GAN-Art: Generating Artwork using Deep Convolutional Generative Adversarial Networks (DCGANs)

Saksham Jindal
UC San Diego
sjindal@ucsd.edu

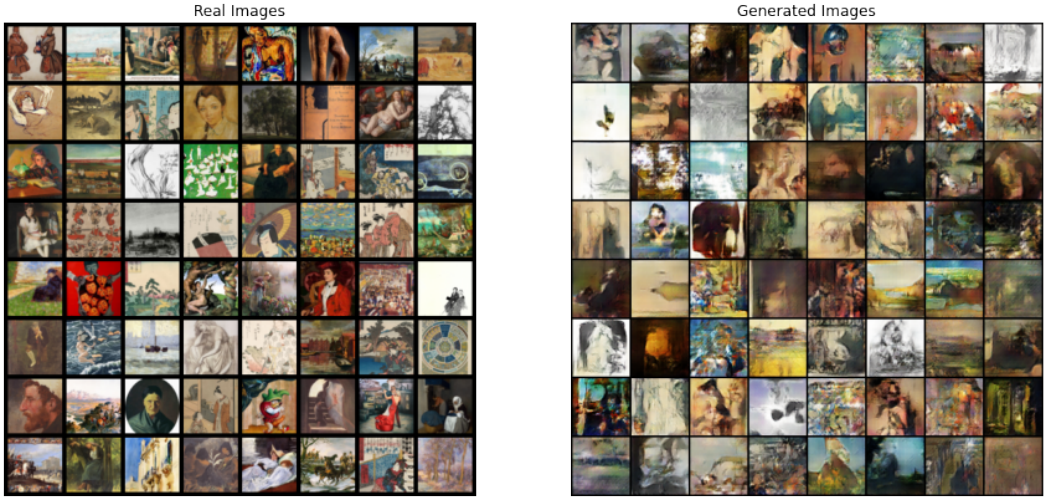


Figure 1: A comparison of real and generated images from DCGAN

1 Introduction

Generative Adversarial Networks (GANs) are a class of deep learning models that can generate realistic images by learning to mimic the training data distribution. GANs have achieved impressive results on generating photorealistic images of objects, scenes, and faces. In this paper, we propose training Deep Convolutional Generative Adversarial Networks (DCGAN) or DCGAN [3], a type of GAN that uses convolutional neural networks, on the ArtBench dataset to generate artistic images. The ArtBench dataset [2] comprises 60,000 images of artwork from 10 different artistic styles, with 5,000 training images and 1,000 testing images per style. We evaluate the DCGAN quantitatively using the Inception Score (IS) and Fréchet Inception Distance (FID) on generated images, and qualitatively by visualizing generated samples. By training a DCGAN on this dataset, we aim to capture the essence of each artistic style and generate new artwork that exhibits similar characteristics.

2 Method

2.1 Deep Convolutional Generative Adversarial Network (DCGAN)

DCGANs are a type of generative adversarial network (GAN) that use convolutional neural networks (CNNs) to generate realistic images. DCGANs consist of two main components: a generator and

a discriminator. The generator is responsible for creating new images, while the discriminator is responsible for distinguishing between real and fake images.

Generator: The generator G takes a random noise vector z, drawn from standard normal distribution, as input and generates synthetic image $G(z)$. It is a deep convolutional neural network, comprising of convolutional-transpose layers, batch norm layers, and ReLU activations, that learns to map the input noise to the data space of the target images.

Discriminator: The discriminator D is a binary classifier that distinguishes between real artwork images and generated images. It is also a deep convolutional neural network that learns to classify images as real or fake. It is made up of strided convolution layers, batch norm layers, and LeakyReLU activations. The input is an image and the output is a probability score of the input being from the real data distribution. The discriminator aims to accurately classify the generated images as fake while correctly classifying real artwork images.

Loss Function: The two networks are trained simultaneously, with the generator trying to create images that are indistinguishable from real images, and the discriminator trying to correctly identify real and fake images. In each training iteration, a batch of real artwork images and a batch of generated images were fed into the discriminator. The discriminator was trained to correctly classify real and fake images using binary cross-entropy loss. Subsequently, the generator was trained to minimize the discriminator's ability to differentiate between real and generated image by trying to generate realistic images.

The discriminator loss is given by

$$\mathcal{L}_D = \mathbb{E}_{x \sim p_{data}(x)}[-\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[-\log(1 - D(G(z)))] \tag{1}$$

The generator loss is given by

$$\mathcal{L}_G = \mathbb{E}_{z \sim p_z(z)}[-\log(D(G(z)))] \tag{2}$$

where p_{data} is the data distribution and p_z is the noise distribution.

2.2 Dataset

The ArtBench dataset[2] contains 60,000 images of artwork in 10 different styles: Baroque, Cubism, Expressionism, Fauvism, Impressionism, Naive Art, Pop Art, Post-Impressionism, Realism, and Surrealism. It is a class-balanced dataset with 5,000 training images and 1,000 testing images for each style. We use the training set to train our DCGAN and evaluate on the testing set.

2.3 Experiments

Data Preprocessing: The ArtBench-10 dataset was preprocessed to resize the images to a consistent resolution of of (64, 64) and normalizing the pixel values using mean and standard deviation.

Network Initialization: The generator and discriminator networks were initialized with random weights from a normal distribution with mean 0 and standard deviation 0.02.

Network Archtecture: We use series of convolution layers for filter extraction (without batch normalization) in the encoder and transpose convolution for upsampling in the decoder. More details on the architecture can be found below.

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(150, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1,
        1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (5): ReLU(inplace=True)
```

```
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1,
        1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1,
        1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
        bias=False)
    (13): Tanh()
  )
)
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
        bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
        bias=False)
    (3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
        bias=False)
    (6): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
        bias=False)
    (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(256, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

Training: We used a batch size of 80 with the Adam optimizer with a learning rate of 0.0001 and trained the model for 50 epochs. At higher learning rates of the scale of 0.01, we observed that the GAN's training process is highly unstable, and the generator's and discriminator's performance oscillates.

