# Using Generative Adversarial Networks to generate art

**Mayank Sharma**
musharma@ucsd.edu
PID:A59019439

## 1 Introduction

The convergence of art and artificial intelligence (AI) presents an intriguing avenue for the creation of novel artistic content. This paper focuses on the application of Generative Adversarial Networks (GANs) to generate art using the Artbench-10 dataset, a rich collection of diverse artworks spanning various styles and periods.

GANs, characterized by their dual model architecture, have demonstrated strong capabilities in generating realistic content, and we seek to apply this potential in the realm of art. The Artbench-10 dataset, with its variety, provides an excellent platform for this exploration.

Our study investigates the proficiency of GANs in capturing and reproducing the essence of different art styles and their potential in contributing to the creation of novel, aesthetically appealing artistic content.

## 2 Dataset

The Artbench-10 dataset is a carefully curated collection of 60,000 pieces of art spanning across different influential artistic movements. The dataset is divided into a training set of 50,000 images and a test set of 10,000 images, offering a balanced mix for both model training and evaluation purposes.

This dataset encompasses a broad spectrum of artistic styles including:

- Art Nouveau
- Baroque
- Expressionism
- Impressionism
- Post-Impressionism
- Realism
- Renaissance
- Romanticism
- Surrealism
- Ukiyo-e

This diversity in the dataset provides a challenging and comprehensive platform to evaluate the capability of Generative Adversarial Networks in understanding and generating different artistic styles. Sample Images from the dataset can be seen in 1

## 3 Method

Generative Adversarial Networks (GANs) are a class of artificial intelligence algorithms used in unsupervised machine learning, first introduced by Ian Goodfellow et al. in 2014. They are composed of two neural networks - a generator and a discriminator - that are trained simultaneously.

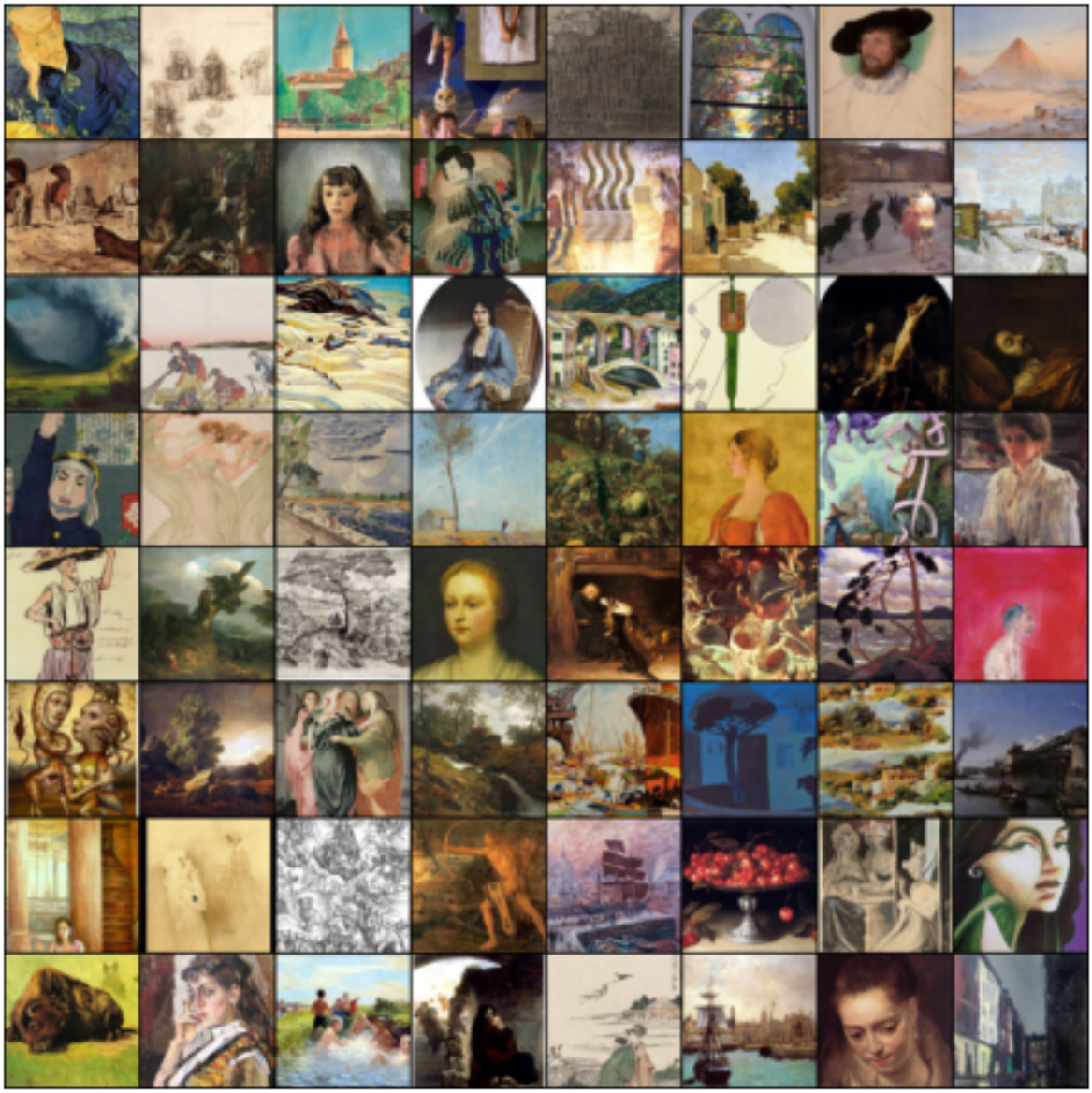


Figure 1: Real Images from the Dataset

The Generator is responsible for producing artificial outputs, such as images, that mimic the real data. It starts by taking a random noise vector as input, and over several layers of convolution and upsampling, generates an output. The objective of the generator is to produce data that is indistinguishable from the true data.

The Discriminator, on the other hand, is trained to differentiate between the real and artificial data. It receives either a real instance from the dataset or an artificial instance from the generator, and its goal is to correctly classify these instances as real or fake.

The key idea behind GANs is the adversarial relationship between the Generator and the Discriminator. The generator is trained to fool the discriminator, and in trying to do so, it learns to produce increasingly realistic data. Simultaneously, the discriminator is trying to correctly classify the data as real or fake, thereby becoming better at identifying the generator's creations.

This adversarial training process continues until the generator produces realistic data indistinguishable by the discriminator from the real data. The generator, thus, captures the data distribution of the training set, enabling the creation of new data instances that could have come from the same distribution.

In the context of generating artwork, GANs have shown impressive capabilities, creating novel and aesthetically pleasing images that are often indistinguishable from real artwork. However, they also pose unique challenges such as mode collapse, wherein the generator produces a limited diversity of samples, and training instability, due to the adversarial training dynamics.

### 3.1 Architecture

Our Generative Adversarial Architecture, implemented in PyTorch, involves two primary components: a generator and a discriminator. The model was trained on GPU for efficient computation

**Generator**:

The generator network is tasked with generating artificial images based on input noise. It starts with a latent space dimension (input noise) and transforms it into a 3x128x128 image. The architecture of the generator can be summarized as follows:

- A fully connected layer that maps the input latent vector to a feature space of size 512x4x4.
- Transposed convolution layers with a kernel size of 4 and stride of 2 to upsample the feature map while reducing the feature depth. These layers increase the spatial dimension from 4x4 to 8x8, then to 16x16, then to 32x32, and finally to 64x64. Each of these layers is followed by a batch normalization layer and a ReLU activation function.
- The final transposed convolution layer reduces the feature depth from 64 to 3 (corresponding to the RGB color channels) and upsamples the spatial dimension to 128x128. The output of this layer passes through a Tanh activation function to scale the pixel values between -1 and 1.

**Discriminator**:

The discriminator network is designed to differentiate between real images from the dataset and artificial images generated by the generator. The architecture of the discriminator is as follows:

- A series of convolutional layers with a kernel size of 4 and stride of 2 that reduce the spatial dimension of the input from 128x128 to 64x64, then to 32x32, then to 16x16, and finally to 8x8 and 4x4. Each of these layers (except the first one) is followed by a batch normalization layer and a LeakyReLU activation function with a negative slope of 0.2.
- The final convolution layer maps the 512x4x4 feature map to a single value that represents the probability that the input image is real. This output passes through a sigmoid activation function to scale the value between 0 and 1.

### 3.2 Loss function

In the adversarial training paradigm of Generative Adversarial Networks (GANs), the choice of a loss function is crucial to facilitate the contest between the generator and discriminator models. For our

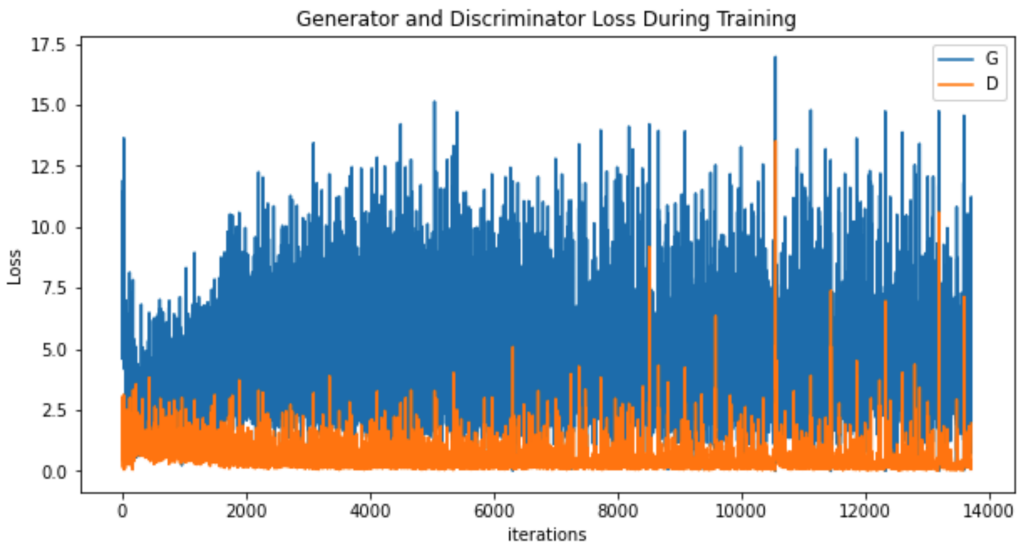



Figure 2: Training Loss Curve for Generator and Discriminator

study, we employ Binary Cross Entropy (BCE) as the primary loss function for both models. Owing to its effectiveness in models with binary outputs, BCE serves to quantify the discrepancy between the predicted and target probability distributions.

The loss function of the discriminator involves the accurate categorization of real images from the dataset and artificial images produced by the generator. Therefore, the target for real images is a vector of ones, whereas it is a vector of zeros for the synthetic images. The BCE loss is evaluated for both real and fake images and the total discriminator loss is the cumulative sum of these two.

The objective of the generator, conversely, is to confound the discriminator by generating images indistinguishable from the real ones. Hence, the target for the generator is a vector of ones, reflecting that the generated images should be classified as real. Consequently, the generator loss corresponds to the BCE loss computed with this particular target.

Through successive optimization of these loss functions, the generator progressively learns to fabricate more authentic images, and simultaneously, the discriminator enhances its ability to differentiate between real and artificial images. This symbiotic game progresses until the discriminator's ability to accurately distinguish between real and artificial images plateaus, signaling that the generator has successfully learned to produce convincing synthetic images.

### 3.3 Hyperparameter Settings

In our study on Generative Adversarial Networks (GANs), we have tuned several hyperparameters to optimize the training process. The number of training epochs is set to 70, ensuring that the generator and discriminator learn sufficiently from the dataset. The batch size of 256 strikes a balance between computational resources and gradient estimation accuracy.

We employ a learning rate of 0.0002 to control the step size during optimization, aiming for a balance between model convergence speed and stability. The input images are standardized to 128x128 pixel resolution and retain the RGB color channels.

For the training of both the generator and discriminator, we use the Adam optimizer, with a learning rate set to 0.0002. The betas parameters for Adam, set as (0.5, 0.999), control the decay rates of gradient and its square's moving averages, thereby affecting the stability and speed of the learning process.

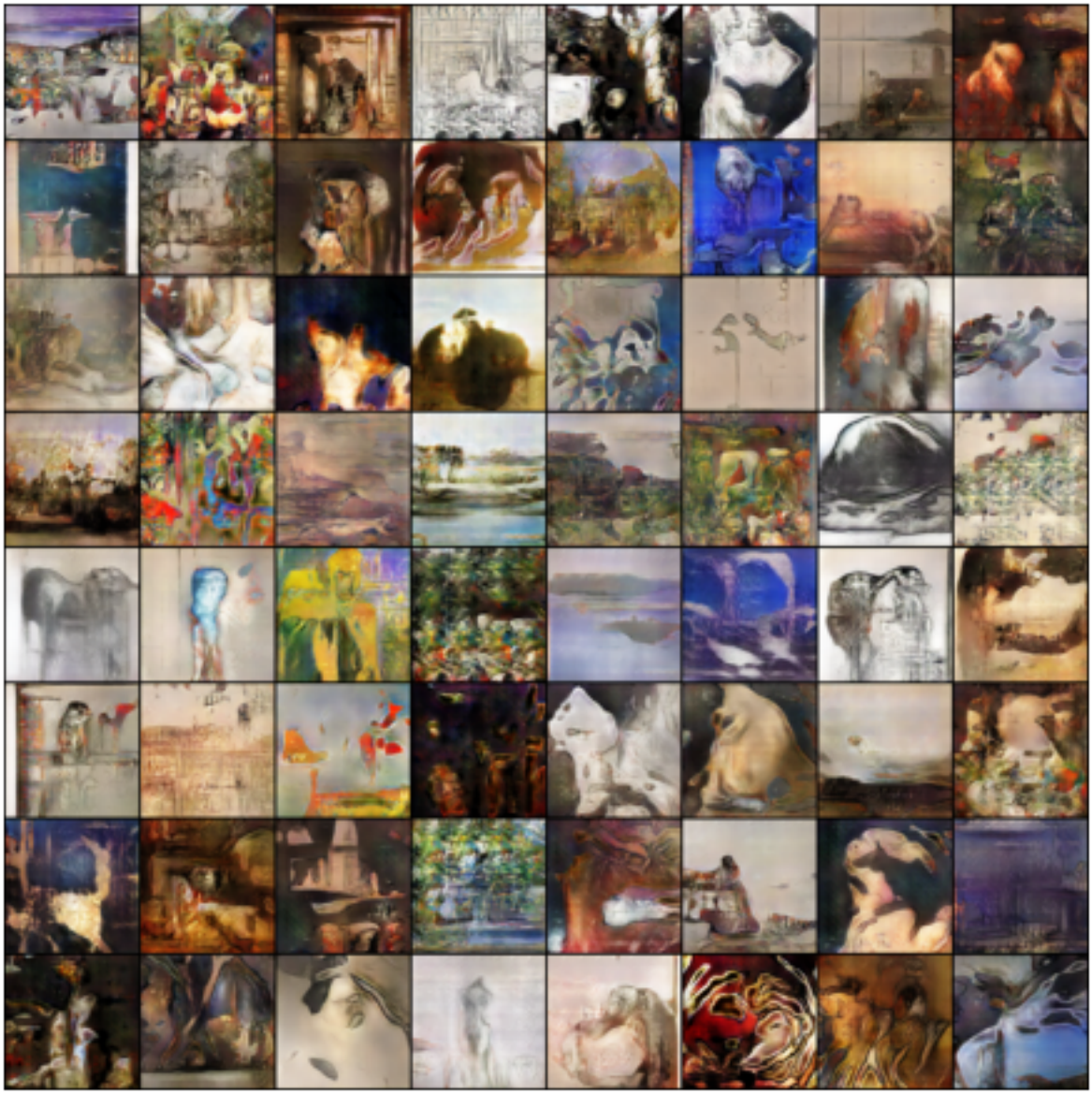


Figure 3: Generated Images from Standard Normal Distribution

### 3.4 Loss Curve

The training loss are shown in the curve 2.

## 4 Generated Art images

We use the trained generators of GANs to generate art images. Sample Images are generated using a random vector sampled from the standard normal distribution. Sample Images can be seen in 3.

## 5 FID and Inception Score

We will be using FID and Inception score as our metrics. FID measures the similarity between two sets of images, typically real images from a dataset and generated images from a model. It computes

a multivariate Gaussian distribution for each set, based on features extracted by an Inception-v3 model, and then calculates the Fréchet distance between these two distributions. A lower FID score indicates that the distributions of real and generated images are closer, signifying higher quality generated images. The Inception Score is a measure of the quality and diversity of images generated by generative models. It calculates the KL-divergence between the marginal class distribution and the conditional class distribution of the generated images. The score is obtained using a pre-trained Inception-v3 model. A higher Inception Score signifies better image quality and higher diversity among the generated images.

The FID and Inception Score obtained with our trained Generative Adversarial Network is mentioned in 1. The FID Score of 0.1562394577335805 suggests that our Generative Adversarial Network has done a good job generating chest X-ray images that share statistical similarities with the original dataset. The Inception Score of 76.5181117284821 suggests that the model is able to generate diverse images.

| Metric | Score |
|---|---|
| FID | 0.1562394577335805 |
| Inception Score | 76.5181117284821 |

Table 1: FID and Inception Scores for our Generative Adversarial Network Model

## 6 Code

### Architecture

```
import torch
from torch import nn

from torchsummary import summary
class Generator3x128x128(nn.Module):
    def __init__(self, latent_dim):
        super(Generator3x128x128, self).__init__()
        self.model = nn.Sequential(
            nn.ConvTranspose2d(latent_dim, 512, 4, 1, 0, bias=
               False),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            # state size. 512 x 4 x 4
            nn.ConvTranspose2d(512, 512, 4, 2, 1, bias=False),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            # state size. 512 x 8 x 8
            nn.ConvTranspose2d(512, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            # state size. 256 x 16 x 16
            nn.ConvTranspose2d(256, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            # state size. 128 x 32 x 32
            nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            # state size. 64 x 64 x 64
            nn.ConvTranspose2d(64, 3, 4, 2, 1, bias=False),
            nn.Tanh()
            # final state size. 3 x 128 x 128
        )

```

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

netG = Generator3x128x128(latent_dim).to(device)
print("Generator Summary")
print(summary(netG, (latent_dim, 1, 1)))

class Discriminator3x128x128(nn.Module):
    def __init__(self):
        super(Discriminator3x128x128, self).__init__()
        self.model = nn.Sequential(
            # input is 3 x 128 x 128
            nn.Conv2d(3, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. 64 x 64 x 64
            nn.Conv2d(64, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. 128 x 32 x 32
            nn.Conv2d(128, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. 256 x 16 x 16
            nn.Conv2d(256, 512, 4, 2, 1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. 512 x 8 x 8
            nn.Conv2d(512, 512, 4, 2, 1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. 512 x 4 x 4
            nn.Conv2d(512, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.model(x)
        return x

netD = Discriminator3x128x128().to(device)
print("Discriminator Summary")
print(summary(netD, (3, 128, 128)))
```

**Training**

```
# Define the optimizers and the loss (BCELoss)
optimizerD = torch.optim.Adam(netD.parameters(), lr=
    learning_rate, betas=(0.5, 0.999))
optimizerG = torch.optim.Adam(netG.parameters(), lr=
    learning_rate, betas=(0.5, 0.999))

criterion = nn.BCELoss()

fixed_noise = torch.randn(64, latent_dim, 1, 1, device=device)

real_label = 1
fake_label = 0

img_list = []
G_losses = []
D_losses = []
iters = 0

real_label = 1
fake_label = 0

# Training loop
for epoch in range(epochs):
    for i, data in enumerate(dataloader, 0):
        netG.train()
        netD.train()

        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, latent_dim, 1, 1, device=
            device)
        fake = netG(noise)
        label.fill_(fake_label)
        output = netD(fake.detach()).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()

```

```
        print('[%d/%d][%d/%d] Loss_D: %.4f Loss_G: %.4f D(x): %.4f D(G(z)): %.4f / %.4f'
              % (epoch, 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 == 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))
        iters+=1
```

**FID and Inception Score**

```
from ignite.metrics import FID, InceptionScore
fid_metric = FID(device=device)
is_metric = InceptionScore(device=device, output_transform=lambda x: x[0])

import PIL.Image as Image


def evaluation_step_new(engine, batch):
    with torch.no_grad():
        actual_batch_size = batch[0].size(0)
        noise = torch.randn(actual_batch_size, latent_dim, 1, 1, device=device)
        netG.eval()
        fake_batch = netG(noise)
        fake = fake_batch
        real = batch[0]
        return fake, real

from ignite.engine import Engine, Events
from ignite.contrib.handlers import ProgressBar

evaluator = Engine(evaluation_step_new)
fid_metric.attach(evaluator, "fid")
is_metric.attach(evaluator, "is")
# Create a progress bar object
pbar = ProgressBar(persist=True)

# Attach progress bar to evaluator
pbar.attach(evaluator)

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

test_dataset = ImageFolder(root='artbench-10-imagefolder-split/test', transform=transform)
```

```
test_dataloader = DataLoader(test_dataset, batch_size=256,
    shuffle=True, num_workers=15)

@evaluator.on(Events.ITERATION_STARTED(every=10))
def log_training_loss(engine):
    print(f"Epoch[{engine.state.epoch}] Iteration[{engine.state
        .iteration}] Loss: {engine.state.output}")


evaluator.run(test_dataloader,max_epochs=1)
metrics = evaluator.state.metrics
fid_score = metrics['fid']
is_score = metrics['is']
print("fid_score", fid_score)
print("is_score", is_score)
```

The entire code used is mentioned in the next page.

# Mayank_Deep_Gen_Assignment_4

May 31, 2023

[4]:
```
import torch
from torch import nn
from torchvision import transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
import torchvision.utils as vutils

# Define hyperparameters
epochs = 70
batch_size = 256
learning_rate = 0.0002
image_size = 128
channels = 3 # RGB images
latent_dim = 100
features_gen = 64
features_dis = 64

# Load and preprocess the ArtBench10 dataset
transform = transforms.Compose([transforms.Resize(image_size),
                                transforms.CenterCrop(image_size),
                                transforms.ToTensor(),
                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.
 ↪5, 0.5))])

dataset = ImageFolder(root='artbench-10-imagefolder-split/train',␣
 ↪transform=transform)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True,␣
 ↪num_workers=15)

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

```
device:  cuda
```

[5]:
```
!pip install torchsummary
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: torchsummary in
```

```
/home/musharma/.local/lib/python3.9/site-packages (1.5.1)
```

[6]:
```
from torchsummary import summary
class Generator3x128x128(nn.Module):
    def __init__(self, latent_dim):
        super(Generator3x128x128, self).__init__()
        self.model = nn.Sequential(
            nn.ConvTranspose2d(latent_dim, 512, 4, 1, 0, bias=False),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            # state size. 512 x 4 x 4
            nn.ConvTranspose2d(512, 512, 4, 2, 1, bias=False),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            # state size. 512 x 8 x 8
            nn.ConvTranspose2d(512, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            # state size. 256 x 16 x 16
            nn.ConvTranspose2d(256, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            # state size. 128 x 32 x 32
            nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            # state size. 64 x 64 x 64
            nn.ConvTranspose2d(64, 3, 4, 2, 1, bias=False),
            nn.Tanh()
            # final state size. 3 x 128 x 128
        )

    def forward(self, x):
        x = self.model(x)
        return x

netG = Generator3x128x128(latent_dim).to(device)
print("Generator Summary")
print(summary(netG, (latent_dim, 1, 1)))

class Discriminator3x128x128(nn.Module):
    def __init__(self):
        super(Discriminator3x128x128, self).__init__()
        self.model = nn.Sequential(
            # input is 3 x 128 x 128
            nn.Conv2d(3, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
```

```
            # state size. 64 x 64 x 64
            nn.Conv2d(64, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. 128 x 32 x 32
            nn.Conv2d(128, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. 256 x 16 x 16
            nn.Conv2d(256, 512, 4, 2, 1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. 512 x 8 x 8
            nn.Conv2d(512, 512, 4, 2, 1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. 512 x 4 x 4
            nn.Conv2d(512, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.model(x)
        return x

netD = Discriminator3x128x128().to(device)
print("Discriminator Summary")
print(summary(netD, (3, 128, 128)))
```

```
Generator Summary
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
   ConvTranspose2d-1            [-1, 512, 4, 4]         819,200
       BatchNorm2d-2            [-1, 512, 4, 4]           1,024
              ReLU-3            [-1, 512, 4, 4]               0
   ConvTranspose2d-4            [-1, 512, 8, 8]       4,194,304
       BatchNorm2d-5            [-1, 512, 8, 8]           1,024
              ReLU-6            [-1, 512, 8, 8]               0
   ConvTranspose2d-7          [-1, 256, 16, 16]       2,097,152
       BatchNorm2d-8          [-1, 256, 16, 16]             512
              ReLU-9          [-1, 256, 16, 16]               0
  ConvTranspose2d-10          [-1, 128, 32, 32]         524,288
      BatchNorm2d-11          [-1, 128, 32, 32]             256
             ReLU-12          [-1, 128, 32, 32]               0
  ConvTranspose2d-13           [-1, 64, 64, 64]         131,072
      BatchNorm2d-14           [-1, 64, 64, 64]             128
```

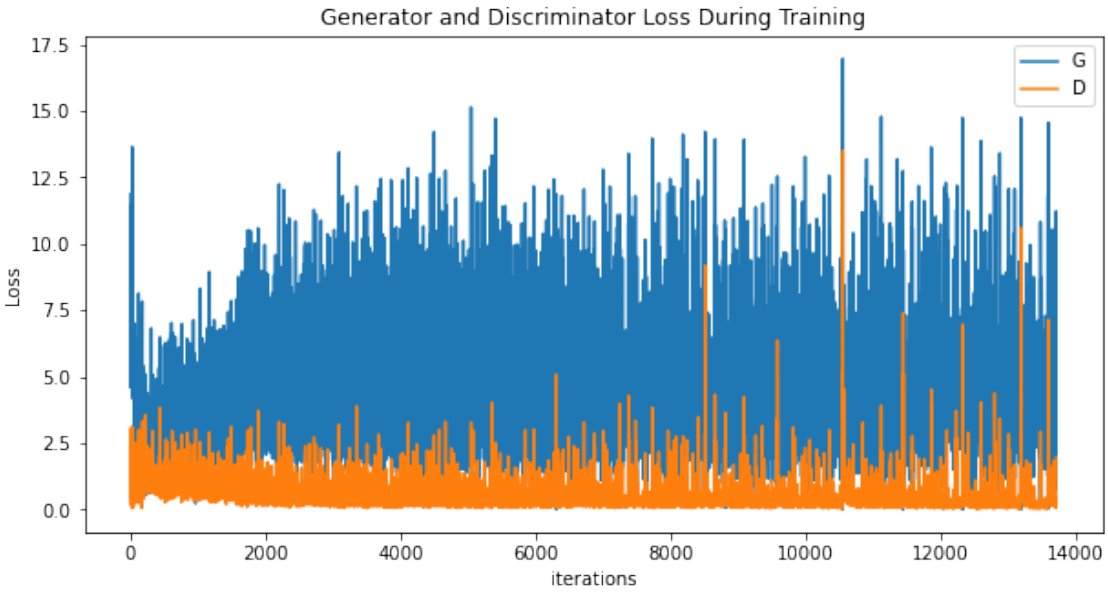
```
            ReLU-15           [-1, 64, 64, 64]               0
  ConvTranspose2d-16          [-1, 3, 128, 128]           3,072
            Tanh-17          [-1, 3, 128, 128]               0
================================================================
Total params: 7,772,032
Trainable params: 7,772,032
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 12.19
Params size (MB): 29.65
Estimated Total Size (MB): 41.84
----------------------------------------------------------------
None
Discriminator Summary
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 64, 64]           3,072
         LeakyReLU-2           [-1, 64, 64, 64]               0
            Conv2d-3          [-1, 128, 32, 32]         131,072
       BatchNorm2d-4          [-1, 128, 32, 32]             256
         LeakyReLU-5          [-1, 128, 32, 32]               0
            Conv2d-6          [-1, 256, 16, 16]         524,288
       BatchNorm2d-7          [-1, 256, 16, 16]             512
         LeakyReLU-8          [-1, 256, 16, 16]               0
            Conv2d-9            [-1, 512, 8, 8]       2,097,152
      BatchNorm2d-10            [-1, 512, 8, 8]           1,024
        LeakyReLU-11            [-1, 512, 8, 8]               0
           Conv2d-12            [-1, 512, 4, 4]       4,194,304
      BatchNorm2d-13            [-1, 512, 4, 4]           1,024
        LeakyReLU-14            [-1, 512, 4, 4]               0
           Conv2d-15              [-1, 1, 1, 1]           8,192
          Sigmoid-16              [-1, 1, 1, 1]               0
================================================================
Total params: 6,960,896
Trainable params: 6,960,896
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.19
Forward/backward pass size (MB): 9.44
Params size (MB): 26.55
Estimated Total Size (MB): 36.18
----------------------------------------------------------------
None
```

[7]:
```
# Define the optimizers and the loss (BCELoss)
optimizerD = torch.optim.Adam(netD.parameters(), lr=learning_rate, betas=(0.5,␣
↪0.999))
optimizerG = torch.optim.Adam(netG.parameters(), lr=learning_rate, betas=(0.5,␣
↪0.999))

criterion = nn.BCELoss()

fixed_noise = torch.randn(64, latent_dim, 1, 1, device=device)

real_label = 1
fake_label = 0

img_list = []
G_losses = []
D_losses = []
iters = 0
```

[ ]:
```
real_label = 1
fake_label = 0

# Training loop
for epoch in range(epochs):
    for i, data in enumerate(dataloader, 0):
        netG.train()
        netD.train()

        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, latent_dim, 1, 1, device=device)
        fake = netG(noise)
        label.fill_(fake_label)
        output = netD(fake.detach()).view(-1)
        errD_fake = criterion(output, label)
        errD_fake.backward()
        D_G_z1 = output.mean().item()

        errD = errD_real + errD_fake
        optimizerD.step()
```

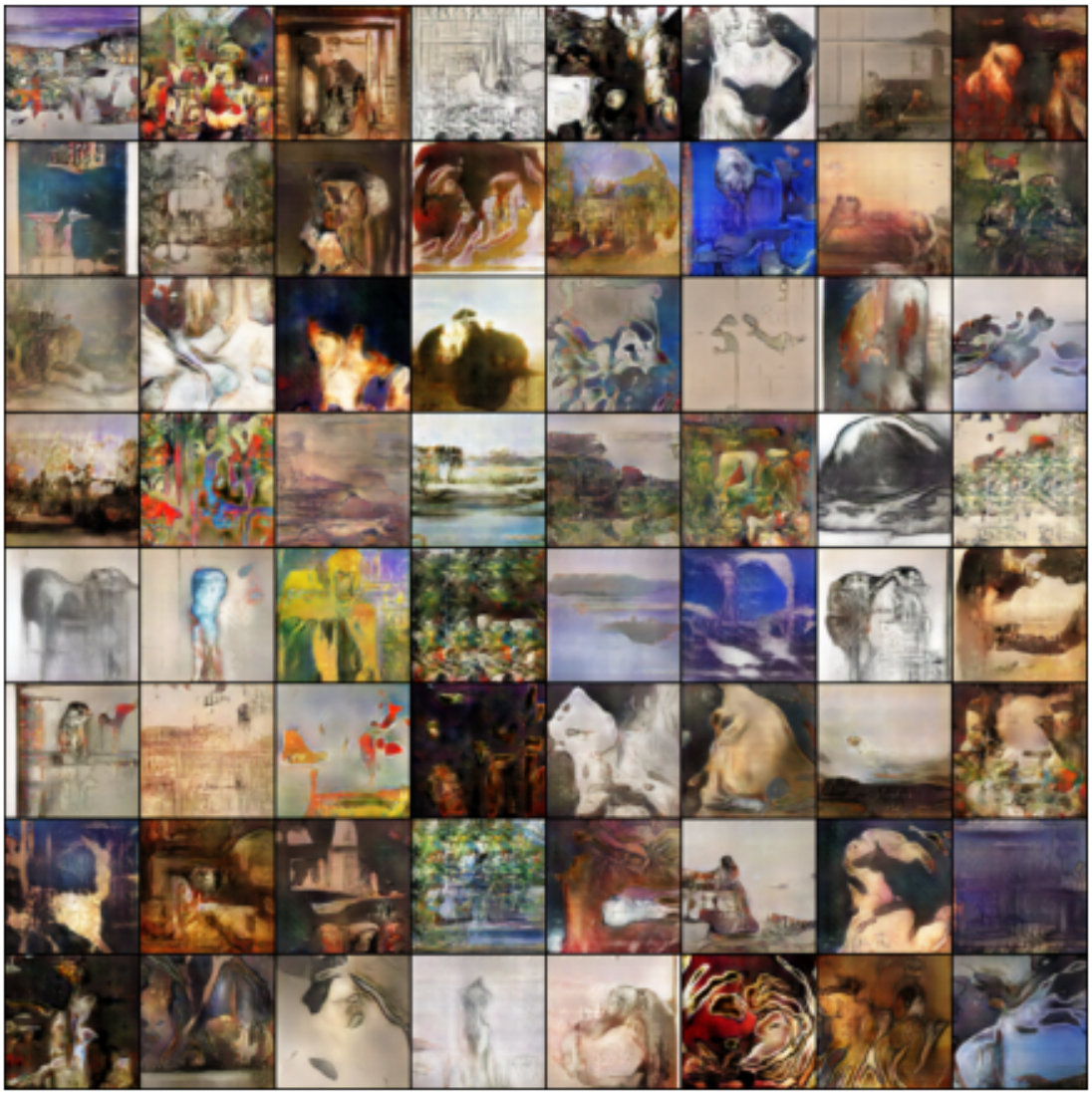
```
        netG.zero_grad()
        label.fill_(real_label)
        output = netD(fake).view(-1)
        errG = criterion(output, label)
        errG.backward()
        D_G_z2 = output.mean().item()

        optimizerG.step()

        print('[%d/%d][%d/%d] Loss_D: %.4f Loss_G: %.4f D(x): %.4f D(G(z)): %.
↪4f / %.4f'
              % (epoch, 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 == 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))
        iters+=1
```

[9]:
```
import matplotlib.pyplot as plt

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




[10]:
```
import numpy as np
import matplotlib.animation as animation
from IPython.display import HTML

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())
```

[10]: <IPython.core.display.HTML object>

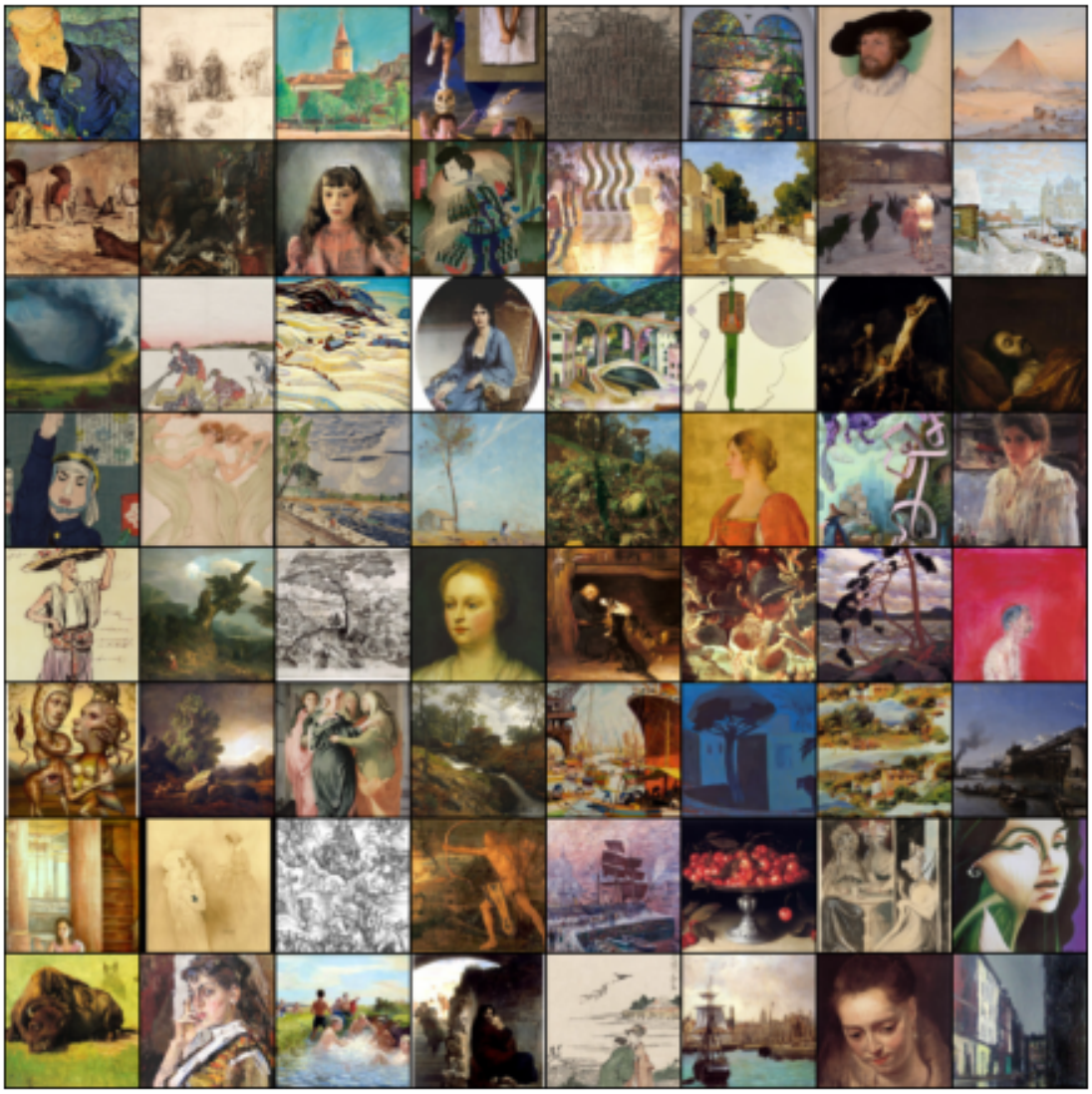


[11]:
```
# Save the models
torch.save(netG.state_dict(), 'generator.pth')
torch.save(netD.state_dict(), 'discriminator.pth')
```

[ ]:
```
netG.load_state_dict(torch.load('generator.pth'))
netD.load_state_dict(torch.load('discriminator.pth'))
```

[91]:
```
import matplotlib.pyplot as plt
import torchvision.utils as vutils

# Assuming that you have loaded data from train_dataloader
dataiter = iter(dataloader)
images, _ = dataiter.next()
```

```
# Normalize the images for display
img_grid = vutils.make_grid(images[:64], padding=2, normalize=True)

# Transpose the image grid for displaying
img_grid = np.transpose(img_grid, (1, 2, 0))

plt.figure(figsize=(8, 8))
plt.axis("off")
plt.imshow(img_grid)
plt.show()
```



[94]:
```
from ignite.metrics import FID, InceptionScore
fid_metric = FID(device=device)
is_metric = InceptionScore(device=device, output_transform=lambda x: x[0])

import PIL.Image as Image


def evaluation_step_new(engine, batch):
    with torch.no_grad():
        actual_batch_size = batch[0].size(0)
        noise = torch.randn(actual_batch_size, latent_dim, 1, 1, device=device)
        netG.eval()
        fake_batch = netG(noise)
        fake = fake_batch
        real = batch[0]
        return fake, real

from ignite.engine import Engine, Events
from ignite.contrib.handlers import ProgressBar

evaluator = Engine(evaluation_step_new)
fid_metric.attach(evaluator, "fid")
is_metric.attach(evaluator, "is")
# Create a progress bar object
pbar = ProgressBar(persist=True)

# Attach progress bar to evaluator
pbar.attach(evaluator)

transform = transforms.Compose([transforms.Resize(image_size),
                                transforms.CenterCrop(image_size),
                                transforms.ToTensor(),
                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.
 ↪5, 0.5))])

test_dataset = ImageFolder(root='artbench-10-imagefolder-split/test',␣
 ↪transform=transform)
test_dataloader = DataLoader(test_dataset, batch_size=256, shuffle=True,␣
 ↪num_workers=15)

@evaluator.on(Events.ITERATION_STARTED(every=10))
def log_training_loss(engine):
    print(f"Epoch[{engine.state.epoch}] Iteration[{engine.state.iteration}]␣
 ↪Loss: {engine.state.output}")


evaluator.run(test_dataloader,max_epochs=1)
```

```
metrics = evaluator.state.metrics
fid_score = metrics['fid']
is_score = metrics['is']
print("fid_score", fid_score)
print("is_score", is_score)
```

```
[1/40]   2%|2          [00:00<?]

Epoch[1] Iteration[10] Loss: None
Epoch[1] Iteration[20] Loss: None
Epoch[1] Iteration[30] Loss: None
Epoch[1] Iteration[40] Loss: None
fid_score 0.1562394577335805
is_score 76.5181117284821
```

[ ]: