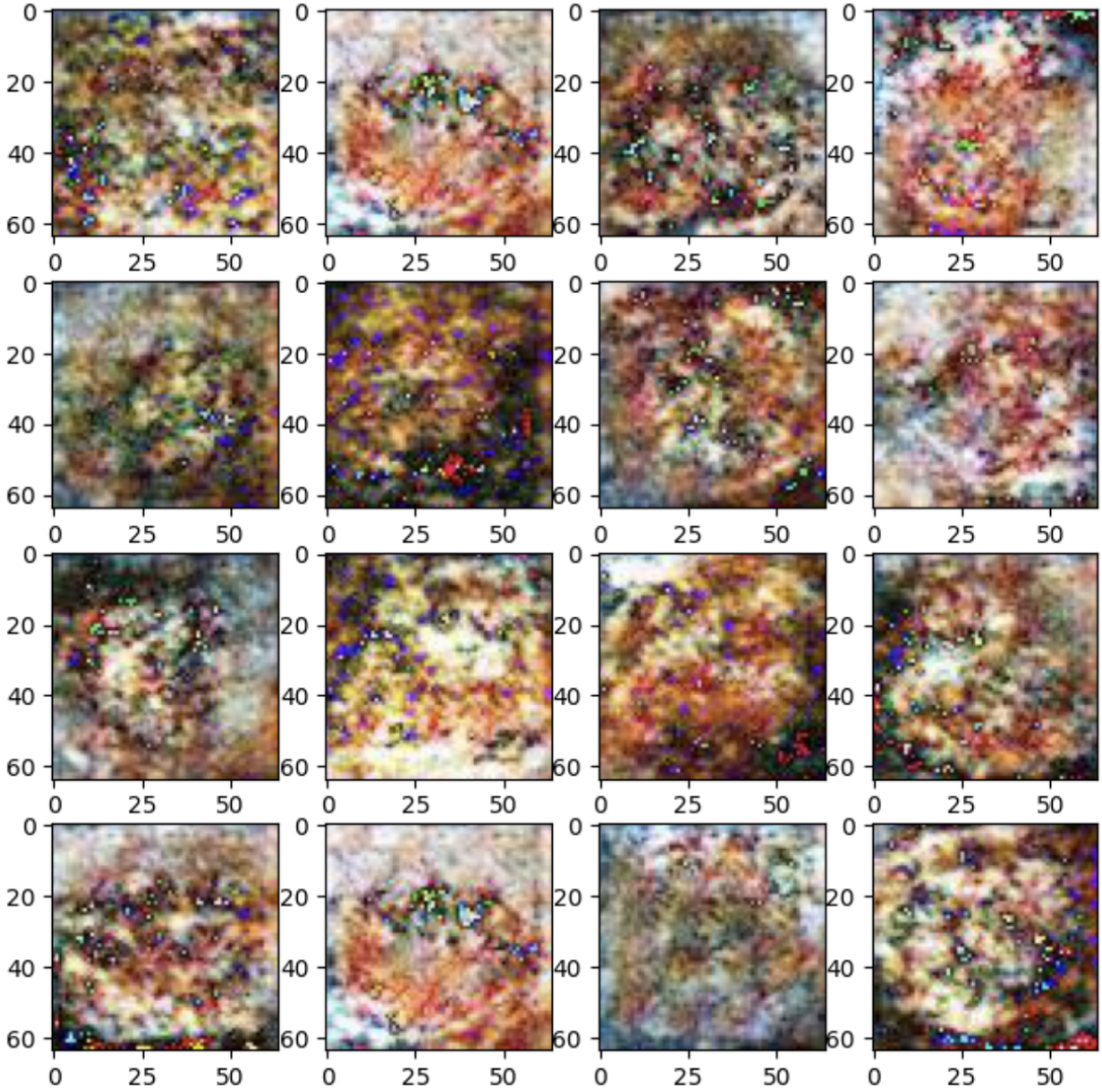
W-GAN:

April 7, 2023



Results:

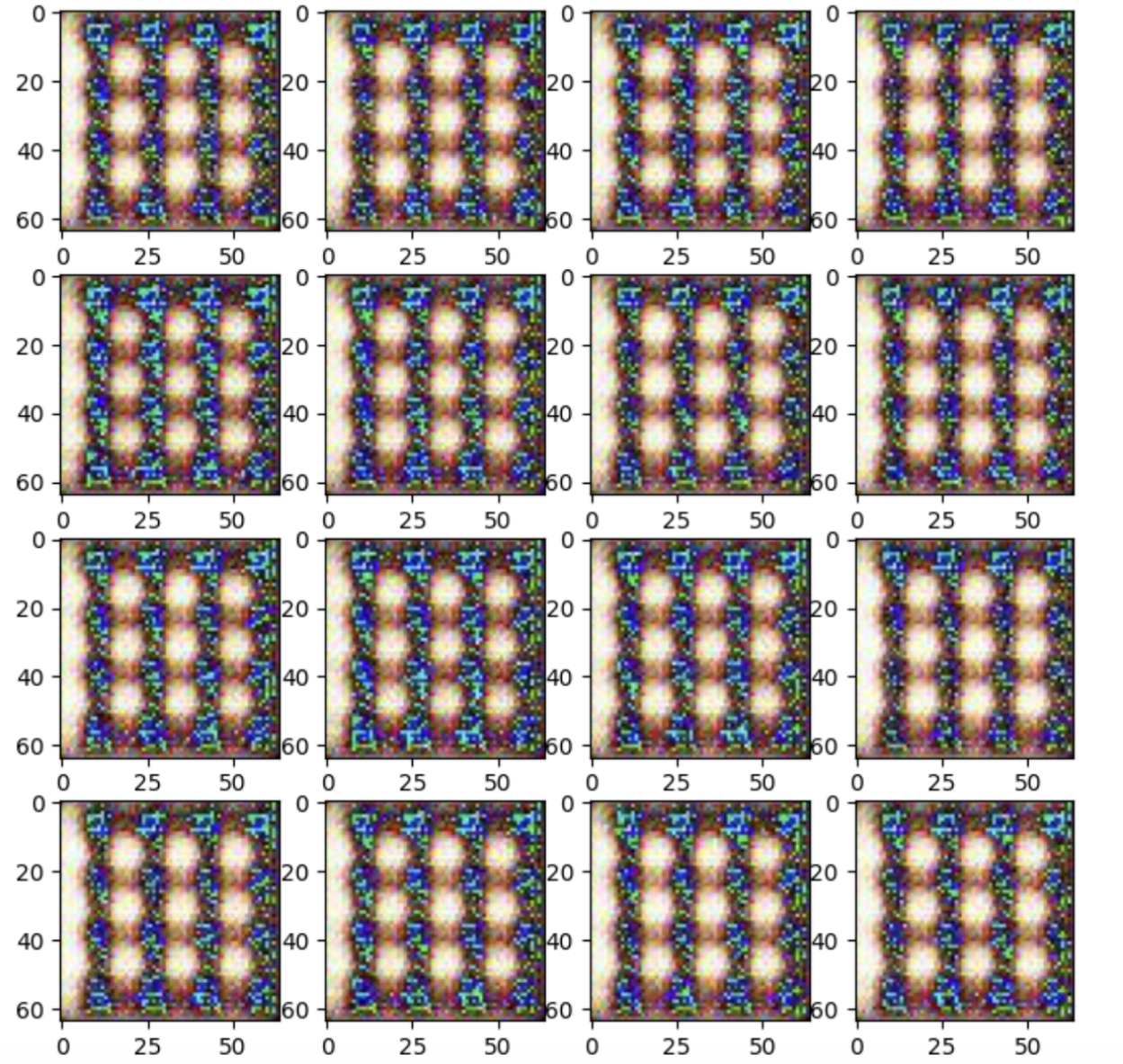
16 random images from the BCE-GAN Trained Generator:



FID for BCE-GAN Generator Images against 1000 Evaluation Images:

```
100%|                    | 20/20
231.69390714090238
```

16 random images from the W-GAN Trained Generator:



FID for W-GAN Generator Images against 1000 Evaluation Images:

100%|[redacted]| 207

397.7164873260314

Results Discussion:

It is pretty apparent that the BCE-GAN is presenting better results. First of all, the generated images look almost like pizzas, but taken with a camera that has some distortion and noise. However, it is pretty apparent that the image quality is not as good as the original training set. Potentially, adding the skip connections to the BCE-GAN would improve the results. For W-GAN, it is apparent that the results suffer from mode collapse. However, it is interesting that the network starts to produce pizza-like colors (the yellowish/red circles) in the middle of the

image, while there is random noise in the background. Potentially, experimenting more with the gradient clipping penalty or lambda values can help improve the results for the W-GAN.

Code:

Dataloader

```
### Create Dataloader
root = './'
folders = ['pizzas/train/0', 'pizzas/eval']
class MyDataset(torch.utils.data.Dataset):
    def __init__(self, root, folder):
        super(MyDataset).__init__()
        self.path = root + folder
        self.imgs = os.listdir(self.path)

        for img in self.imgs:
            if (img == "DS_Store"):
                self.imgs.remove(".DS_Store") #handle case when image isn't an image. Just remove it from the
                #image list.
                #print('removed DS store')
        self.to_Tensor_and_Norm = tvt.Compose([tvt.ToTensor(), tvt.Normalize([0], [1])])
        #print(len(self.imgs))

    def __len__(self):
        #return 102
        return len(self.imgs)

    def __getitem__(self, index):
        PIL_img = Image.open(self.path + '/' + self.imgs[index])
        torch_img = self.to_Tensor_and_Norm(PIL_img)
        return  torch_img

train_dataset = MyDataset(root, folders[0])
val_dataset = MyDataset(root, folders[1])

index = 2
print(train_dataset[index].shape)


train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=64, num_workers = 0, drop_last=False)
val_dataloader = torch.utils.data.DataLoader(val_dataset, batch_size = 64, num_workers = 0, drop_last = False)

```

Generator and Discriminator Network for BCE-GAN. Note that the code was slightly modified from Professor Kak's DLStudio code, but mostly the same.

```python
class DiscriminatorDG1(nn.Module):
    """
    This is an implementation of the DCGAN Discriminator. I refer to the DCGAN network topology as
    the 4-2-1 network.  Each layer of the Discriminator network carries out a strided
    convolution with a 4x4 kernel, a 2x2 stride and a 1x1 padding for all but the final
    layer. The output of the final convolutional layer is pushed through a sigmoid to yield
    a scalar value as the final output for each image in a batch.

    Class Path:  AdversarialLearning  ->  DataModeling  ->  DiscriminatorDG1
    """
    def __init__(self):
        super(DiscriminatorDG1, self).__init__()
        self.conv_in = nn.Conv2d(  3,    64,      kernel_size=4,      stride=2,    padding=1)
        self.conv_in2 = nn.Conv2d( 64,   128,     kernel_size=4,      stride=2,    padding=1)
        self.conv_in3 = nn.Conv2d( 128,  256,     kernel_size=4,      stride=2,    padding=1)
        self.conv_in4 = nn.Conv2d( 256,  512,     kernel_size=4,      stride=2,    padding=1)
        self.conv_in5 = nn.Conv2d( 512,  1,       kernel_size=4,      stride=1,    padding=0)
        self.bn1  = nn.BatchNorm2d(128)
        self.bn2  = nn.BatchNorm2d(256)
        self.bn3  = nn.BatchNorm2d(512)
        self.sig = nn.Sigmoid()
    def forward(self, x):
        x = torch.nn.functional.leaky_relu(self.conv_in(x), negative_slope=0.2, inplace=True)
        x = self.bn1(self.conv_in2(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.bn2(self.conv_in3(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.bn3(self.conv_in4(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.conv_in5(x)
        x = self.sig(x)
        return x

class GeneratorDG1(nn.Module):
    """
    This is an implementation of the DCGAN Generator. As was the case with the Discriminator network,
    you again see the 4-2-1 topology here.  A Generator's job is to transform a random noise
    vector into an image that is supposed to look like it came from the training dataset. (We refer
    to the images constructed from noise vectors in this manner as fakes.)  As you will see later
    in the "run_gan_code()" method, the starting noise vector is a 1x1 image with 100 channels.  In
    order to output 64x64 output images, the network shown below use the Transpose Convolution
    operator nn.ConvTranspose2d with a stride of 2.  If (H_in, W_in) are the height and the width
    of the image at the input to a nn.ConvTranspose2d layer and (H_out, W_out) the same at the
    output, the size pairs are related by
                H_out   =   (H_in - 1) * s   +   k   -   2 * p
                W_out   =   (W_in - 1) * s   +   k   -   2 * p

    were s is the stride and k the size of the kernel.  (I am assuming square strides, kernels, and
    padding). Therefore, each nn.ConvTranspose2d layer shown below doubles the size of the input.

    Class Path:  AdversarialLearning  ->  DataModeling  ->  GeneratorDG1
    """
    def __init__(self):
        super(GeneratorDG1, self).__init__()
        self.latent_to_image = nn.ConvTranspose2d( 100,   512,  kernel_size=4, stride=1, padding=0, bias=False)
        self.upsampler2 = nn.ConvTranspose2d( 512, 256, kernel_size=4, stride=2, padding=1, bias=False)
        self.upsampler3 = nn.ConvTranspose2d (256, 128, kernel_size=4, stride=2, padding=1, bias=False)
        self.upsampler4 = nn.ConvTranspose2d (128, 64,  kernel_size=4, stride=2, padding=1, bias=False)
        self.upsampler5 = nn.ConvTranspose2d(  64,  3,  kernel_size=4, stride=2, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(512)
        self.bn2 = nn.BatchNorm2d(256)
        self.bn3 = nn.BatchNorm2d(128)
        self.bn4 = nn.BatchNorm2d(64)
        self.tanh  = nn.Tanh()
    def forward(self, x):
        x = self.latent_to_image(x)
        x = torch.nn.functional.relu(self.bn1(x))
        x = self.upsampler2(x)
        x = torch.nn.functional.relu(self.bn2(x))
        x = self.upsampler3(x)
        x = torch.nn.functional.relu(self.bn3(x))
        x = self.upsampler4(x)
        x = torch.nn.functional.relu(self.bn4(x))
        x = self.upsampler5(x)
        x = self.tanh(x)
        return x
```

Critic and Generator for the W-GAN. Again, mostly taken from the DLStudio implementation.

```
####################################   Critic-Generator CG2   ####################################
class CriticCG2(nn.Module):
    """
    For the sake of variety, the Critic implementation in CG2 as the same Marvin Cao's Discriminator:

            https://github.com/caogang/wgan-gp

    which in turn is the PyTorch version of the Tensorflow based Discriminator presented by the
    authors of the paper "Improved Training of Wasserstein GANs" by Gulrajani, Ahmed, Arjovsky, Dumouli,
    and Courville.

    Class Path:  AdversarialLearning  ->  DataModeling  ->  CriticCG2
    """
    def __init__(self):
        super(CriticCG2, self).__init__()
        self.DIM = 64
        main = nn.Sequential(
            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.main = main
        self.output = nn.Linear(4*4*4*self.DIM, 1)

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

class GeneratorCG2(nn.Module):
    """
    The Generator code remains the same as for DG1 shown earlier.

    Class Path:  AdversarialLearning  ->  DataModeling  ->  GeneratorCG2
    """
    def __init__(self):
        super(GeneratorCG2, self).__init__()
        self.latent_to_image = nn.ConvTranspose2d( 100,   512,  kernel_size=4, stride=1, padding=0, bias=False)
        self.upsampler2 = nn.ConvTranspose2d( 512, 256, kernel_size=4, stride=2, padding=1, bias=False)
        self.upsampler3 = nn.ConvTranspose2d (256, 128, kernel_size=4, stride=2, padding=1, bias=False)
        self.upsampler4 = nn.ConvTranspose2d (128, 64,  kernel_size=4, stride=2, padding=1, bias=False)
        self.upsampler5 = nn.ConvTranspose2d(  64,  3,  kernel_size=4, stride=2, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(512)
        self.bn2 = nn.BatchNorm2d(256)
        self.bn3 = nn.BatchNorm2d(128)
        self.bn4 = nn.BatchNorm2d(64)
        self.tanh  = nn.Tanh()
    def forward(self, x):
        x = self.latent_to_image(x)
        x = torch.nn.functional.relu(self.bn1(x))
        x = self.upsampler2(x)
        x = torch.nn.functional.relu(self.bn2(x))
        x = self.upsampler3(x)
        x = torch.nn.functional.relu(self.bn3(x))
        x = self.upsampler4(x)
        x = torch.nn.functional.relu(self.bn4(x))
        x = self.upsampler5(x)
        x = self.tanh(x)
        return x
####################################   CG
```

A “mini DL studio” to input epochs, batch sizes, gpu preferences, and some other basic parameters in the experiments:

```
class MiniDLStudio(nn.Module):
    def __init__(self, epochs, use_gpu, train_dataloader, val_dataloader, batch_size, lr, LAMBDA):
        super(MiniDLStudio, self).__init__()
        self.epochs = epochs
        self.val_dataloader = val_dataloader
        self.train_dataloader = train_dataloader
        self.batch_size = batch_size
        self.learning_rate = lr
        self.LAMBDA = LAMBDA
        if use_gpu is not None:
            self.use_gpu = use_gpu
            if use_gpu is True:
                if torch.cuda.is_available():
                    self.device = torch.device("cuda:0")
                else:
                    raise Exception("You requested GPU support, but there's no GPU on this machine")
            else:
                self.device = torch.device("cpu")
                print("assigned self.device")

    def weights_init(self,m):
        """
        Uses the DCGAN initializations for the weights
        """
        classname = m.__class__.__name__
        if classname.find('Conv') != -1:
            nn.init.normal_(m.weight.data, 0.0, 0.02)
        elif classname.find('BatchNorm') != -1:
            nn.init.normal_(m.weight.data, 1.0, 0.02)
            nn.init.constant_(m.bias.data, 0)

    def calc_gradient_penalty(self, netC, real_data, fake_data):
        """
        Implementation by Marvin Cao: https://github.com/caogang/wgan-gp
        Marvin Cao's code is a PyTorch version of the Tensorflow based implementation provided by
        the authors of the paper "Improved Training of Wasserstein GANs" by Gulrajani, Ahmed,
        Arjovsky, Dumouli,  and Courville.
        """
        BATCH_SIZE = self.batch_size
        LAMBDA = self.LAMBDA
        epsilon = torch.rand(1).cuda() #may need to change to .cuda() when using GPU
        interpolates = epsilon * real_data + ((1 - epsilon) * fake_data)
        interpolates = interpolates.requires_grad_(True).cuda() #may need to change to .cuda() on GPU
        critic_interpolates = netC(interpolates)
        gradients = torch.autograd.grad(outputs=critic_interpolates, inputs=interpolates,
                                        grad_outputs=torch.ones(critic_interpolates.size()).cuda(), #may need to change to .cuda() later!
                                        create_graph=True, retain_graph=True, only_inputs=True)[0]
        gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean() * LAMBDA
        return gradient_penalty
```

April 7, 2023

Training and Inference Loop for the W-GAN:

```
def run_wgan_with_gp_code(self, critic, generator, results_dir):
            """
            This function is meant for training a CG2-based Critic-Generator WGAN. Regarding how
            to set the parameters of this method, see the following script in the
            "ExamplesAdversarialLearning" directory of the distribution:

                        wgan_with_gp_CG2.py
            """
            dir_name_for_results = results_dir_wgan_gp
#             if os.path.exists(dir_name_for_results):
#                 files = glob.glob(dir_name_for_results + "/*")
#                 for file in files:
#                     if os.path.isfile(file):
#                         os.remove(file)
#                     else:
#                         files = glob.glob(file + "/*")
#                         list(map(lambda x: os.remove(x), files))
#             else:
#                 os.mkdir(dir_name_for_results)
            #  Set the number of channels for the 1x1 input noise vectors for the Generator:
            nz = 100
            netC = critic.to(self.device)
            netG = generator.to(self.device)
            #  Initialize the parameters of the Critic and the Generator networks according to the
            #  definition of the "weights_init()" method:
            netC.apply(self.weights_init)
            netG.apply(self.weights_init)
            #  We will use a the same noise batch to periodically check on the progress made for the Generator:
            fixed_noise = torch.randn(self.batch_size, nz, 1, 1, device=self.device)
            #  These are for training the Critic, 'one' is for the part of the training with actual
            #  training images, and 'minus_one' is for the part based on the images produced by the
            #  Generator:
            one = torch.FloatTensor([1]).to(self.device)
            minus_one = torch.FloatTensor([-1]).to(self.device)
            #  Adam optimizers for the Critic and the Generator:
            optimizerC = optim.Adam(netC.parameters(), lr=self.learning_rate, betas=(beta1, 0.999))
            optimizerG = optim.Adam(netG.parameters(), lr=self.learning_rate, betas=(beta1, 0.999))
            img_list = []
            Gen_losses = []
            Cri_losses = []
            iters = 0
            gen_iterations = 0
            start_time = time.perf_counter()
            dataloader = self.train_dataloader
            # For each epoch
            for epoch in range(self.epochs):
                data_iter = iter(dataloader)
                i = 0
                ncritic   = 5
                #  In this version of WGAN training, we enforce the 1-Lipschitz condition on the function
                #  being learned by the Critic by requiring that the partial derivatives of the output of
                #  the Critic with respect to its input equal one in magnitude. This is referred as imposing
                #  a Gradient Penalty on the learning by the Critic.  As in the previous training
                #  function, we start by turning on the "requires_grad" property of the Critic parameters:
                while i < len(dataloader):
                    for p in netC.parameters():
                        p.requires_grad = True
                    ic = 0
                    while ic < ncritic and i < len(dataloader):
                        ic += 1
                        #  The first two parts of what it takes to train the Critic are the same as for
                        #  a regular WGAN.  We want to train the Critic to recognize the training images and,
                        #  at the same time, the Critic should try to not believe the output of the Generator.
                        netC.zero_grad()
                        real_images_in_batch =  next(data_iter) # was data_iter.next() before
                        i += 1
                        real_images_in_batch =  real_images_in_batch[0].to(self.device)
                        #  Need to know how many images we pulled in since at the tailend of the dataset, the
                        #  number of images may not equal the user-specified batch size:
                        b_size = real_images_in_batch.size(0)
                        #  Note that a single scalar is produced for all the data in a batch.
                        critic_for_reals_mean = netC(real_images_in_batch)     ## this is a batch based mean
                        #  The gradient target is 'minus_one'.  Note that the gradient here is one of output of
                        #  the network with respect to the learnable parameters:
                        critic_for_reals_mean.backward(minus_one)
                        #  The second part of Critic training requires that we apply the Critic to the images
```

```
                    critic_for_reals_mean.backward(minus_one)
                    #  The second part of Critic training requires that we apply the Critic to the images
                    #  produced by the Generator from a fresh batch of input noise vectors.
                    noise = torch.randn(b_size, nz, 1, 1, device=self.device)
                    fakes = netG(noise)
                    #  Again, a single number is produced for the whole batch:
                    critic_for_fakes_mean = netC(fakes.detach())  ## detach() returns a copy of the 'fakes' tensor that has
                                                                  ## been removed from the computational graph. This ensures
                                                                  ## that a subsequent call to backward() will only update the Critic
                    #  The gradient target is 'one'.  Note that the gradient here is one of output of
                    #  the network with respect to the learnable parameters:
                    critic_for_fakes_mean.backward(one)
                    #  For the third part of Critic training, we need to first estimate the Gradient Penalty
                    #  of the function being learned by the Critics with respect to the input to the function.
                    gradient_penalty = self.calc_gradient_penalty(netC, real_images_in_batch, fakes)
                    gradient_penalty.backward()
                    loss_critic = critic_for_fakes_mean - critic_for_reals_mean + gradient_penalty
                    wasser_dist = critic_for_reals_mean - critic_for_fakes_mean
                    #  Update the Critic
                    optimizerC.step()

                #  That brings us to the training of the Generator.  First we must turn off the "requires_grad"
                #  of the Critic parameters since the Critic and the Generator are to be updated independently:
                for p in netC.parameters():
                    p.requires_grad = False
                netG.zero_grad()
                #  This is again a single scalar based characterization of the whole batch of the Generator images:
                noise = torch.randn(b_size, nz, 1, 1, device=self.device)
                fakes = netG(noise)
                critic_for_fakes_mean = netC(fakes)
                loss_gen = critic_for_fakes_mean
                #  The gradient target is 'minus_one'.  Note that the gradient here is one of output of the network
                #  with respect to the learnable parameters:
                loss_gen.backward(minus_one)
                #  Update the Generator
                optimizerG.step()
                gen_iterations += 1
                if i % (ncritic * 20) == 0:
                    current_time = time.perf_counter()
                    elapsed_time = current_time - start_time
                    print("[epoch=%d/%d   i=%4d   el_time=%5d secs]     loss_critic=%7.4f   loss_gen=%7.4f   Wasserstein_dist=%7.4f" %  (epoch+1,self
                Gen_losses.append(loss_gen.data[0].item())
                Cri_losses.append(loss_critic.data[0].item())
                #  Get G's output on fixed_noise for the GIF animation:
                if (iters % 500 == 0) or ((epoch == self.epochs-1) and (i == len(self.train_dataloader)-1)):
                    with torch.no_grad():
                        fake = netG(fixed_noise).detach().cpu()  ## detach() removes the fake from comp. graph
                                                                 ## in order to produce a CPU compatible tensor
                    img_list.append(torchvision.utils.make_grid(fake, padding=1, pad_value=1, normalize=True))
                iters += 1

        #RUN INFERENCE ON VAL DATA

        print("about to run inference!!!")
        with torch.no_grad():
          count = 0
          for n, data in enumerate(self.val_dataloader):
#            if n< 1:
             noise = torch.randn(self.batch_size, nz, 1, 1, device=self.device)
             fakes_inference = netG(noise)
             print(fakes_inference.shape)
             print("n is", n)

             for num, img in enumerate(fakes_inference):
#               if num < 5:
                #convert tensor to jpg, save to wgan_inference_images_fid, store in a list of "fake_paths"
                transform = tvt.ToPILImage()
                temp_jpg = transform(img)
                #print(type(temp_jpg))
                temp_num = num + (count)*self.batch_size
                filename = './wgan_inference_images_fid/' + str(temp_num) + '.jpg'
                temp_jpg.save(fp = filename)
                #print('saved to ', filename)
                fake_paths.append(filename)
                if num == self.batch_size-1:
                  count = count+1
```

Training and Inference Loop for BCE-GAN:

```
fake_paths_dcgan = []
def run_gan_code(self, discriminator, generator, results_dir):
    """
    This function is meant for training a Discriminator-Generator based Adversarial Network.
    The implementation shown uses several programming constructs from the "official" DCGAN
    implementations at the PyTorch website and at GitHub.

    Regarding how to set the parameters of this method, see the following script

                dcgan_DG1.py

    in the "ExamplesAdversarialLearning" directory of the distribution.
    """
    dir_name_for_results = results_dir
#     if os.path.exists(dir_name_for_results):
#         files = glob.glob(dir_name_for_results + "/*")
#         for file in files:
#             if os.path.isfile(file):
#                 os.remove(file)
#             else:
#                 files = glob.glob(file + "/*")
#                 list(map(lambda x: os.remove(x), files))
#     else:
#         os.mkdir(dir_name_for_results)
    #  Set the number of channels for the 1x1 input noise vectors for the Generator:
    nz = 100
    netD = discriminator.to(self.device)
    netG = generator.to(self.device)
    #  Initialize the parameters of the Discriminator and the Generator networks according to the
    #  definition of the "weights_init()" method:
    netD.apply(self.weights_init)
    netG.apply(self.weights_init)
    #  We will use a the same noise batch to periodically check on the progress made for the Generator:
    fixed_noise = torch.randn(self.batch_size, nz, 1, 1, device=self.device)
    #  Establish convention for real and fake labels during training
    real_label = 1
    fake_label = 0
    #  Adam optimizers for the Discriminator and the Generator:
    optimizerD = optim.Adam(netD.parameters(), lr=self.learning_rate, betas=(adversarial_beta1, 0.999))
    optimizerG = optim.Adam(netG.parameters(), lr=self.learning_rate, betas=(adversarial_beta1, 0.999))
    #  Establish the criterion for measuring the loss at the output of the Discriminator network:
    criterion = nn.BCELoss()
    #  We will use these lists to store the results accumulated during training:
    img_list = []
    G_losses = []
    D_losses = []
    iters = 0
    print("\n\nStarting Training Loop...\n\n")
    start_time = time.perf_counter()
    for epoch in range(self.epochs):
        g_losses_per_print_cycle = []
        d_losses_per_print_cycle = []
        # For each batch in the dataloader
        for i, data in enumerate(self.train_dataloader, 0):
            #print("starting new batch")
            ##  Maximization Part of the Min-Max Objective of Eq. (3):
            ##
            ##  As indicated by Eq. (3) in the DCGAN part of the doc section at the beginning of this
            ##  file, the GAN training boils down to carrying out a min-max optimization. Each iterative
            ##  step of the max part results in updating the Discriminator parameters and each iterative
            ##  step of the min part results in the updating of the Generator parameters.  For each
            ##  batch of the training data, we first do max and then do min.  Since the max operation
            ##  affects both terms of the criterion shown in the doc section, it has two parts: In the
            ##  first part we apply the Discriminator to the training images using 1.0 as the target;
            ##  and, in the second part, we supply to the Discriminator the output of the Generator
            ##  and use 0 as the target. In what follows, the Discriminator is being applied to
            ##  the training images:
            netD.zero_grad()
            #real_images_in_batch = data[0].to(self.device)
            real_images_in_batch = data.to(self.device)
            #print("real_images_in_batch is:", real_images_in_batch.shape)
            #print("data shape is", data.shape)
            #  Need to know how many images we pulled in since at the tailend of the dataset, the
            #  number of images may not equal the user-specified batch size:
            b_size = real_images_in_batch.size(0)
            #print("b_size is", b_size)
            label = torch.full((b_size,), real_label, dtype=torch.float, device=self.device)
```

```
            ##  That brings us the second part of what it takes to carry out the max operation on the
            ##  min-max criterion shown in Eq. (3) in the doc section at the beginning of this file.
            ##  part calls for applying the Discriminator to the images produced by the Generator from
            ##  noise:
            noise = torch.randn(b_size, nz, 1, 1, device=self.device)
            fakes = netG(noise)
            label.fill_(fake_label)
            ##  The call to fakes.detach() in the next statement returns a copy of the 'fakes' tensor
            ##  that does not exist in the computational graph. That is, the call shown below first
            ##  makes a copy of the 'fakes' tensor and then removes it from the computational graph.
            ##  The original 'fakes' tensor continues to remain in the computational graph.  This ploy
            ##  ensures that a subsequent call to backward() in the 3rd statement below would only
            ##  update the netD weights.
            output = netD(fakes.detach()).view(-1)
            lossD_for_fakes = criterion(output, label)
            ##  At this point, we do not care if the following call also calculates the gradients
            ##  wrt the Discriminator weights since we are going to next iteration with 'netD.zero_grad()':
            lossD_for_fakes.backward()
            lossD = lossD_for_reals + lossD_for_fakes
            d_losses_per_print_cycle.append(lossD)
            ##  Only the Discriminator weights are incremented:
            optimizerD.step()

            ##  Minimization Part of the Min-Max Objective of Eq. (3):
            ##
            ##  That brings to the min part of the max-min optimization described in Eq. (3) the doc
            ##  section at the beginning of this file.  The min part requires that we minimize
            ##  "1 - D(G(z))" which, since D is constrained to lie in the interval (0,1), requires that
            ##  we maximize D(G(z)).  We accomplish that by applying the Discriminator to the output
            ##  of the Generator and use 1 as the target for each image:
            netG.zero_grad()
            label.fill_(real_label)
            output = netD(fakes).view(-1)
            lossG = criterion(output, label)
            g_losses_per_print_cycle.append(lossG)
            lossG.backward()
            ##  Only the Generator parameters are incremented:
            optimizerG.step()

            if i % 100 == 99:
                current_time = time.perf_counter()
                elapsed_time = current_time - start_time
                mean_D_loss = torch.mean(torch.FloatTensor(d_losses_per_print_cycle))
                mean_G_loss = torch.mean(torch.FloatTensor(g_losses_per_print_cycle))
                print("[epoch=%d/%d   iter=%4d   elapsed_time=%5d secs]     mean_D_loss=%7.4f    mean_G_loss=%7.4f" %
                              ((epoch+1),self.epochs,(i+1),elapsed_time,mean_D_loss,mean_G_loss))
                d_losses_per_print_cycle = []
                g_losses_per_print_cycle = []
            G_losses.append(lossG.item())
            D_losses.append(lossD.item())
            if (iters % 500 == 0) or ((epoch == self.epochs-1) and (i == len(self.train_dataloader)-1)):
                with torch.no_grad():
                    fake = netG(fixed_noise).detach().cpu()  ## detach() removes the fake from comp. graph.
                                                             ## for creating its CPU compatible version
                img_list.append(torchvision.utils.make_grid(fake, padding=1, pad_value=1, normalize=True))
            iters += 1
```

```
    #run inference using trained generator:
    with torch.no_grad():
      count = 0
      for n, data in enumerate(val_dataloader):
        #if n< 1:
          data_shp = data.shape[0]
          print(data_shp)
          noise = torch.randn(data_shp, nz, 1, 1, device=self.device)
          fakes_inference = netG(noise)
          print("fakes_inference shape", fakes_inference.shape)
          #print("n is", n)

          for num, img in enumerate(fakes_inference):
             #if num < 5:
                #convert tensor to jpg, save to wgan_inference_images_fid, store in a list of "fake_paths"
                transform = tvt.ToPILImage()
                temp_jpg = transform(img)
                #print(type(temp_jpg))
                temp_num = num + (count)*self.batch_size
                filename = './dcgan_inference_images_fid/' + str(temp_num) + '.jpg'
                temp_jpg.save(fp = filename)
                #print('saved to ', filename)
                fake_paths_dcgan.append(filename)
                if num == self.batch_size-1:
                  count = count+1

      print("fake paths is", fake_paths_dcgan, "and has \n", len(fake_paths_dcgan), "images")
```

Code to kick off training and inference for both networks:

```
#### CODE TO KICK OFF DCGAN TRAINING/EVAL ###

mdls = MiniDLStudio(epochs = 100, use_gpu = True, train_dataloader = train_dataloader, val_dataloader = val_dataloader, batch_size=64,
                    lr = 1e-5, LAMBDA = LAMBDA)
run_gan_code(mdls, discriminator, generator, results_dir)
```

```

#run wgan code:
critic = CriticCG2()
generator = GeneratorCG2()
print(os.getcwd())
learning_rate = 0.00001
#wgan.show_sample_images_from_dataset(dls)
mdls_wgan = MiniDLStudio(epochs = 100, use_gpu = True, train_dataloader = train_dataloader, val_dataloader = val_dataloader, batch_size=40,
                         lr = learning_rate, LAMBDA = LAMBDA)
run_wgan_with_gp_code(mdls_wgan, critic=critic, generator=generator,
                          results_dir=results_dir_wgan_gp)


```

Code to Compute FID scores (note that for both networks they are very similar, so, only one is pasted):

```
### compute FID for DC GAN ###
#first, test opening fake paths:
PIL_img = Image.open('./dcgan_inference_images_fid/0.jpg')
display(PIL_img)

#put real validation images in list:
real_paths_dcgan = []
d = os.listdir('./pizzas/eval')
print(len(d))
for n, i in enumerate(d):
  real_paths_dcgan.append('./pizzas/eval/' + str(i))

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

dims = 2048
block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
model = InceptionV3([block_idx]).to('cuda:0')
m1, s1 = calculate_activation_statistics(real_paths_dcgan, model, device='cuda:0')
m2, s2 = calculate_activation_statistics(fake_paths_dcgan, model, device='cuda:0')
fid_value = calculate_frechet_distance(m1, s1, m2, s2)
print(fid_value)
```

Code to plot generated images:

```
#generate 4 x 4 plot of generated images:
fig, ax = plt.subplots(4, 4, figsize = (8,8))
for i in range(4):
  for j in range(4):
    temp_num = np.random.randint(low=0, high=999, size=None, dtype=int)
    PIL_img = Image.open('./dcgan_inference_images_fid/' + str(temp_num) + '.jpg')
    ax[i,j].imshow(np.uint8(PIL_img))

```