Metrics: To evaluate the quality and diversity of generated images in our model, we employed two popular metrics: Inception Score (IS) and Fréchet Inception Distance (FID). These metrics provide quantitative measures that help assess the performance of generative models. Fréchet Inception Distance (FID) is a metric that is used to assess the similarity between the distribution of generated images and the distribution of real images. Inception Score (IS) is a metric that is used to assess the quality and diversity of generated images from a generative model. Usually, a higher IS typically indicates better quality and diversity in generated samples and a lower FID score indicates a better match between the distribution of generated and real images.

3 Results

3.1 Training Loss Curves

Figure 2 shows the training loss curves of the DCGAN during the training process. The x-axis represents the number of training iterations, while the y-axis represents the corresponding loss values. The 2 losses: generator loss and discriminator loss are plotted separately. We observe that that both

the generator and discriminator losses gradually decrease over time, indicating the improvement in DCGAN's performance over the training period.

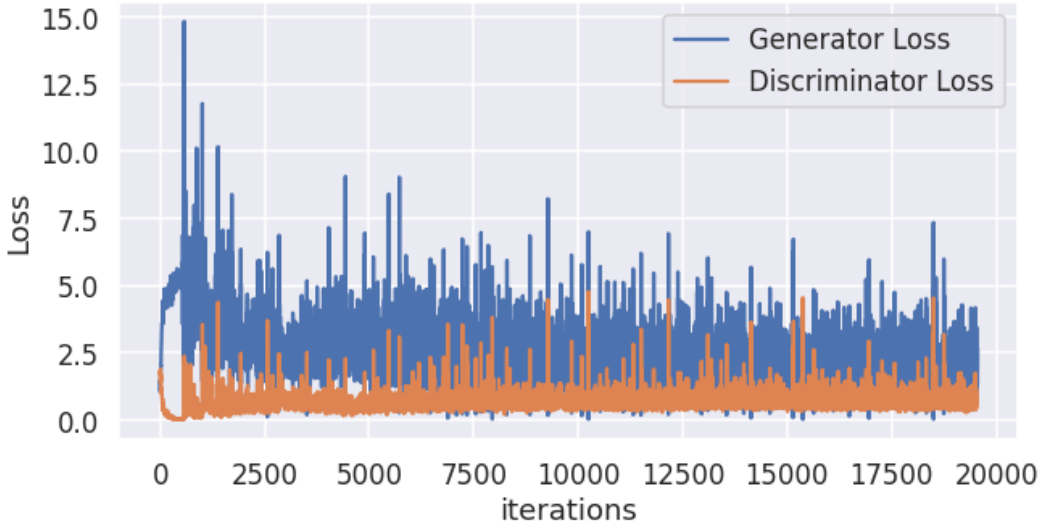


Figure 2: Training loss curves of the DCGAN

3.2 Quantitative Evaluation

We evaluate the DCGAN using the Inception Score (IS) and Fréchet Inception Distance (FID) on 10,0000 generated samples each and 10,000 real image from the test dataset, we obtained an Inception Score of 2.0025 and FID score of 0.0164 which indicates the generated samples are diverse and realistic. The FID score of 0.0164 on the test set indicates a reasonably close similarity between the distribution of the generated images and the distribution of real images. The Inception Score of 2.0025 suggests that , though the generated samples have a moderate level of quality and diversity, the model is still able to generate images that are visually similar to real images and have diverse patterns.

3.3 Qualitative Evaluation

Figure 3 showcases a selection of artwork generated by the trained DCGAN. The images demonstrate the DCGAN's ability to capture the distinctive artistic styles present in the ArtBench-10 dataset. We observe that the DCGAN is able to generate a good diversity of images and generation quality is reasonably closer to the real images.

4 Acknowledgment

We would like to acknowledge that our implementation of DCGAN for training, hyperparameter tuning, and generating artwork images was heavily inspired by the tutorial provided by PyTorch on their website titled "DCGAN Tutorial" (available here). We appreciate the efforts of the authors of the tutorial.

5 Discussion

In this paper, we trained a Deep Convolutional Generative Adversarial Network (DCGAN) on the ArtBench dataset to generate artistic images. We evaluated the DCGAN quantitatively using the Inception Score and Fréchet Inception Distance, and qualitatively by visualizing generated samples. The results show that the DCGAN can generate diverse, high-quality artistic images that capture the styles in the ArtBench dataset. The samples appear realistic and exhibit characteristics of different artistic forms. As a part of future work, we could be using a larger discriminator architecture may

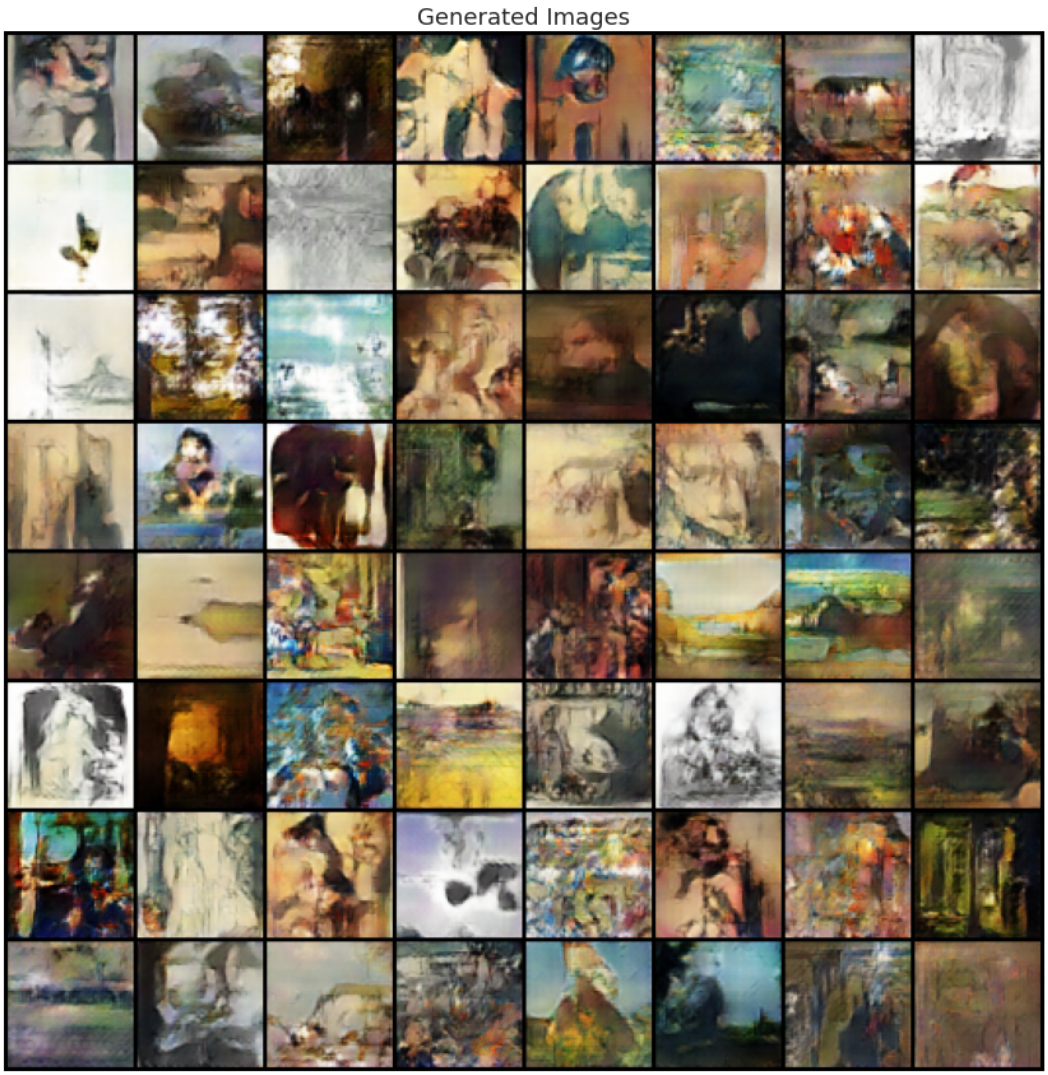


Figure 3: Generated samples from the trained DCGAN

better model the data distribution and improve the FID score. One could also use a pretrained discriminator like ResNet provides a stronger discriminator for higher quality generations. One could also use a different loss function, such as suggested in WGAN [1] which solves the problem of mode collapse and instability in vanilla GANs, or use conditional generation where class labels are provided as input to generate specific artistic styles.

In summary, we have demonstrated that DCGANs are capable of generating artistic images by learning from a dataset of artwork. The DCGAN could be enhanced by a more powerful discriminator, regularization techniques, different loss functions, pretrained networks or longer training.

References

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

[2] Peiyuan Liao, Xiuyu Li, Xihui Liu, and Kurt Keutzer. The artbench dataset: Benchmarking generative models with artworks. *arXiv preprint arXiv:2206.11404*, 2022.

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

In [1]:
```
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
```

In [2]:
```
dataroot = "../Datasets/wikiart_resized/wikiart/"
workers = 8
batch_size = 80
n_class = 10
image_size = 64
nc = 3
nz = 150
ngf = 64
ndf = 32
num_epochs = 50
lr = 0.0001
beta1 = 0.5
ngpu = 1
```

In [4]:
```
import torch
from torch.utils.data import DataLoader
import torchvision
import torchvision.transforms as transforms
import numpy as np
from PIL import Image
```



```
def get_dataset(x):
```

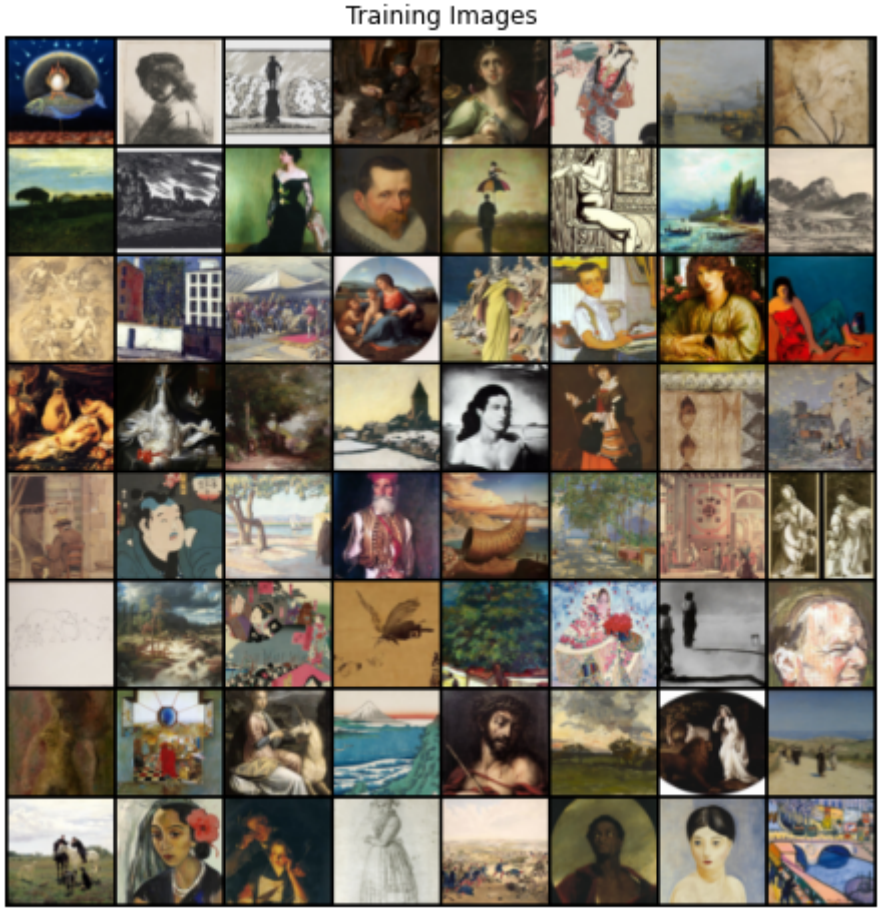
```
    image_size = 64
    transform = transforms.Compose([transforms.Resize((image_size, image_size)),
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

    dataset = torchvision.datasets.ImageFolder(root="./data/artbench-10-imagefolder-split/{}/".format(x),transform=

    return DataLoader(dataset, batch_size, shuffle=True, num_workers=workers, pin_memory=True)

# Create the dataloader
dataloader = get_dataset("train")

# Create the dataloader
valloader = get_dataset("test")
```

In [5]:
```
# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)


# Decide which device we want to run on
device = torch.device("cuda:1" if (torch.cuda.is_available() and ngpu > 0) else "cpu")
print(device)
```

```
Random Seed:  999
cuda:1
```

In [6]:
```
# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64], padding=2, normalize=True).cpu(),(1,2,0)))
```

Out[6]:
```
<matplotlib.image.AxesImage at 0x7f969c62d710>
```



In [7]:
```
# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

```
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
        )

    def forward(self, input):
        return self.main(input)


# Discriminator Model
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
```

```
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)
```

In [8]:
```
# Create the generator
netG = Generator(ngpu).to(device)
netG.apply(weights_init)
print(netG)

# Create the Discriminator
netD = Discriminator(ngpu).to(device)
netD.apply(weights_init)
print(netD)
```

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(150, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(256, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

In [9]:
```
# Initialize BCELoss function
criterion = nn.BCELoss()

#  at we will use to visualize
#  the progression of the generator
```

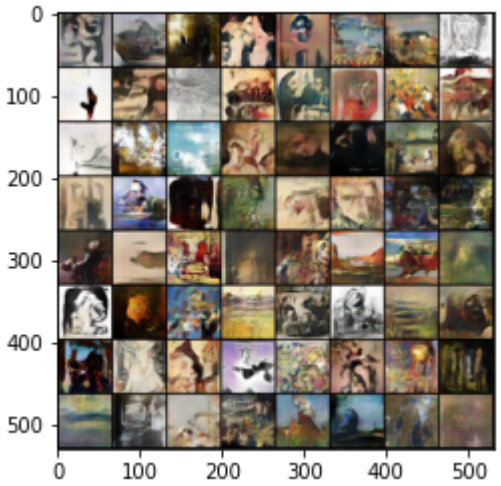
```
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1.
fake_label = 0.

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
```

In [51]:
```
real_cpu.shape
```

Out[51]:
```
torch.Size([80, 3, 64, 64])
```

In [10]:
```
# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
for epoch in range(num_epochs):
    for i, data in enumerate(dataloader, 0):

        netD.zero_grad()
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, dtype=torch.float, device=device)
        output = netD(real_cpu).view(-1)
        errD_real = criterion(output, label)
        errD_real.backward()
        D_x = output.mean().item()

        noise = torch.randn(b_size, nz, 1, 1, device=device)
        fake = netG(noise)
        label.fill_(fake_label)
                                    ).view(-1)
        errD_fake = criterion(output, label)
```

```
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        errD = errD_real + errD_fake
        optimizerD.step()

        netG.zero_grad()
        label.fill_(real_label)
        output = netD(fake).view(-1)
        errG = criterion(output, label)
        errG.backward()
        D_G_z2 = output.mean().item()
        optimizerG.step()

        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

        G_losses.append(errG.item())
        D_losses.append(errD.item())

        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i == len(dataloader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

            if iters % 2000 == 0:
                from IPython.display import clear_output
                clear_output(wait=True)
                plt.imshow(np.transpose(img_list[-1],(1,2,0)))
                plt.show()

        iters += 1
```

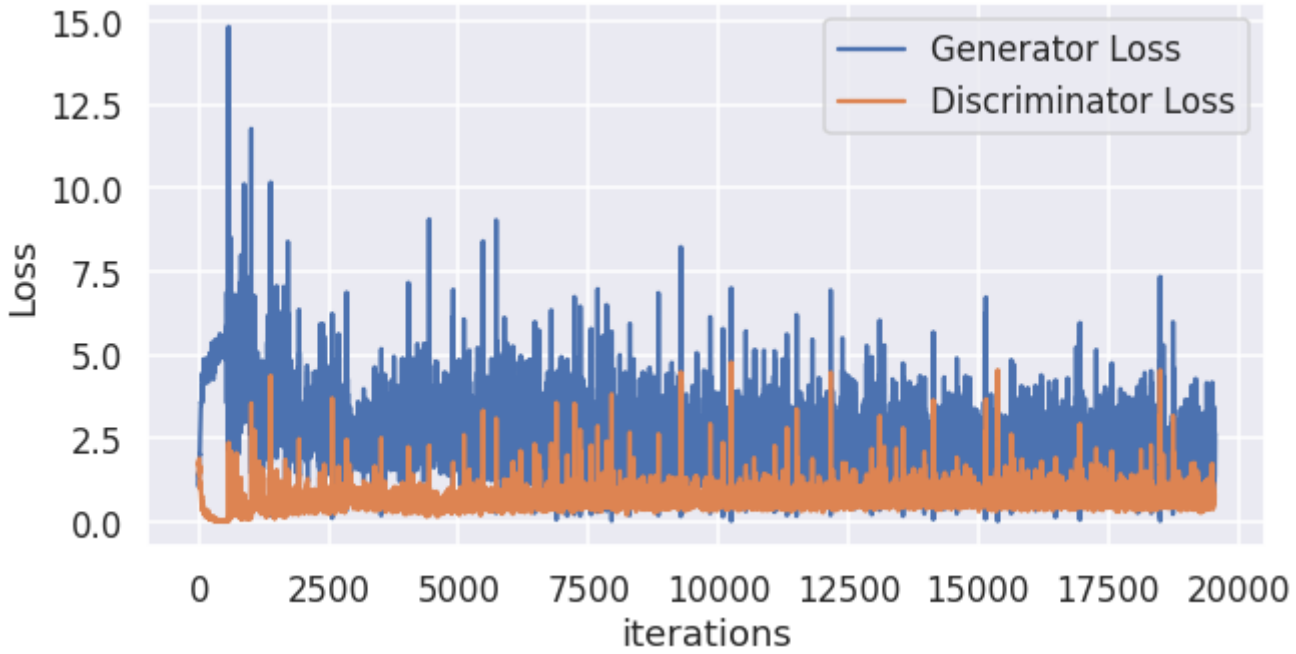


```
[46/50][50/391] Loss_D: 0.6193  Loss_G: 1.8647  D(x): 0.7617    D(G(z)): 0.2699 / 0.1783
[46/50][100/391]        Loss_D: 0.6315  Loss_G: 2.7376  D(x): 0.8861    D(G(z)): 0.3712 / 0.0832
[46/50][150/391]        Loss_D: 1.0081  Loss_G: 3.9434  D(x): 0.8684    D(G(z)): 0.5399 / 0.0287
[46/50][200/391]        Loss_D: 0.7171  Loss_G: 1.4903  D(x): 0.5964    D(G(z)): 0.1278 / 0.2678
[46/50][250/391]        Loss_D: 0.6504  Loss_G: 1.5800  D(x): 0.6975    D(G(z)): 0.2118 / 0.2473
[46/50][300/391]        Loss_D: 0.9808  Loss_G: 1.0193  D(x): 0.5199    D(G(z)): 0.2067 / 0.4001
[46/50][350/391]        Loss_D: 0.7105  Loss_G: 2.3447  D(x): 0.7267    D(G(z)): 0.2867 / 0.1264
[47/50][0/391]  Loss_D: 0.6053  Loss_G: 1.9207  D(x): 0.7807    D(G(z)): 0.2720 / 0.1726
[47/50][50/391] Loss_D: 0.5028  Loss_G: 2.1132  D(x): 0.8324    D(G(z)): 0.2537 / 0.1483
[47/50][100/391]        Loss_D: 0.6428  Loss_G: 2.0850  D(x): 0.7003    D(G(z)): 0.2174 / 0.1524
[47/50][150/391]        Loss_D: 0.9768  Loss_G: 2.5736  D(x): 0.8110    D(G(z)): 0.4803 / 0.1030
[47/50][200/391]        Loss_D: 0.6971  Loss_G: 2.0850  D(x): 0.6680    D(G(z)): 0.2129 / 0.1659
[47/50][250/391]        Loss_D: 0.6252  Loss_G: 1.9277  D(x): 0.7416    D(G(z)): 0.2493 / 0.1763
[47/50][300/391]        Loss_D: 0.5630  Loss_G: 2.0446  D(x): 0.8212    D(G(z)): 0.2820 / 0.1648
[47/50][350/391]        Loss_D: 0.6777  Loss_G: 1.4281  D(x): 0.6397    D(G(z)): 0.1584 / 0.2763
[48/50][0/391]  Loss_D: 0.6603  Loss_G: 2.6419  D(x): 0.8896    D(G(z)): 0.3882 / 0.0928
[48/50][50/391] Loss_D: 0.7849  Loss_G: 2.6424  D(x): 0.8224    D(G(z)): 0.4108 / 0.0952
[48/50][100/391]        Loss_D: 0.5159  Loss_G: 1.7206  D(x): 0.7955    D(G(z)): 0.2261 / 0.2179
[48/50][150/391]        Loss_D: 0.6360  Loss_G: 1.9760  D(x): 0.7095    D(G(z)): 0.2249 / 0.1708
[48/50][200/391]        Loss_D: 0.5180  Loss_G: 2.3675  D(x): 0.8571    D(G(z)): 0.2853 / 0.1117
[48/50][250/391]        Loss_D: 0.3567  Loss_G: 2.5601  D(x): 0.8664    D(G(z)): 0.1789 / 0.0979
[48/50][300/391]        Loss_D: 0.4887  Loss_G: 1.8650  D(x): 0.9100    D(G(z)): 0.3057 / 0.1869
[48/50][350/391]        Loss_D: 0.7300  Loss_G: 1.5562  D(x): 0.7531    D(G(z)): 0.3168 / 0.2488
[49/50][0/391]  Loss_D: 0.7758  Loss_G: 2.5398  D(x): 0.9562    D(G(z)): 0.4782 / 0.1060
[49/50][50/391] Loss_D: 1.0909  Loss_G: 0.8025  D(x): 0.4192    D(G(z)): 0.0727 / 0.4966
[49/50][100/391]        Loss_D: 0.6751  Loss_G: 1.9659  D(x): 0.6984    D(G(z)): 0.2281 / 0.1762
[49/50][150/391]        Loss_D: 0.6944  Loss_G: 1.6658  D(x): 0.5748    D(G(z)): 0.0866 / 0.2307
[49/50][200/391]        Loss_D: 0.6483  Loss_G: 2.4497  D(x): 0.6986    D(G(z)): 0.2221 / 0.1118
[49/50][250/391]        Loss_D: 0.6075  Loss_G: 2.0317  D(x): 0.7327    D(G(z)): 0.2249 / 0.1682
[49/50][300/391]        Loss_D: 0.7098  Loss_G: 1.6584  D(x): 0.7160    D(G(z)): 0.2718 / 0.2324
[49/50][350/391]        Loss_D: 0.3911  Loss_G: 2.5195  D(x): 0.8156    D(G(z)): 0.1516 / 0.1049
```

In [33]:

```
# Plot Losses

import seaborn as sns
import matplotlib.pyplot as plt

%matplotlib inline
sns.set_style('darkgrid')
sns.set_context('talk')
```

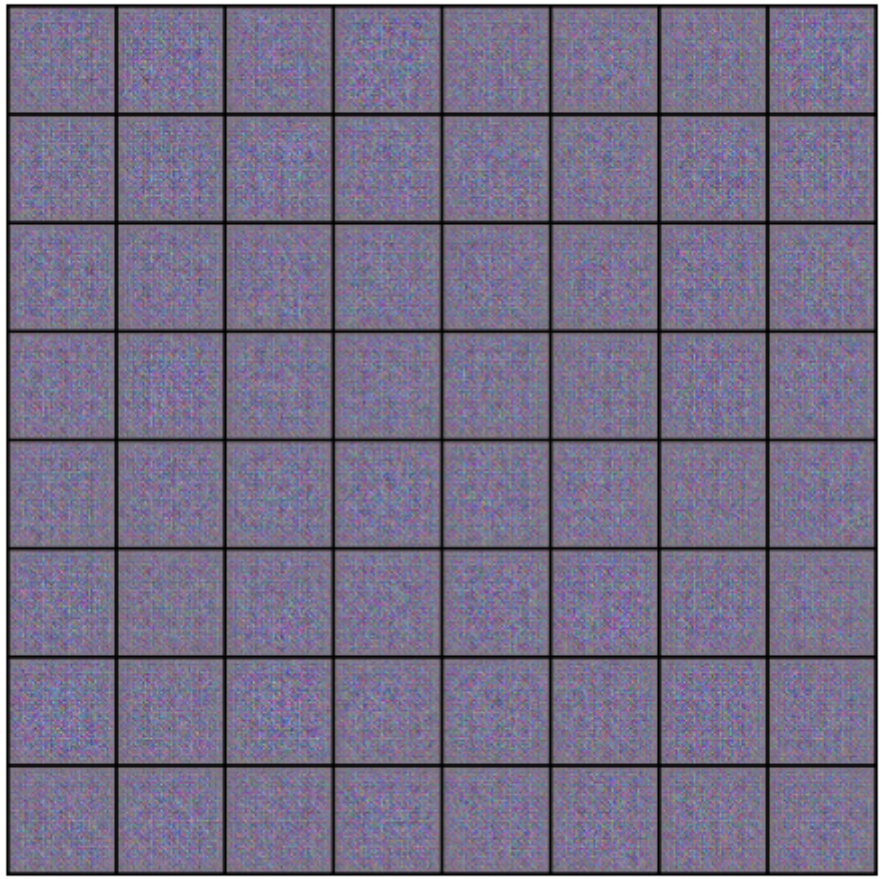
```
plt.figure(figsize=(10,5))
plt.plot(G_losses,label="Generator Loss")
plt.plot(D_losses,label="Discriminator Loss")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



In [12]:
```
# Visualize G's Progress
fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000, blit=True)

HTML(ani.to_jshtml())
```

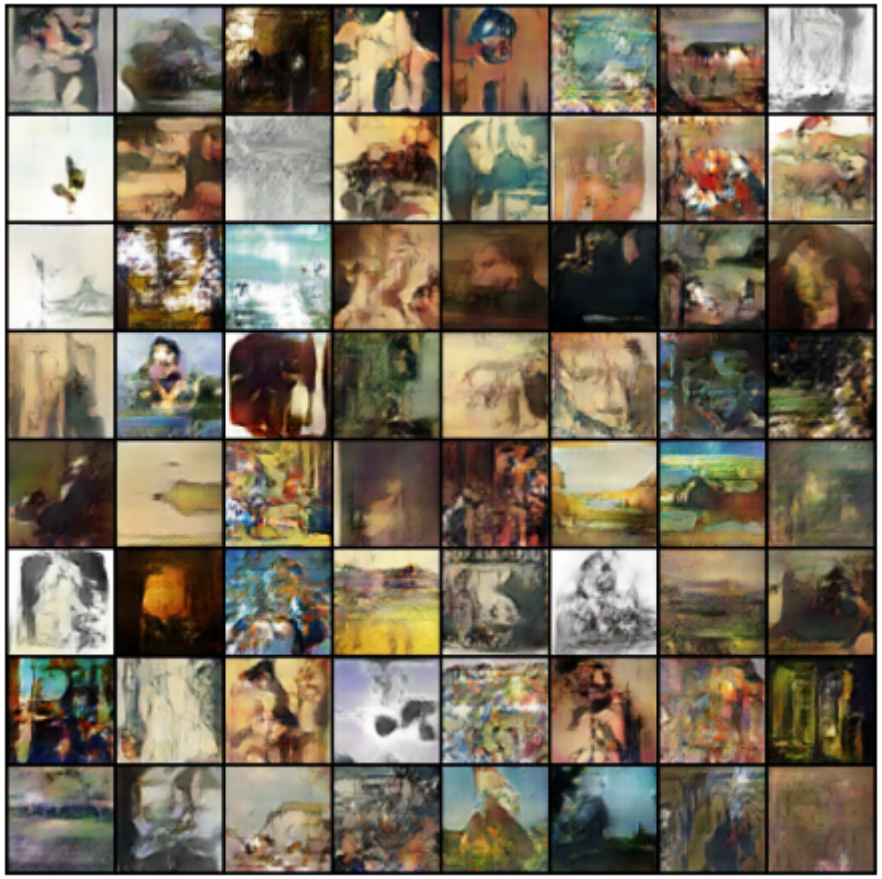
```
Animation size has reached 21475863 bytes, exceeding the limit of 20971520.0. If you're sure you want a larger ani
mbed_limit rc parameter to a larger value (in MB). This and further frames wil
l be dropped.
```

Out[12]:



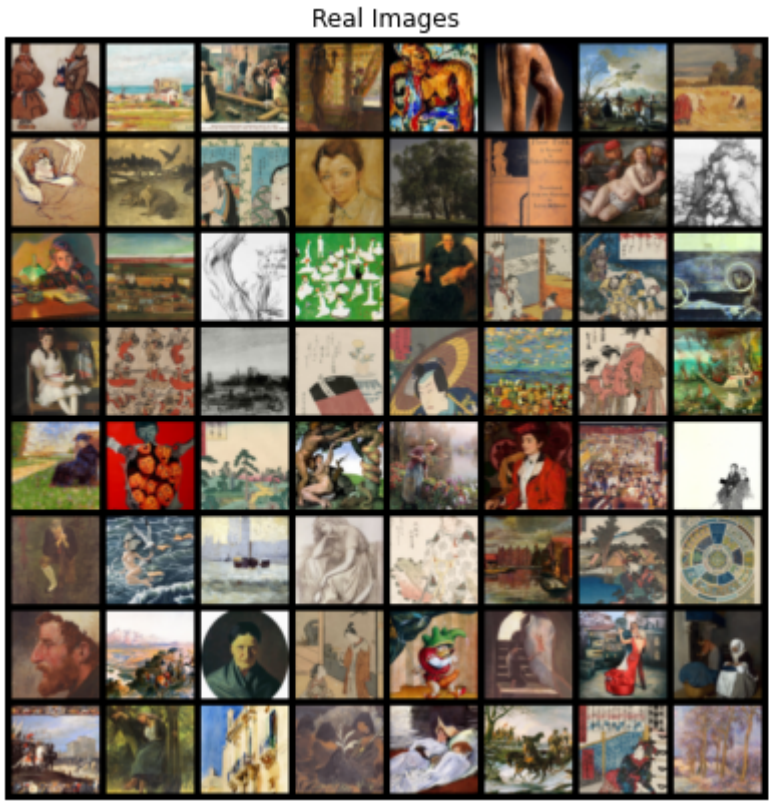
Once Loop Reflect

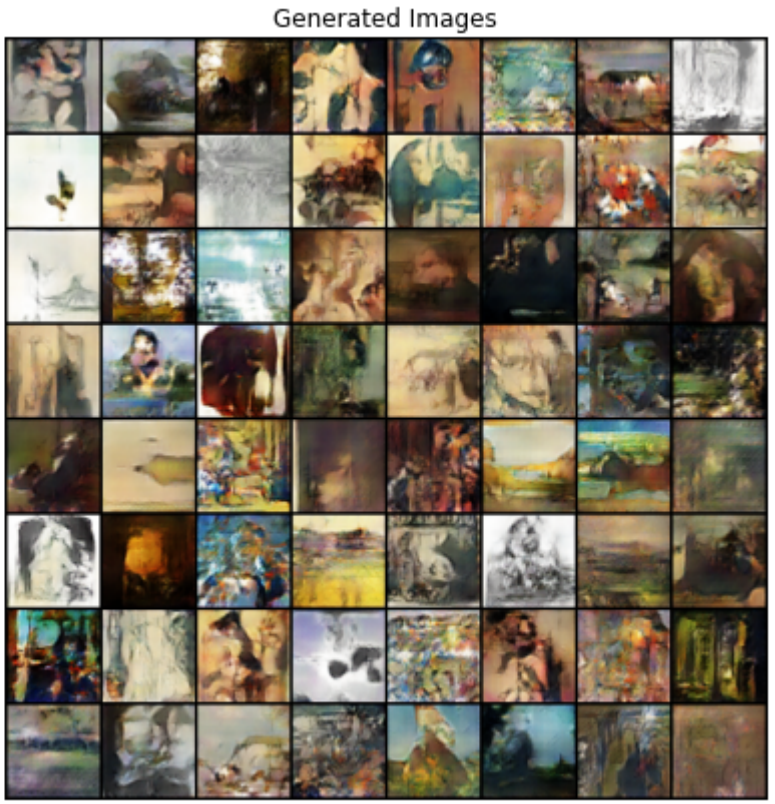
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js



In [14]:
```
# Real vs Fake
# Grab a batch of real images from the dataloader
real_batch = next(iter(dataloader))

# Plot the real images
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64], padding=5, normalize=True).cpu(),(1,2,0)))

                                                    epoch
plt.subplot(1,2,2)
```

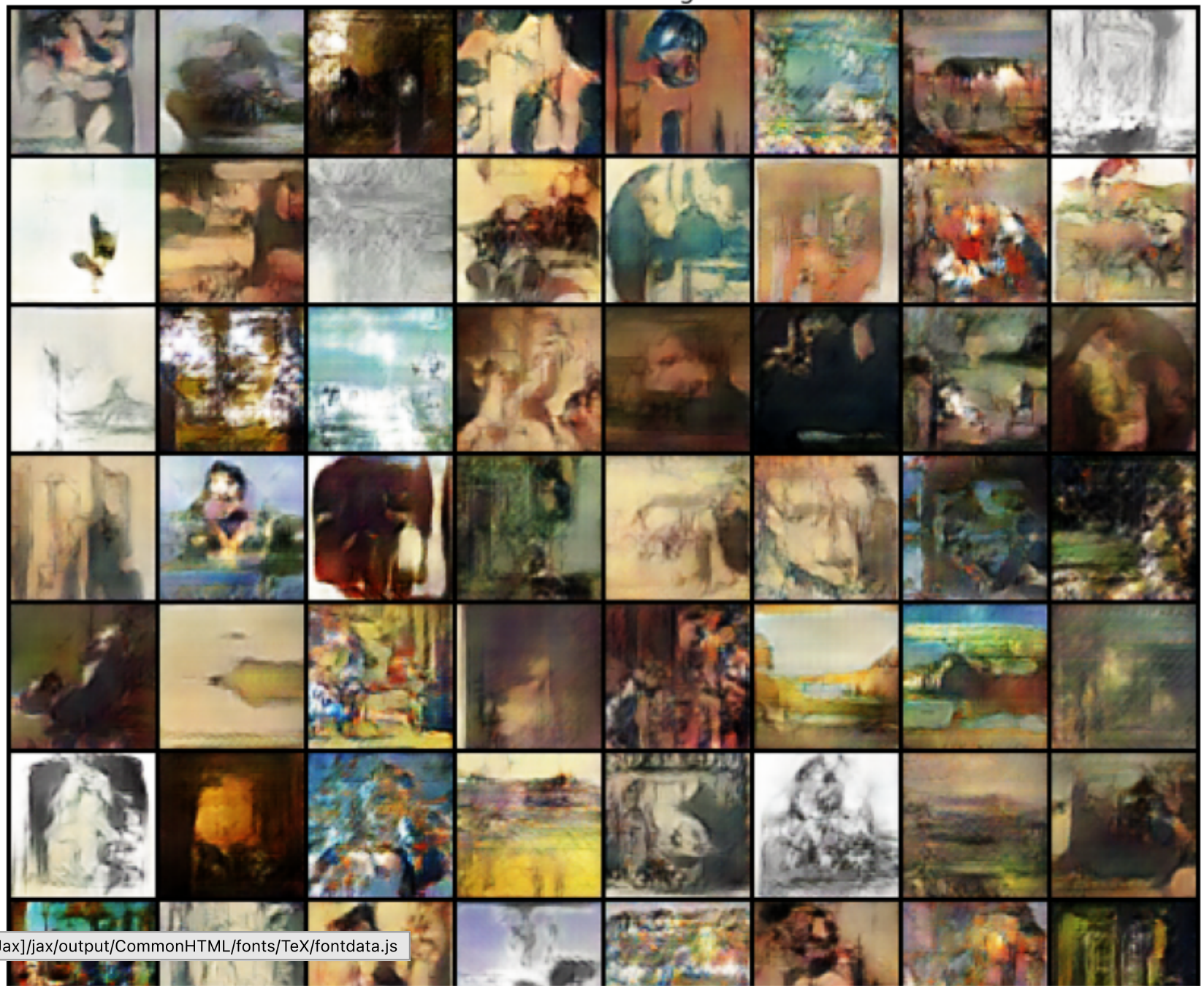
```
plt.axis("off")
plt.title("Generated Images")
plt.imshow(np.transpose(img_list[-1],(1,2,0)))
plt.show()
```





In [34]:
```
# Plot the fake images from the last epoch
plt.figure(figsize=(15,15))
plt.axis("off")
plt.title("Generated Images")
plt.imshow(np.transpose(img_list[-1],(1,2,0)))
plt.show()
```

Generated Images



In [24]:
```
# save the models

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

In [57]:
```
def get_dataset(x):
    image_size = 64
    batch_size = 80
    transform = transforms.Compose([transforms.Resize((image_size, image_size)),
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

    dataset = torchvision.datasets.ImageFolder(root="./data/artbench-10-imagefolder-split/{}/".format(x),transform=

    return DataLoader(dataset, batch_size, shuffle=True, num_workers=workers, pin_memory=True)

# Create the dataloader
dataloader = get_dataset("train")

# Create the dataloader
valloader = get_dataset("test")
```

In [62]:
```
import PIL.Image as Image
from ignite.engine import Engine, Events

from ignite.metrics import FID, InceptionScore

fid_metric = FID(device=device)
                                                    evice, output_transform=lambda x: x[0])
```

```
def interpolate(batch):
    arr = []
    for img in batch:
        pil_img = transforms.ToPILImage()(img.cpu())
        resized_img = pil_img.resize((299,299), Image.BILINEAR)
        arr.append(transforms.ToTensor()(resized_img))
    return torch.stack(arr)


def evaluation_step(engine, batch):
    with torch.no_grad():
        noise = torch.randn(80, nz, 1, 1, device=device)
        netG.eval()
        fake_batch = netG(noise)
        fake = interpolate(fake_batch)
        real = interpolate(batch[0])
        return fake, real

evaluator = Engine(evaluation_step)
fid_metric.attach(evaluator, "fid")
is_metric.attach(evaluator, "is")
```

In [63]:
```
# run the evaluator on the val data loader
evaluator.run(dataloader, max_epochs=1)
metrics = evaluator.state.metrics
fid_score = metrics['fid']
is_score = metrics['is']
print("====> For the train data loader:")
print("FID score: {}".format(fid_score))
print("Inception score: {}".format(is_score))
```

```
====> For the train data loader:
FID score: 0.016321113219323877
Inception score: 2.014124538808895
```

In [59]:
```
evaluator.run(valloader, max_epochs=1) # use your test data loader, NOT training data loader
metrics = evaluator.state.metrics
fid_score = metrics['fid']
is_score = metrics['is']
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
er:")
```

```
print("FID score: {}".format(fid_score))
print("Inception score: {}".format(is_score))
```

```
====> For the test data loader:
FID score: 0.016423227010168934
Inception score: 2.0025260146905204
```

In [ ]: