I am Monet: Generating Monet Paintings with Deep Learning

CS 5661 Final Project Report

Ly Jacky Nhiayi, Erica Payne, Leonardo Ramirez, Leonard Garcia, Jackson Bentley, Angel Villalobos, Jesus Cruz, David Camacho, Youssef Elzein, Matthew Johnson

Table of Contents

Introduction:

For this project, we set out to determine how machine learning image generation has advanced in recent years. With the recent boom AI generation has been enjoying, it is clear that the technology is now able to mimic objects in a convincing way - but can it produce works of art similar to those of a legendary artist like Monet? Setting aside the metaphysical questions of the nature of art and whether a machine could ever truly create it, we sought to take a practical approach by striving to replicate Monet's visual style with leading data science techniques at our disposal.

For this project, we split into two teams to explore different potential avenues of progress. One team focused on leveraging Generative Adversarial Networks (GANs), a powerful framework for generating realistic images. The second team adopted diffusion models, which are another type of generative model; unlike GANs, diffusion models operate by iteratively updating pixel values based on neighboring information, which leads to complex and organic patterns. Each team used different methods and variations of the two models to determine which one actually produced the best results. As it turned out, the varied strengths and weaknesses of each model made for an interesting and unpredictable comparison.

Goal:

Our primary goal was to train a neural network to generate Monet-style paintings using the dataset provided by Kaggle. We aimed to create new images that capture his style and compare the generated art that was produced by two different categories of models, GANs and Diffusion.

Dataset:

For this project, we decided to use one of the datasets offered by Kaggle and in a competition to see who could obtain the best results. The dataset contained four directories, two which were in jpeg format and the other two in a TFRecords format. They provided both because the latter is a somewhat new format which some frameworks might not accommodate for yet. We used the TFRecords version for this project as it was a great way to get familiar with the new format.

Monet_jpg - 300 Monet paintings sized 265x256 in JPEG format

Photo_jpg - 7028 photos sized 256x256 in JPEG format

The photo_jpg directory contained photos which served as templates for demonstrating the working Monet image generation for the purposes of the competition, but we opted to neglect this step and simply focus on creating the best output that we could.

GANS:

A GAN (Generative Adversarial Network) is a machine learning model in which two neural networks compete with each other using deep learning methods to become more accurate. GANs are a significant innovation in machine learning - they run unsupervised typically and use a cooperative zero-sum game framework to learn, where one person’s gain is another person’s loss.

Generative

The generative part of GANs is responsible for creating fake data. Its main purpose is to make the discriminator classify its output as real samples incorrectly. As such, the main application of a typical GAN generator is to create fake samples that look realistic.

Adversarial

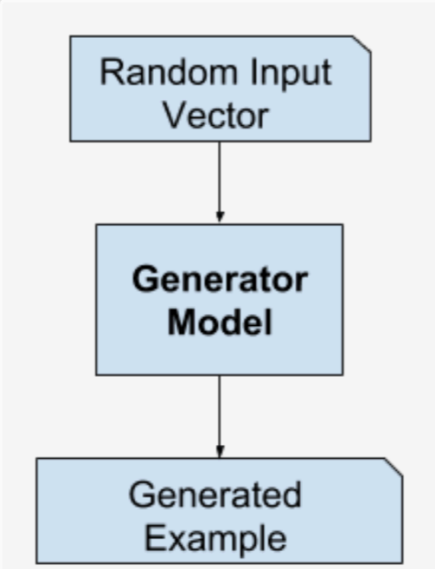
The generator's goal is to maximize the error probability of the discriminator while the discriminator's goal is to minimize its error probability. Through training iterations both networks evolve until they reach an equilibrium state. Supposing the training was successful, in this final state the discriminator cannot distinguish between real and fake data, which means the generator is able to produce very convincing fake images.

Network

Both the generator and discriminator are neural networks where the generator's output is directly connected to the discriminator's input. The networks' structures can vary significantly, for example they could take the form of a deep convolutional or fully connected network.

Discriminator

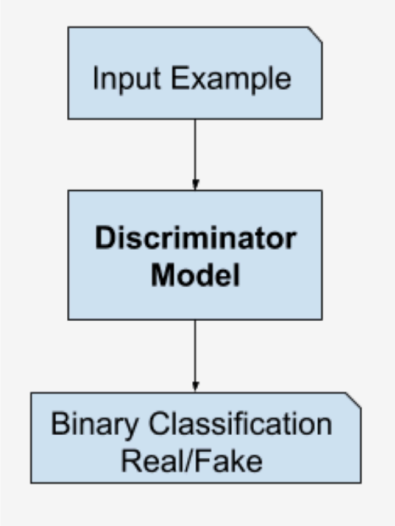
The generator is often a convolutional neural network (CNN), a deep learning algorithm that can process an input image, differentiate between the objects within it, and determine labels for given inputs. In the case of a GAN, the discriminator strives to accurately classify images as being either output from the generator or belonging to the training set. Importantly, in an ideal training for a GAN, the discriminator fails at this task, reaching equilibrium at a state where it cannot differentiate between the two.



Example of the GAN Generator Model

Generator

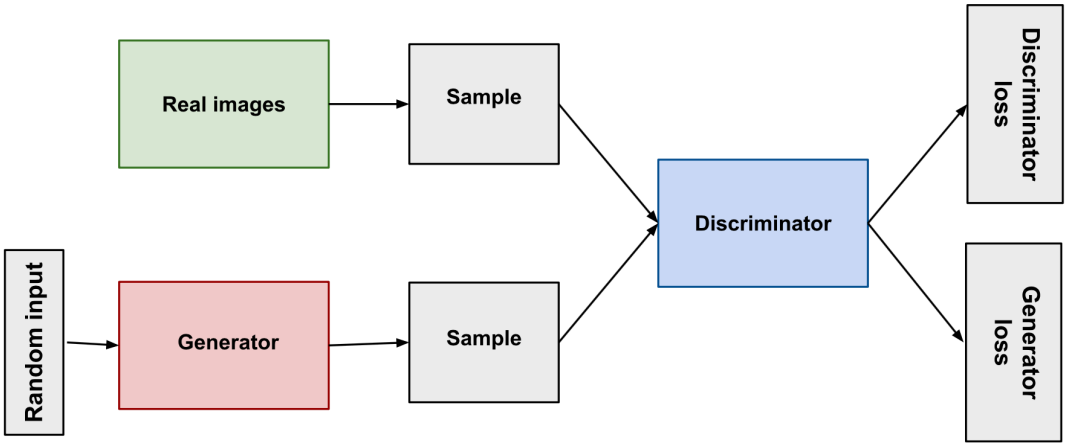
These are deconvolutional neural networks (DNN). These algorithms work in reverse to CNNs, and aim to identify the features of an input that were either missed by the CNN or convoluted with other signals. As mentioned, the goal is to fool the discriminator by producing highly realistic fake output.



Example of the GAN Discriminator Model

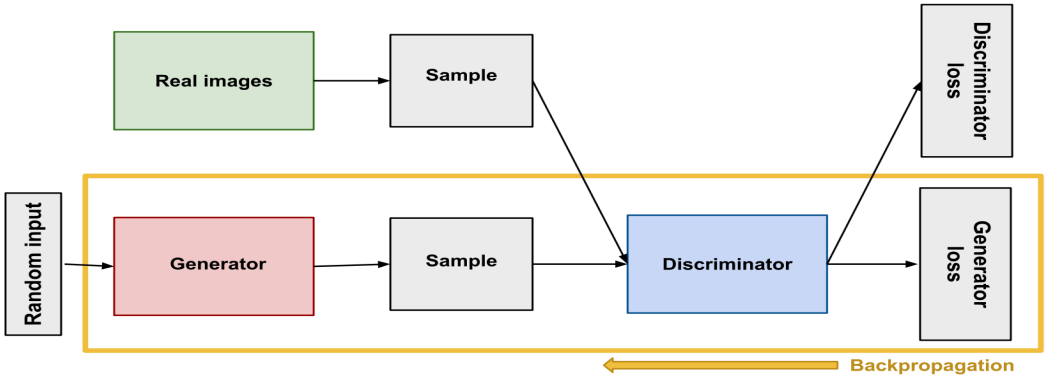
Generator Loss

The generator loss measures how effectively the generator is able to deceive the discriminator in classifying the data - typically, this is calculated using a binary cross entropy loss. The generator loss penalizes the generator when the discriminator classifies the generated examples as fake, prompting the generator to update its weights. In GANs the generator is not directly connected to its loss, as instead the generator's loss is connected to the discriminator's output. This loss is calculated after the discriminator classifies the data produced by the generator.



Generator Backpropagation

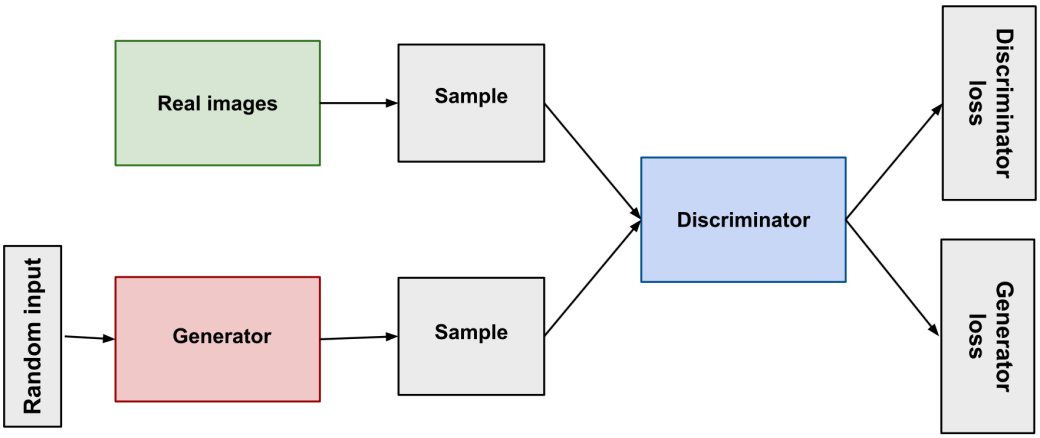
Once the loss is computed, the gradients of the loss function with respect to the parameters of the generator network are calculated using backpropagation. This process involves propagating the error backwards through the network computing the gradients of the loss with respect to each parameter at every layer of the network. After computing the gradients the parameters of the generator network are updated using an optimization algorithm such as stochastic gradient descent.



Discriminator Loss

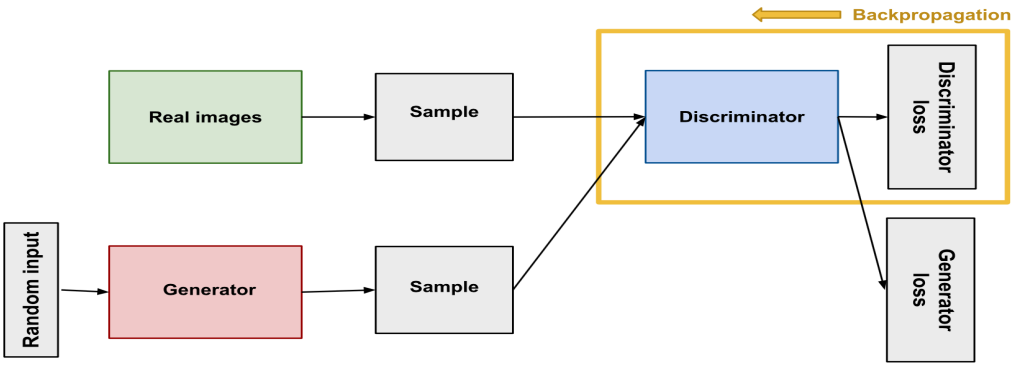
The discriminator loss measures how effectively the discriminator network can distinguish between real data and fake data produced by the generator. This is typically calculated using a

binary cross entropy loss, penalizing the discriminator for misclassifying the real data as fake and the fake data as real. The discriminator performs two separate loss calculations: one loss associated with correctly classifying real samples as real and the other associated with correctly classifying the fake samples as fake. At the end both losses are combined to obtain the total discriminator loss.



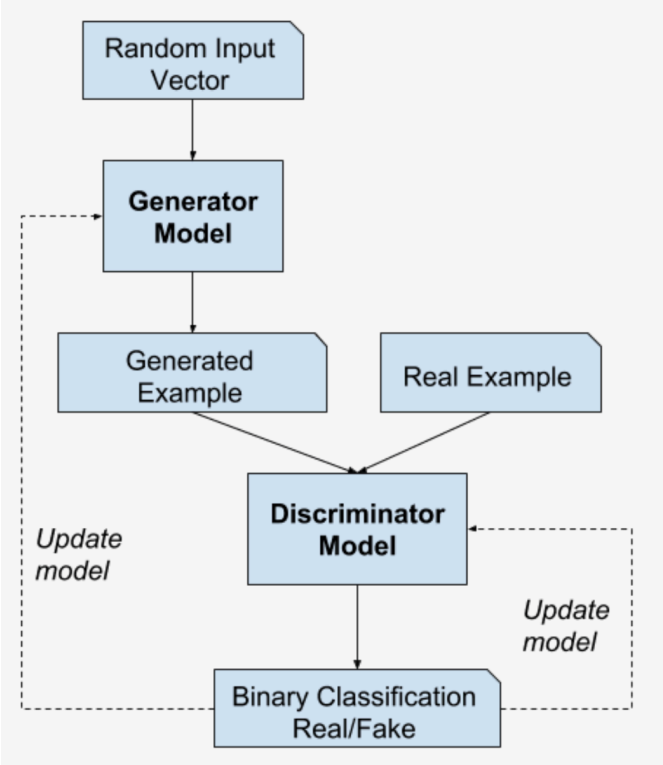
Discriminator Backpropagation

Once the discriminator loss is computed the gradients of the loss function with respect to the parameters of the discriminator network are calculated using backpropagation. This process involves propagating the error backwards through the network. After computing the gradients at each layer of the network. The parameters of the discriminator network are updated using the optimization algorithm SGD or ADAM.



Training

The generator and discriminator are trained together. The process involves feeding both the fake data produced by the generator and real samples from the domain set into the discriminator. If the discriminator successfully identifies real and fake samples it is rewarded with no changes to its model parameters. On the other hand if the generator fails to deceive the discriminator it gets penalized with updates to its model parameters. Similarly, when the generator deceives the discriminator, it gets rewarded with no change to its model parameters while being penalized for failure with updates to its model parameters. After many iterations of training, both the generator and discriminator may reach a state where neither can significantly improve their performance. This state is known as equilibrium state or convergence. At this point the generator ideally becomes good at producing synthetic data that resembles data from the domain set while the discriminator is unable to distinguish between real and fake samples.



Example of the Generative Adversarial Network Model Architecture

Loss Function

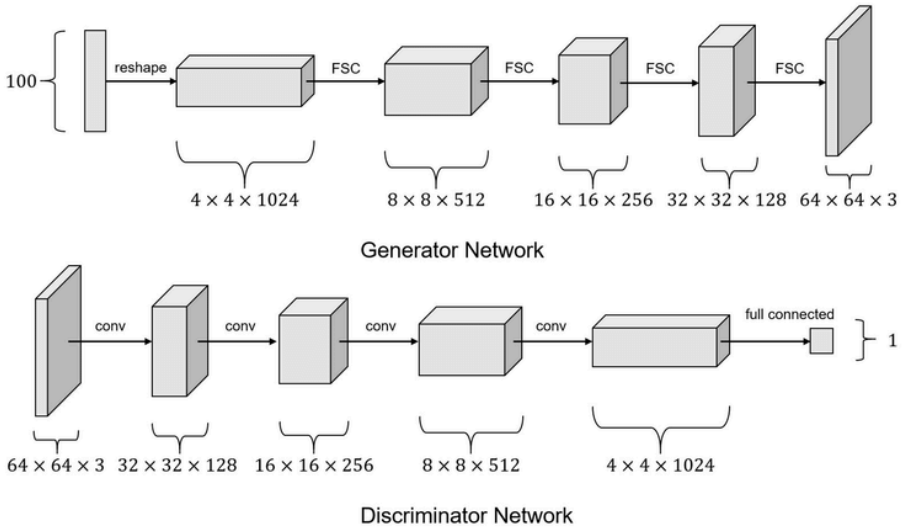
D and G play the following two-player minimax game with value function V(G,D):

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

- D(x) = probability that x came from the data rather than pg (generator's distribution).
- Train D to maximize the probability of assigning the correct label to both training examples and samples from G.
- Simultaneously train G to minimize log(1-D(G(Z))

DCGANS:

- Type of GAN which uses convolutional layers



- Generator and discriminator are CNN models
- Input noise is deconvoluted into an image, which is then classified
- **Data Augmentation:**
    - Expanding training set of other machine learning models such as CNN by generating similar data
    - Has been used to improve models where available training data is insufficient

Architecture: DCGAN integrates CNN structures to both generator and discriminator with some constraints. Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).

- Strided convolutions:
    - Reduces spatial dimensions of feature maps.
    - Allows network to its own spatial downsampling
- Fractional convolution:

- Similar to strided but UPSAMPLES.

The first layer takes uniform random values (Z) that are reshaped to a 4D tensor for the generator.

- Discriminator: Last layer is flattened and placed in sigmoid output indicating probability of real or fake.

Use batch normalization in both the generator and the discriminator for every unit (neuron) input.

- NOT applied G output layer and D input layer.
- Helps gradients flow (deep models) and prevent generator collapse.

Remove fully connected hidden layers for deeper architectures.

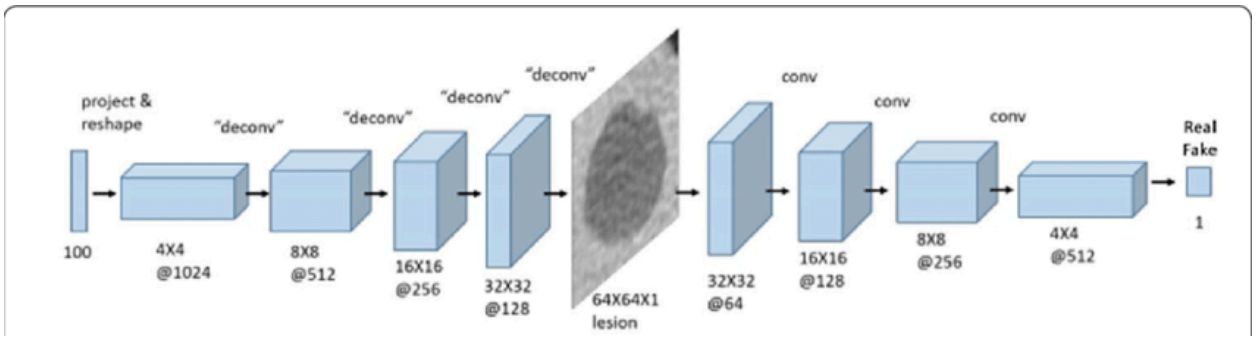
- Reduce parameters

  GAN Impact: Allows to better learn from spatial features.

Use ReLU activation in the generator for all layers except for the output, which uses Tanh.

Use LeakyReLU activation in the discriminator for all layers

- Regular GAN would use maxout activation



DCGAN Generator

```
def make_generator_model():
    generator = Sequential(name="Generator")
    generator.add(Dense(16*16*512,
```

```
                        input_shape=(100,)))
    generator.add(Reshape((16,16,512)))

    # layer 2
    generator.add(Conv2DTranspose(filters=256, kernel_size=(6,6),
strides=(2,2), padding='same'))
    generator.add(LeakyReLU(alpha=0.2))

    #layer 3
    generator.add(Conv2DTranspose(filters=128, kernel_size=(6,6),
strides=(2,2), padding='same'))
    generator.add(LeakyReLU(alpha=0.2))

    #layer 4
    generator.add(Conv2DTranspose(filters=64, kernel_size=(6,6), strides=(2,2),
padding='same'))
    generator.add(LeakyReLU(alpha=0.2))

    #layer 5
    generator.add(Conv2DTranspose(filters=32, kernel_size=(6,6), strides=(2,2),
padding='same'))
    generator.add(LeakyReLU(alpha=0.2))

    #output
    generator.add(Conv2DTranspose(3, kernel_size=(6,6), activation='tanh',
strides=(1,1), padding='same'))

    return generator
```

```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

DCGAN Discriminator

```
# Discriminator
def make_discriminator_model():

    discriminator = Sequential()

    #layer 1
    discriminator.add(Conv2D(filters=32, kernel_size=(6,6), strides=(2, 2),
padding='same', input_shape=[256, 256, 3]))
```

```
    discriminator.add(LeakyReLU(0.2))

    #layer 2
    discriminator.add(Conv2D(filters=64, kernel_size=(6,6), strides=(2, 2),
padding='same'))
    discriminator.add(BatchNormalization())
    discriminator.add(LeakyReLU(0.2))

    #layer 3
    discriminator.add(Conv2D(filters=128, kernel_size=(6,6), strides=(2, 2),
padding='same'))
    discriminator.add(BatchNormalization())
    discriminator.add(LeakyReLU(0.2))

    #layer 4
    discriminator.add(Conv2D(filters=256, kernel_size=(6,6), strides=(2, 2),
padding='same'))
    discriminator.add(BatchNormalization())
    discriminator.add(LeakyReLU(0.2))

    # Flatten and Output Layers
    discriminator.add(Flatten())
    discriminator.add(Dropout(0.3))
    discriminator.add(Dense(1, activation='sigmoid'))
    # discriminator.add(Dense(1, activation='binary_crossentropy'))

    return discriminator
```

```
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss
```

WGAN-GP

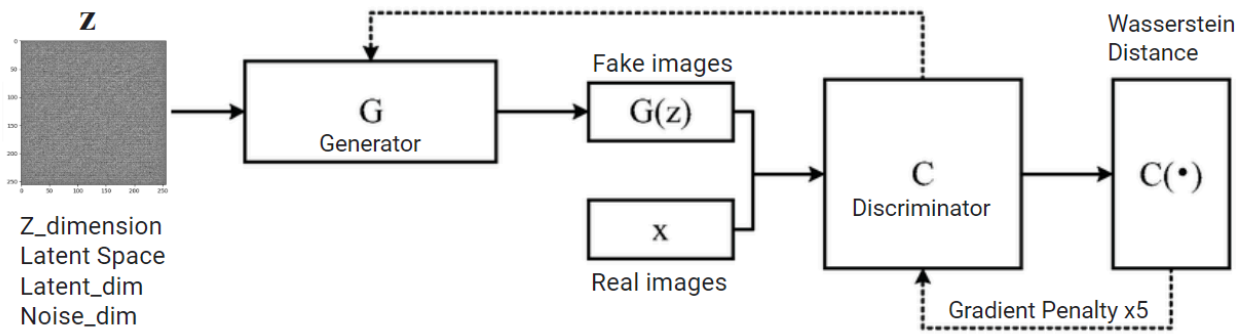
Before moving forward with WGAN-GP, we'd like to cite the source Gulrajani et al. (2017) who did very incredible work which helped us in our process. With WGAN-GP, a big difference is that WGAN-GP uses Wasserstein distance as its loss function instead of the Jensen-Shannon divergence (JS divergence) common to other GAN models. Wasserstein distance is the minimum

transport cost (also known as Earth Mover Distance). Rather than discriminating between real or fake images, WGANs "critique" by comparing to the training set directly. This helps avoid both the vanishing gradient problem and the exploding gradient problem.

Overview of Training:

We begin the training process by creating a tensor of random numbers with a certain batch size, such as 16, and a latent dimension of 100, which is essentially a noisy image that is passed to the generator G as input. The noisy tensor is processed through the generator architecture where the output layer is a 2D transposed convolutional layer with 128 input channels and 3 output channels. The noisy tensor is now a 16x3x64x64 tensor G(z) which means the batch size of 16 and 3x64x64 nested tensors representing pixel values of the images.

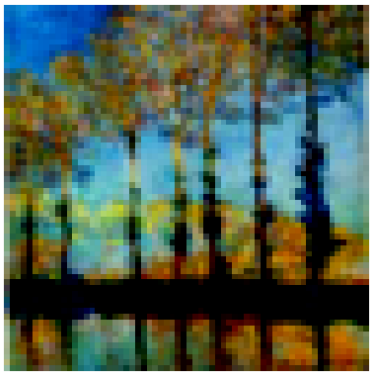
The discriminator uses both the fake images G(z) and the real images x as inputs to "score" how realistic the images look. In this process, we train the discriminator 5 times before the generator to make up for the slowed down training process caused by the penalty. The gradient penalty is important because it ensures that the discriminator's gradients do not change too rapidly or too slowly. We then use the "scores" of G(z) and x, lambda gradient penalty which is 10, and the gradient penalty for the discriminator's loss function. Now that the discriminator has updated its weights, we move on to training the generator and updating the generator's weights.



WGAN-GP Training Overview

Gradient Penalty

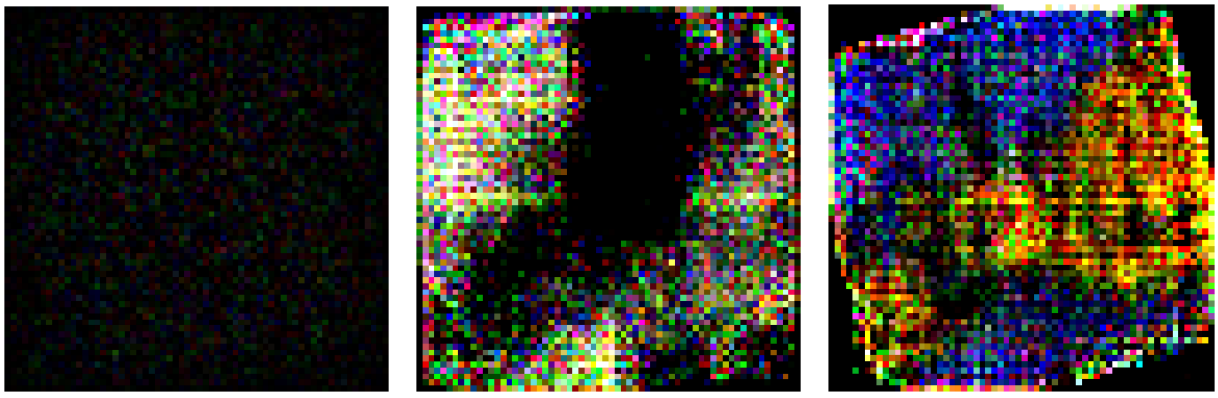
Another big difference was implementing the gradient penalty. The inputs for the gradient penalty consist of the discriminator, real images, and fake images. We create interpolated images by overlapping the real and fake images using certain percentages for each (i.e. 92.82% of real image + 07.18% of fake image).

	
Interpolated image: 92.82% real, 07.18% fake	Interpolated image: 13.13% real, 86.87% fake

Visual examples of some Interpolated images

The percentages are calculated by using PyTorch's rand method which returns a tensor filled with random numbers from a uniform distribution. We take those interpolated images and pass them into the discriminator to obtain mixed scores since the interpolated images are mixed. Now

with both interpolated images and mixed scores, we can obtain the gradients of the output (mixed scores) with respect to the input (interpolated images).



Visual examples of the gradients of interpolated images and mixed scores before 25 epochs
Note: These had the random rotate and random erase data augmentation

The gradients are now flattened to a tensor size of 16x12288. The penalty term is calculated as the squared difference from the norm of 1 for these gradients.

WGAN-GP Generator

Within the Generator architecture, all of the layers are 2D transposed convolutional layers with different input sizes, output sizes, and activation functions. In the hyperparameters, we set "generator_features" to 64. The input layer takes in a latent dimension of 100 and passes that onto the first hidden layer, which consists of 1024 nodes, kernel size of (4,4), stride of (2,2), padding of (1,1), and ReLU activation function. The rest of the hidden layers have similar kernels, strides, and paddings but different input and outputs such as the second hidden layer, which has 512 nodes followed by the third hidden layer has 256 nodes. The output layer takes in a tensor of size 128 and outputs a 3x64x64 tensor that represents pixel values.

```
class Generator(nn.Module):
```

```
  def __init__(self, latent_dim, num_of_channels, generator_features):
    super(Generator, self).__init__()
    self.generator = nn.Sequential(

      #  Input Layer
      nn.Sequential(
        nn.ConvTranspose2d(latent_dim, generator_features * 16, kernel_size=4,
stride=1, padding=0, bias=False,),
        nn.BatchNorm2d(generator_features * 16),
        nn.ReLU(),
      ),

      #  Second Layer
      nn.Sequential(
        nn.ConvTranspose2d(generator_features * 16, generator_features * 8,
kernel_size=4, stride=2, padding=1, bias=False,),
        nn.BatchNorm2d(generator_features * 8),
        nn.ReLU(),
      ),

      #  Third Layer
      nn.Sequential(
        nn.ConvTranspose2d(generator_features * 8, generator_features * 4,
kernel_size=4, stride=2, padding=1, bias=False,),
        nn.BatchNorm2d(generator_features * 4),
        nn.ReLU(),
      ),

      #  Fourth Layer
      nn.Sequential(
        nn.ConvTranspose2d(generator_features * 4, generator_features * 2,
kernel_size=4, stride=2, padding=1, bias=False,),
        nn.BatchNorm2d(generator_features * 2),
        nn.ReLU(),
      ),

      #  Output Layer
      nn.ConvTranspose2d(generator_features * 2, num_of_channels,
kernel_size=4, stride=2, padding=1),
      nn.Tanh(),
  )

  def forward(self, x):
    return self.generator(x)
```

WGAN-GP Discriminator

Within the discriminator architecture, all of the layers are 2D convolutional layers with different input sizes, output sizes, and activation functions. In the hyperparameters, we set "discriminator_features" to 64. The input layer takes in a number of channels of 3 and passes that onto the first hidden layer, which consists of 64 nodes, kernel size of (4,4), stride of (2,2), padding of (1,1), and the LeakyReLU activation function. The rest of the hidden layers have similar kernels, strides, and paddings but different input and outputs such as the second hidden layer which has 128 nodes followed by the third hidden layer has 256 nodes. The output layer takes in a tensor of size 512, uses 0 padding, and outputs a value that represents how real the image appears.

```
class Discriminator(nn.Module):

  def __init__(self, num_of_channels, discriminator_features):
    super(Discriminator, self).__init__()
    self.discriminator = nn.Sequential(

      # Input Layer
      nn.Conv2d(num_of_channels, discriminator_features, kernel_size = 4,
stride = 2, padding = 1),
      nn.LeakyReLU(0.2),

      # Second Layer
      nn.Sequential(
        nn.Conv2d(discriminator_features, discriminator_features * 2,
kernel_size=4, stride=2, padding=1, bias=False,),
        nn.InstanceNorm2d(discriminator_features * 2, affine=True),
        nn.LeakyReLU(0.2),
      ),

      # Third Layer
      nn.Sequential(
        nn.Conv2d(discriminator_features * 2, discriminator_features * 4,
kernel_size=4, stride=2, padding=1, bias=False,),
        nn.InstanceNorm2d(discriminator_features * 4, affine=True),
        nn.LeakyReLU(0.2),
```

```
      ),

      # Fourth Layer
      nn.Sequential(
        nn.Conv2d(discriminator_features * 4, discriminator_features * 8,
kernel_size=4, stride=2, padding=1, bias=False,),
        nn.InstanceNorm2d(discriminator_features * 8, affine=True),
        nn.LeakyReLU(0.2),
      ),

      # Output Layer is 4x4 (Conv2d turns into 1x1)
      nn.Conv2d(discriminator_features * 8, 1, kernel_size=4, stride=2,
padding=0)
    )

  def forward(self, x):
    return self.discriminator(x)
```

Diffusion Model:

A class of generative models, diffusion models both take their name and are inspired by the concept of diffusion in physics: the process by which particles spread from areas of high concentration to low concentration. They consist of two main processes: a forward diffusion process and a reverse diffusion process. In the forward diffusion process, the structure of an image's pixel value distribution is slowly broken down by iteratively adding noise to the image. In the reverse diffusion process, neural networks are trained to iteratively recover less noisy versions of an image. The end result is a model that will have learned to take noise and generate recognizable images from a desired distribution.

Diffusion Original (BASE)

Inspiration:

This model was inspired by an implementation of the diffusion model from https://www.youtube.com/watch?v=a4Yfz2FxXiY&t=995s . Where their main goal was to generate images of random cars given the StanfordCars Dataset. We utilized their model as a baseline for the performance of our implementation of the diffusion model. The model utilizes the U-Net model and a Positional Encoder to generate images of cars. However, for our work we transformed and changed some of the structure of the authors diffusion model to create our diffusion model.

Forward Diffusion Process:

In the forward diffusion process, the structure of an image is broken down into noise. This is done by iteratively adding noise to the image. This can be modeled as a Markov chain of T steps. At every step t of the Markov chain, Gaussian noise with with variance β_t is added to x_{t-1} to produce x_t. This can be represented by the following formula:

$$q(x_t|x_{t-1}) = N(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I)$$

Where:

q(x) = real data distribution

x_i = image at time i

I = identity matrix

$\beta_t \in (0,1)$

By using parameterization, it can be shown that

$$x_t = \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}\epsilon_{t-1}$$
$$= \sqrt{\underline{\alpha}_t}x_0 + \sqrt{1-\underline{\alpha}_t}\epsilon_o$$

Where:

$\alpha_t = 1 - \beta_t$

$\underline{\alpha}_t = \prod_{s=0}^{t} \alpha_s$

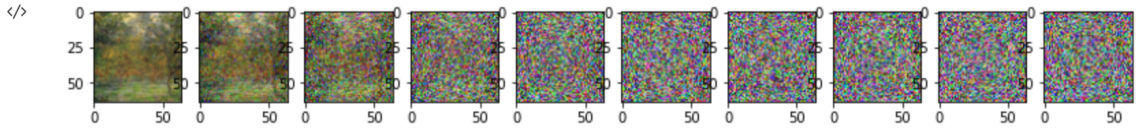
$\epsilon_0 \sim N(0,1)$

Because β_t is a hyperparameter, $\underline{\alpha}_t$ can be computed for any timestep. This allows x_t to be computed for any timestep. Moreover, β_t, which determines how much noise is added at step t, can be set to a constant value or set according to a function over the number of timesteps. Early diffusion models made use of a linear schedule, that is they used a value of β_t that was increased by a constant amount at every step. Later research suggested that using a cosine schedule, that is scaling the value of β_t according to the cosine function, resulted in better performance in model training. A comparison of the forward diffusion process between linear scheduling and cosine scheduling can be seen in the following figure:

Linear scheduling (top) and cosine scheduling(bottom). Source: Nichol & Dhariwal 2021

It can be seen that using linear scheduling results in the image becoming noise more quickly and steadily than with cosine scheduling.

Linear Scheduling:



Reverse Diffusion Process:

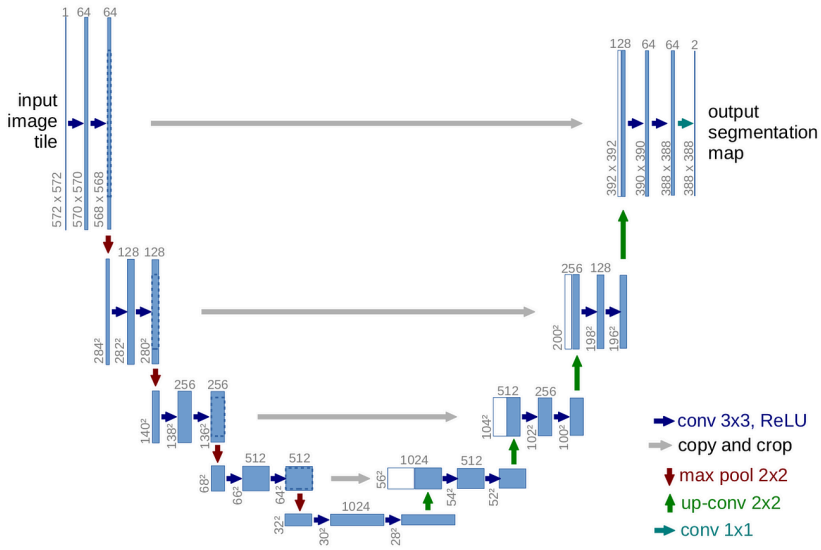
In the reverse diffusion process, noise is given structure creating a recognizable image. Similar to the forward diffusion process, this is done iteratively. At every step, we take in some image and try to recover the less noisy version of the image. That is, the goal of the model is to learn the reverse distribution $q(x_t|x_{t-1})$. Or given x_t, the goal is to determine x_{t-1}. In practice, $q(x_t|x_{t-1})$ is approximated using a parameterized model

$$p_\theta(x_{t-1}|x_t) = N(x_t, \mu_\theta(x_t, t), \sum_\theta(x_t, t))$$

By conditioning on the time step t, a neural network can be trained to predict the parameters $\mu_\theta(x_t, t)$ and $\sum_\theta(x_t, t)$ for each value of t. The end result of training would be a model that can accurately approximate $q(x_t|x_{t-1})$. This would allow for a sample x_T to be taken from $N(0, I)$, the reverse diffusion process to be run, and a sample from $q(x_0)$ to be acquired. This means the model can be given an image consisting of gaussian noise and it could generate a new data point, or image, from the original distribution.

U-NET:

When considering the model architecture, it should be noted that the model's input and output should be the same size. For this reason, a U-Net is commonly used. A U-Net is a type of convolutional neural network architecture commonly used in image processing. Its name comes from its U-shaped architecture which consists of a contracting path followed by an expanding path. The contracting path is an encoder that progressively down samples the input image. The expanding path is a decoder that progressively upsamples the feature maps to the original input size. Moreover, U-nets include skip connections which connect corresponding layers between the encoder and decoder blocks. This allows the network to retain spatial information during the upsampling process.



The U-Net Architecture. Source: Ronneberger et al.

Each block in the simple U-net is defined as:

```
class Block(nn.Module):

  def __init__(self, in_ch, out_ch, time_emb_dim, up=False):
    super().__init__()
    self.linear =  nn.Linear(time_emb_dim, out_ch)
    if up:
```

```
            self.conv1 = nn.Conv2d(2*in_ch, out_ch, 3, padding=1)
            self.transform = nn.ConvTranspose2d(out_ch, out_ch, 4, 2, 1)
        else:
            self.conv1 = nn.Conv2d(in_ch, out_ch, 3, padding=1)
            self.transform = nn.Conv2d(out_ch, out_ch, 4, 2, 1)
        self.conv2 = nn.Conv2d(out_ch, out_ch, 3, padding=1)
        self.bnorm1 = nn.BatchNorm2d(out_ch)
        self.bnorm2 = nn.BatchNorm2d(out_ch)
        self.relu  = nn.ReLU()
```

Every block starts with a linear layer whose input is the time embedding dimension which is defaulted to 32 and output channels which are changed for each block call. When an upsampling is called, typically the in_ch is larger than the out_ch because we want to decrease the depth of the feature maps. Conversely, when a downsampling is called the output channel dimension would be higher than the input channel as we are trying to extract features from the input. Upsampling layers are used to increase the spatial dimensions of the feature maps while reducing their depth while the downsampling layers are used to reduce the spatial dimensions of the feature maps while increasing their depth. An example of downsampling would be increasing the depth from 128 to 256 in self.down2 = Block(128,256,32).

```
class SimpleUnet(nn.Module):
  def __init__(self):
      super().__init__()
      image_channels = 3
      down_channels = (64, 128, 256, 512, 1024)
      up_channels = (1024, 512, 256, 128, 64)
      out_dim = 3
```

```
    time_emb_dim = 32

    self.time_mlp = nn.Sequential(
            SinusoidalPositionEmbeddings(time_emb_dim),
            nn.Linear(time_emb_dim, time_emb_dim),
            nn.ReLU()
        )

    self.conv0 = nn.Conv2d(image_channels, down_channels[0], 3, padding=1)

    self.downs = nn.ModuleList([Block(down_channels[i], down_channels[i+1], \
                            time_emb_dim) \
                for i in range(len(down_channels)-1)])
    self.ups = nn.ModuleList([Block(up_channels[i], up_channels[i+1], \
                                time_emb_dim, up=True) \
                for i in range(len(up_channels)-1)])
    self.output = nn.Conv2d(up_channels[-1], out_dim, 1)
```

Forward Process:

$$q(x_t|x_0) = N\left(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I\right)$$

```
# Values for closed form
betas = linear_beta_schedule(timesteps=T)
```

```
alphas = 1. - betas
alphas_cumprod = torch.cumprod(alphas, axis=0)
alphas_cumprod_prev = F.pad(alphas_cumprod[:-1], (1, 0), value=1.0)
sqrt_recip_alphas = torch.sqrt(1.0 / alphas)
sqrt_alphas_cumprod = torch.sqrt(alphas_cumprod)
sqrt_one_minus_alphas_cumprod = torch.sqrt(1. - alphas_cumprod)
posterior_variance = betas * (1. - alphas_cumprod_prev) / (1. - alphas_cumprod)
```

These variables are the most important part of the diffusion model. They are the variables that make up the closed form of the forward diffusion process. By defining all these variables in terms of their mathematical terms, by utilizing the equation that the DDPM paper gives, we are able to calculate the amount of noise that will be added to the image. Note that these terms are used universally for making diffusion models, and any state of the art Diffusion model would define these terms to create their diffusion models.

Pre Processing Data for Diffusion Model:

For training the Diffusion model we had to make a MonetDataset class which extends the VisionDataset, this will allow us to train the Diffusion much more easily by utilizing the DataLoader which interacts well with children that inherit the VisionDataset class. By creating the MonetDataset we are also able to easily perform operations on the dataset with some parameters. Torchvision provides us the transform function which allows us to scale, augment, and tensor-fy our images.

Loss Function:

The loss function is the F1 which stands for the Mean Absolute Error: The distance between the actual noise of the image and the predicted noise of the image. The U-Net model along with the

positional encoder inside the U-net model are responsible for predicting the amount of noise the input image has. The image's noise is calculated using the formula to calculate the forward diffusion process. We then compare them using the F1 function to calculate the loss to make the model learn from the loss between these two noises .To get a denoised image, we simply subtract the noise that we get from the model and then subtract that noise from the original input image.

Diffusion From Library

```
from denoising_diffusion_pytorch import Unet, GaussianDiffusion, Trainer

model = Unet(
  dim = 64,
  dim_mults = (1, 2, 4, 8),
  flash_attn = True
)

diffusion = GaussianDiffusion(
  model,
  image_size = 128,
  timesteps = 1000,          # number of steps
  sampling_timesteps = 250
)

trainer = Trainer(
  diffusion,
  'userpath',
  train_batch_size = 32,
```

```
  train_lr = 8e-5,
  train_num_steps = 1000,
  gradient_accumulate_every = 2,
  ema_decay = 0.995,
  amp = True,
  calculate_fid = True
)

trainer.train()
```

The denoising-diffusion-pytorch package allows you to train a diffusion model on a specific dataset. First we must import the necessary packages which are torch, Unet, GaussianDiffusion, and the Trainer.

```
import torch
from denoising_diffusion_pytorch import Unet, GaussianDiffusion
```

We then have to define the network architecture aka the U-Net. The dim parameter specifies the number of feature maps before the first down-sampling, and the dim_mults parameter provides multiplicands for this value and successive down-samplings.

```
model = Unet(
      dim = 64,
      dim_mults = (1, 2, 4, 8)
)
```

Once the network architecture is defined, we must define the Diffusion Model itself. We pass in the U-Net model that we defined along with several parameters including the size of the images

to generate, the number of timesteps in the diffusion process, and the number of sampling timesteps.

```
diffusion = GaussianDiffusion(
      model,
      image_size = 128,
      timesteps = 1000,
       sampling_timesteps = 250
)
```

Then we can train this to a specific dataset with a path to our selected dataset directory path and pass that in the Trainer() object. We then change the image_size to the appropriate value, specify our training batch size, the training learning rate, total training steps, the gradient accumulation steps, the exponential moving average decay, whether to turn on mixed precision, and whether to calculate Frechet inception distance (FID) during training. After this, then we can simply run the code to train the model. One thing to note is that PyTorch must be compiled with CUDA enabled in order to use this class so we had to install CUDA as well.

```
trainer = Trainer(
    diffusion,
    'path/to/your/images',
    train_batch_size = 32,
    train_lr = 2e-5,
    train_num_steps = 700000,         # total training steps
    gradient_accumulate_every = 2,    # gradient accumulation steps
    ema_decay = 0.995,                # exponential moving average
decay
    amp = True                        # turn on mixed precision
)

trainer.train()
```

Diffusion With Improved U-Net AND Cosine Scheduling

```
class SimpleUnet(nn.Module):

  def __init__(self):
      super().__init__()
      image_channels = 3
      down_channels = (64)
      up_channels = (64)
      out_dim = 3
      time_emb_dim = 32

      # Time embedding
      self.time_mlp = nn.Sequential(
            SinusoidalPositionEmbeddings(time_emb_dim),
            nn.Linear(time_emb_dim, time_emb_dim),
            nn.ReLU()
          )

      # Initial projection
      self.conv0 = nn.Conv2d(image_channels, down_channels[0], 3, padding=1)

      self.down1 = Block(64,128,32)
      self.sa1 = SelfAttention(128,32)
      self.down2 = Block(128,256,32)
      self.sa2 = SelfAttention(256,16)
      self.down3 = Block(256,512,32)
      self.sa3 = SelfAttention(512,8)
      self.down4 = Block(512,1024,32)
      self.sa4 = SelfAttention(1024,4)

      self.up1 = Block(1024,512,32,up=True)
      self.sa5 = SelfAttention(512,8)
      self.up2 = Block(512,256,32,up=True)
      self.sa6 = SelfAttention(256,16)
      self.up3 = Block(256,128,32,up=True)
      self.sa7 = SelfAttention(128,32)
```

```
self.up4 = Block(128,64,32,up=True)
self.sa8 = SelfAttention(64,64)
self.output = nn.Conv2d(up_channels[-1], out_dim, 1)
```

Cosine Scheduling:

```
def cosine_schedule(num_timesteps, s=0.008):
 def f(t):
  return torch.cos((t / num_timesteps + s) / (1 + s) * 0.5 * torch.pi) ** 2
 x = torch.linspace(0, num_timesteps, num_timesteps + 1)
 alphas_cumprod = f(x) / f(torch.tensor([0]))
 betas = 1 - alphas_cumprod[1:] / alphas_cumprod[:-1]
 betas = torch.clip(betas, 0.0001, 0.999)
 return betas
```

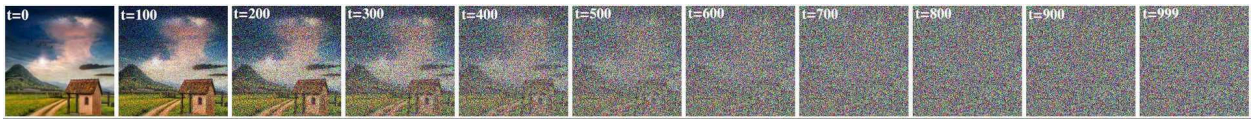
$$\bar{\alpha}_t = \frac{f(t)}{f(0)} \; where \; f(t) = cos(\frac{t/T + s}{1 + s} * \frac{\pi}{2})^2$$

Cosine scheduling is a popular scheduler that is used widely, the implementation of the cosine scheduler is used widely for diffusion models. This was invented by the same authors that created the Denoising Diffusion Probabilistic Models paper where they have stated that the cosine scheduler was superior to the linear scheduler. The following cosine function can be found as a default when using a cosine scheduler for a diffusion model. The main purpose of the cosine scheduler is to make sure not too much noise gets applied to the image in the earlier timesteps . In linear scheduling, the information may sometimes get destroyed too fast which

may result in redundant noise in the later timesteps. The cosine scheduler solves that by slowing down the rate of applying the noise while keeping a good amount of noise at the end of the last timestep.

EXAMPLE:

LINEAR BETA SCHEDULING:



COSINE BETA SCHEDULING:

As you can see, for the forward process, the image slowly transitions into having more noise as opposed to the linear time scheduling. We have trained 2 diffusion models, one that had used linear scheduling and one that had used cosine scheduling. On paper, the cosine scheduling should give better results, however, the linear time scheduling resulted in better results.

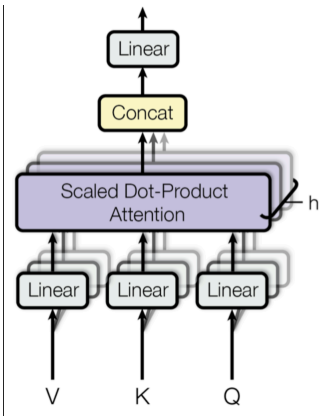
Improved U-NET Model

Self Attention:

```
class SelfAttention(nn.Module):
  def __init__(self, channels, size):
    super(SelfAttention, self).__init__()
    self.channels = channels
    self.size = size
    self.mha = nn.MultiheadAttention(channels, 4, batch_first=True)
```

```
        self.ln = nn.LayerNorm([channels])
        self.ff_self = nn.Sequential(
            nn.LayerNorm([channels]),
            nn.Linear(channels, channels),
            nn.GELU(),
            nn.Linear(channels, channels),
        )
```

The self attention layers consist of the multihead attention mechanism from the transformers, defined as nn.MultiheadAttention. Multihead attention performs the attention mechanisms multiple times concurrently. This would allow the U-net Model to learn the patterns of the features the convolutional layers provided in a more efficient way. Multihead attention serves to find the direct relationship between all of the features of the convolutional layers simultaneously.



Diffusion With Self Attention U-Net AND Linear Scheduling

Instead of using Cosine scheduling with the self attention U-net our group decided to see how the diffusion model with linear scheduling. Linear scheduling was more straightforward to implement than the cosine beta scheduling, which is why we want to see if it could perform

better due to the possible error inside the cosine beta scheduler. This model remains the same as the previous model with the only difference being its scheduler.

Results and Analysis

CUDA VS CPU

Training diffusion models and GAN models are very computationally expensive. When solely training the models on CPU, training for 100 epochs resulted in a 2 hour waiting period. However, by using the CUDA (Compute Unified Device Architecture) which is a software framework that allows the developers to access the power of NVIDIA GPUs. Where training with the GPU for 100 epochs only took 5 minutes. The difference in time between using a GPU and CPU was incredible. GPU's are extremely fast at processing similar tasks in parallel, which is why many AI companies such as META or OpenAI use thousands of GPU units to train their models.

GANs

DCGANs:

For DCGANs, we used Google Colab to train the models where we implemented TensorFlows libraries to build the generator and the discriminator. We were using the free version of Colab so

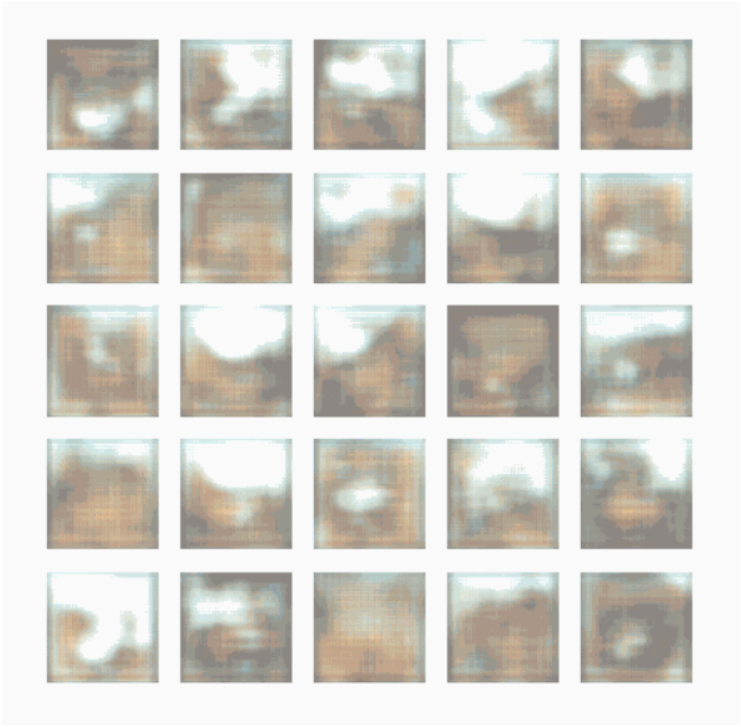
we'd occasionally get timed out and have to wait a couple of hours to connect to an accelerator. A work-around we implemented was to use different Google accounts and alternate between them after being timed out.

First attempts with DCGANs began with figuring out how to prepare the dataset of images with DCGANs training for 50 epochs. We kept trying different methods for processing the image data as well as looking at different tutorials by PyTorch and TensorFlow to understand more about different approaches and to see what could work.

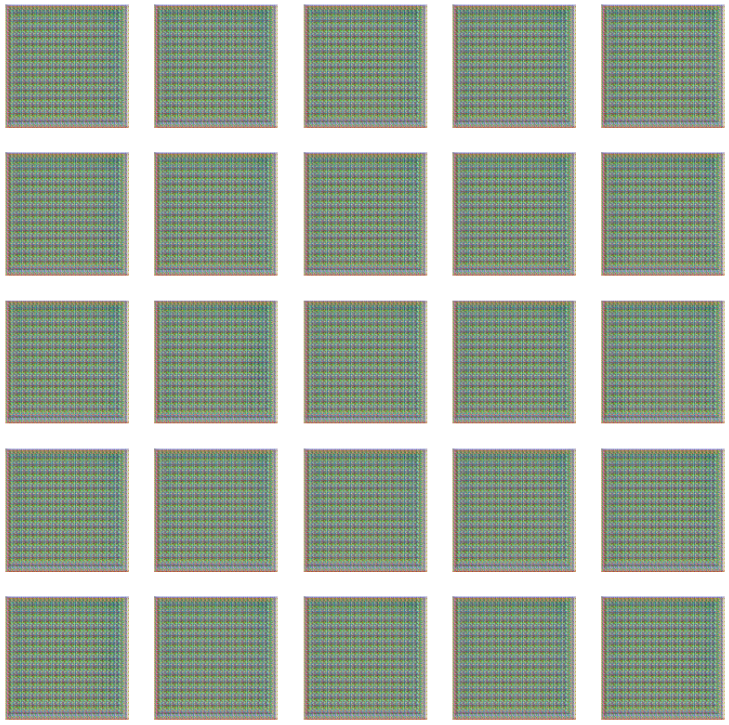
Details: - Type: DCGAN - 1st attempt - Epochs: 50 - Latent Dimension: 100 - First Hidden Layer of Generator: 512 - Augmentation: None - Original Dataset: 300	

The second attempt with DCGAN was from a different tutorial implemented a different way of displaying the results which helped us understand more about what is going on with the images

being generated. We increased the number of epochs to 600 to see if the generator improves over time.

Details: - Type: DCGAN - 2nd attempt - Epochs: 300 - Latent Dimension: 100 - First Layer of Generator: 512 - Augmentation: None - Original Dataset: 300	

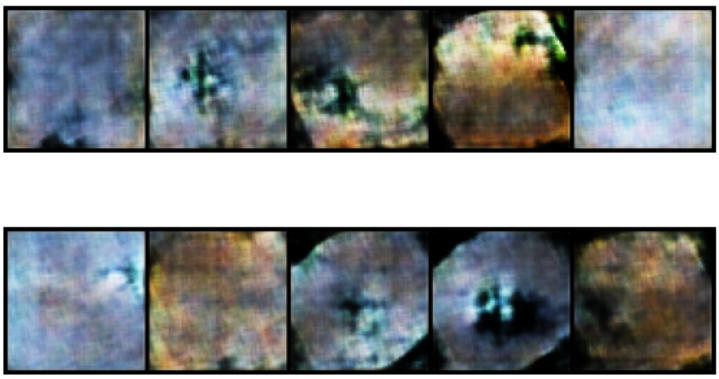
This is more progress than before, but the results were mixed because it was random shapes and colors. We tried retraining different times to see if any changes occurred but we continued to get similar results. After multiple attempts, we noticed recurring patterns especially around the 100-200 epoch range where the generator begins the early stages of mode collapse.

Details: - Type: DCGAN - 3rd attempt - Epochs: 250 - Latent Dimension: 100 - First Layer of Generator: 512 - Augmentation: None - Original Dataset: 300	

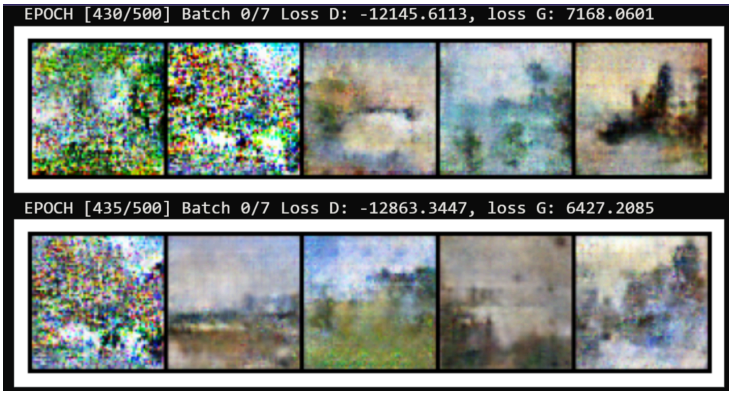
We noticed that the generator produced images that consisted of green grids after enough epochs and never improved after that. At this point, we agreed that this was our models experiencing mode collapse. Our next step was to look at research how to avoid mode collapse.

WGAN-GP:

We looked into what can be done to avoid mode collapse and found many ideas. We tried making the DCGAN models less complex and more complex, but those yielded similar results. Different types of image transformations were attempted including random horizontal flip, random erasing, resizing with larger numbers. Those transformations directly impacted generated images as sometimes the results had random black spots from the random erased spots or slightly flipped images which was something we did not want to proceed with.

Details:
- Type: WGAN-GP
- Epochs 30/50
- Augmentation:
  - Five crop
  - Random erase
  - Random rotate
- Full Dataset: 1800
  - Original (300) + Augmented (1500)
	

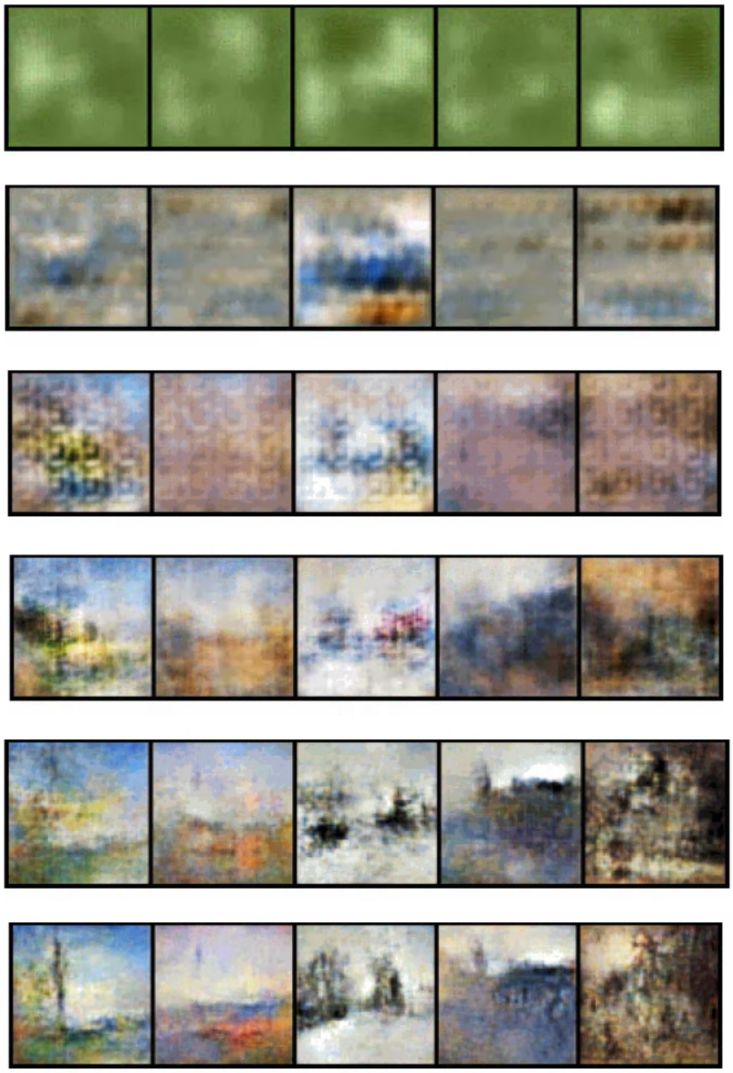
We found that what helps mode collapse the most was to use WGAN-GP along with mixed results from the five crop transformations. We found a paper that introduced WGAN-GP, Gulrajani et al. (2017), which helped us learn more about the discriminators gradients experiencing exploding and vanishing gradients.

Details:
- Type: WGAN-GP
  - 1st attempt
- Epochs [430/500]
- Latent Dimension: 125
- First Layer of Generator: 1536
- Augmentation: Five crop
- Full Dataset: 1800
  - Original (300) + Augmented (1500)
	

This was better progress as we were able to make it to 430 epochs without mode collapse and some of the images look like monet paintings that had very bright and vibrant colors. However, some of the images generated still had poor results such as grainy static colors.

Details:

- Type: WGAN-GP
  - 2nd attempt
- Epochs: 300
- Latent Dimension: 100
- First Layer of Generator: 1024
- Augmentation: None
- Original Dataset: 300



Generated same images with the same noise (seed) over time

Generated images with different random noise inputs

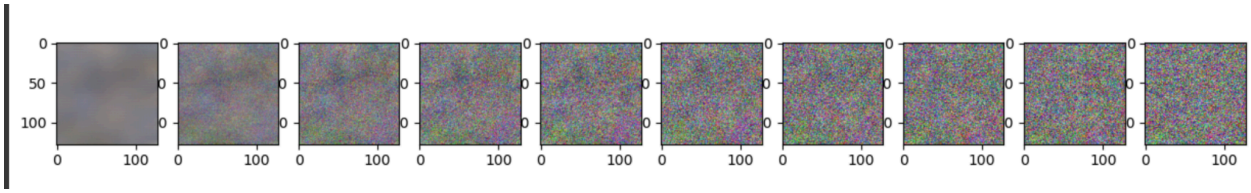
We decided to use similar numbers as Gulrajani et al. (2017) which include latent dimension as 100 and the first hidden layer of generator as 1024 instead of 1536 in our first attempt. We also decided to not use augmented images because the training process was taking longer and the results were not improving.

Diffusion Model Reg U-Net

The Diffusion model, when coupled with the standard U-net and linear scheduling, exhibited the poorest performance among all non-Library models assessed. Our training regimen for this model was limited to 300 epochs, as the loss function plateaued without further improvement, and the images generated during training bore no resemblance to Monet's paintings whatsoever. Despite its underwhelming performance, this model serves as a foundational baseline against which we can measure and potentially surpass the results of other models. Our aim is to enhance the performance of subsequent models to outperform this initial benchmark.

IMG_SIZE = 64, BATCH_SIZE = 128, EPOCH = 300

Training Image at 300 Epochs:



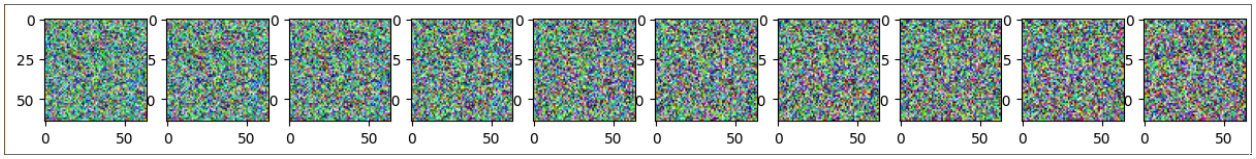
Diffusion Model Improved U-Net With Cosine Scheduling

Despite its theoretical superiority over linear scheduling, the diffusion model incorporating attention alongside cosine beta scheduling failed to match the performance of its counterpart utilizing attention alongside linear beta scheduling. Nevertheless, this variant notably outperformed the diffusion model lacking attention paired with linear beta scheduling. Our

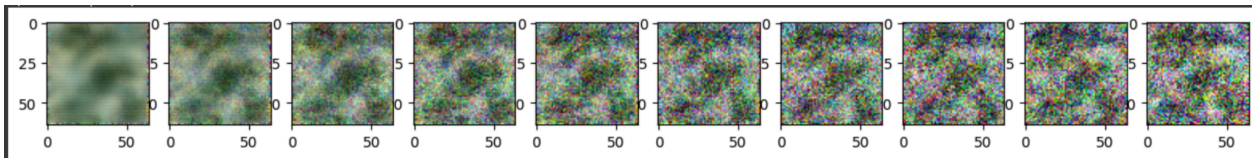
training regimen extended over 2000 epochs, during which the model exhibited images reminiscent of Monet's paintings in the training phase. However, upon attempting to generate images in practice, only a subset bore resemblance to paintings.

IMG_SIZE = 64, BATCH_SIZE = 128, EPOCH = 2000

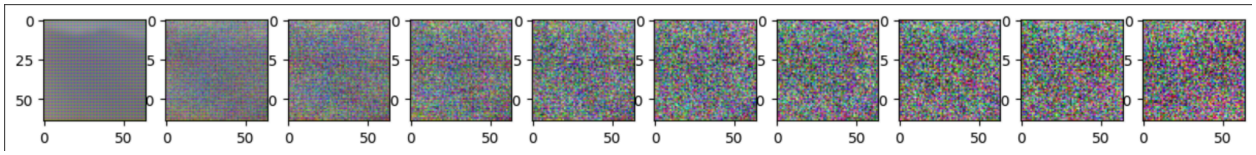
0 Epoch samples:



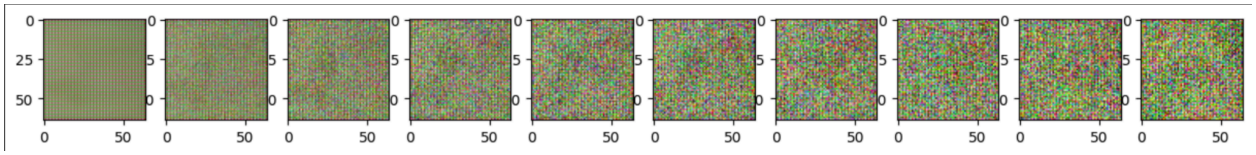
10 Epoch samples:



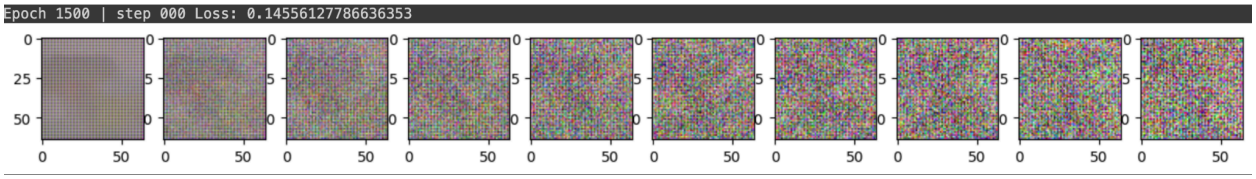
500 Epoch samples:



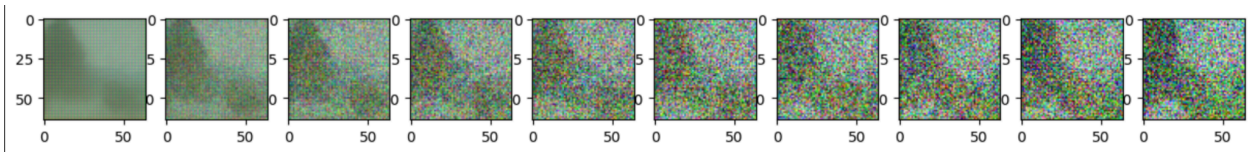
1000 Epoch Samples:



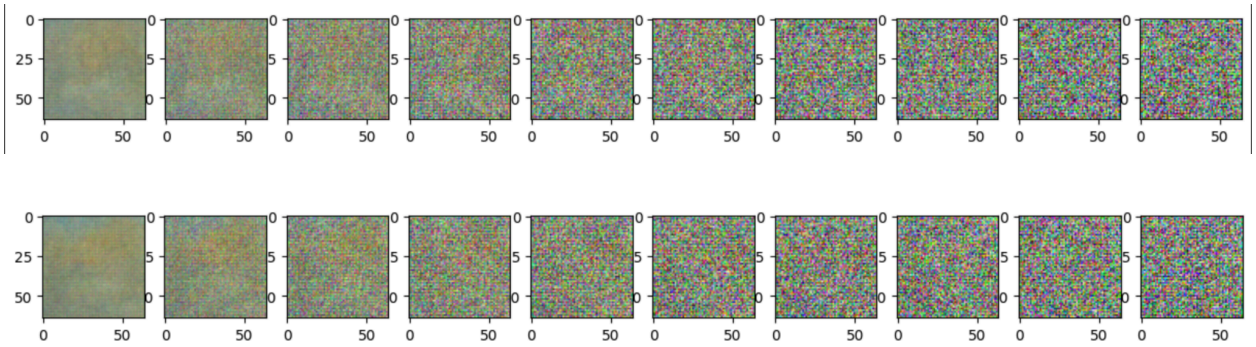
1500 Epoch Samples:



2000 Epoch Samples:



Results:



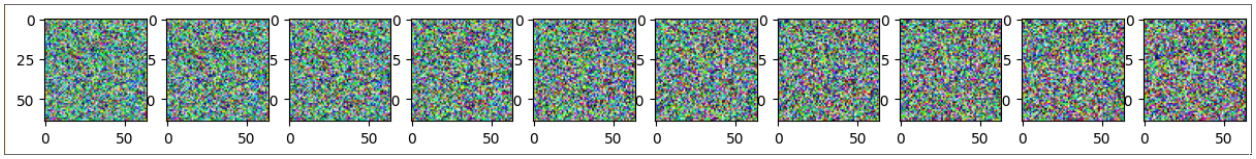
Diffusion Model Improved U-Net With Linear Scheduling

Following an extensive training session spanning 2500 epochs, the diffusion model began to exhibit promising advancements. Unlike earlier iterations that produced noisy images, the diffusion model, when integrated with self-attention and linear scheduling, crafted an image evocative of a castle. This discovery proved profoundly unexpected, particularly in light of our

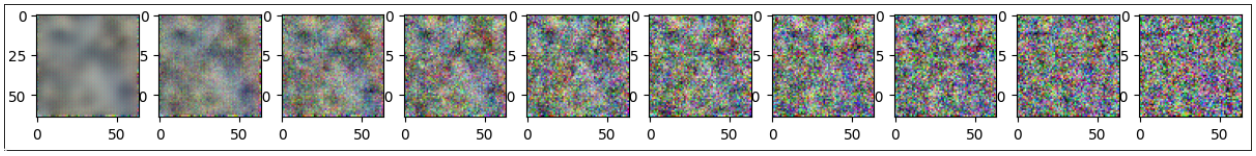
anticipation that the cosine scheduler, as posited in the "Improved Denoising Diffusion Probabilistic Models" paper, would yield superior outcomes. Notably, around the 2000-epoch mark, a remarkable development unfolded: the loss function dipped below 0.10, a milestone unprecedented in preceding diffusion models. Subsequent to an additional 500 epochs of training, discernible enhancements in image clarity emerged. Upon the model's completion of training, an unprecedented revelation awaited us: the generation of images devoid of noise, exemplified by the depiction of a castle in Figure 2.

IMG_SIZE = 64, BATCH_SIZE = 128, EPOCH = 2000

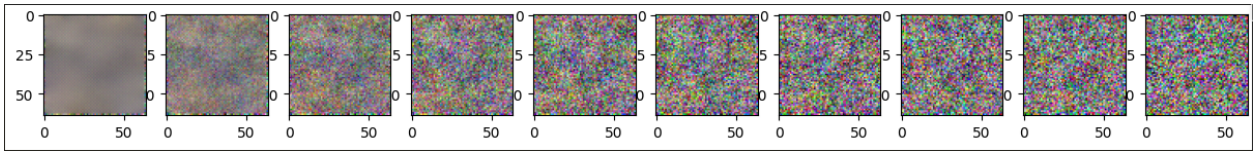
0 Epoch samples:



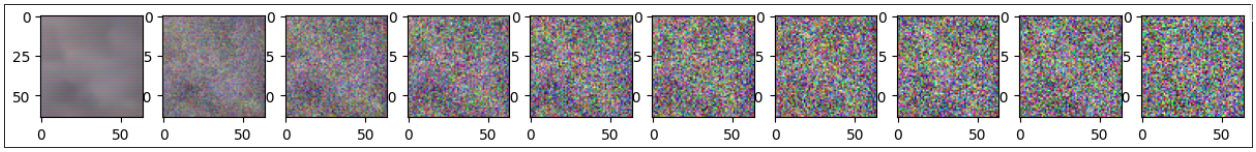
10 Epoch samples:



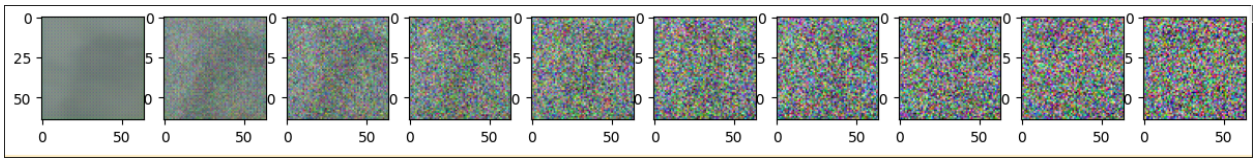
25 Epoch samples:



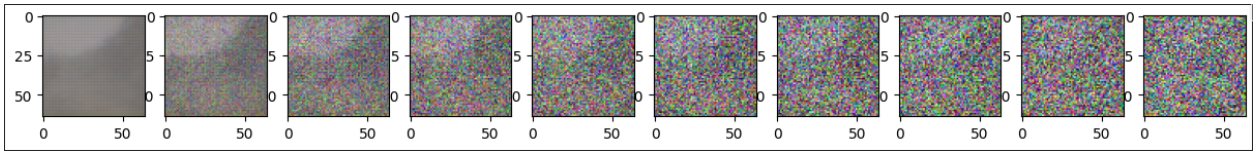
500 Epoch Samples:



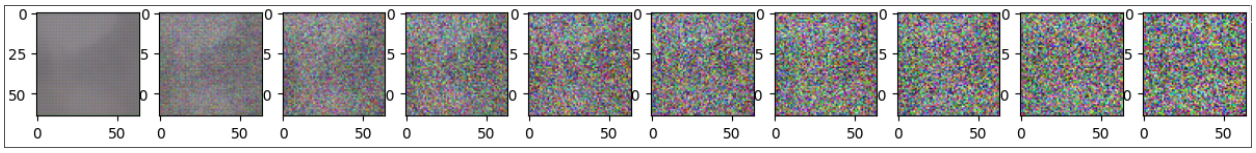
1500 Epoch Samples:



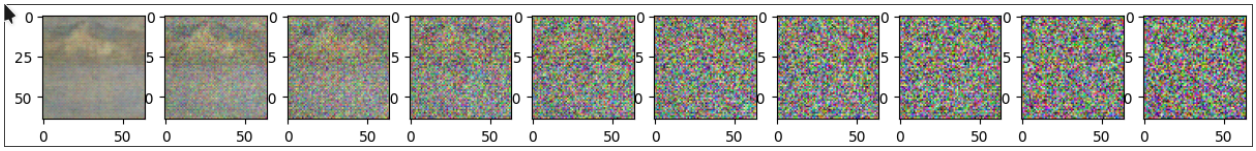
2000 Epoch Samples:

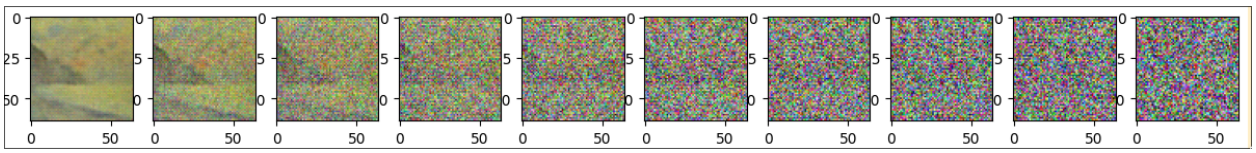


2500 Epoch Samples:



Results:





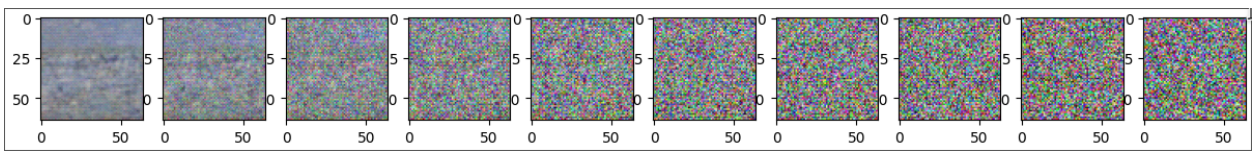


Figure 1.

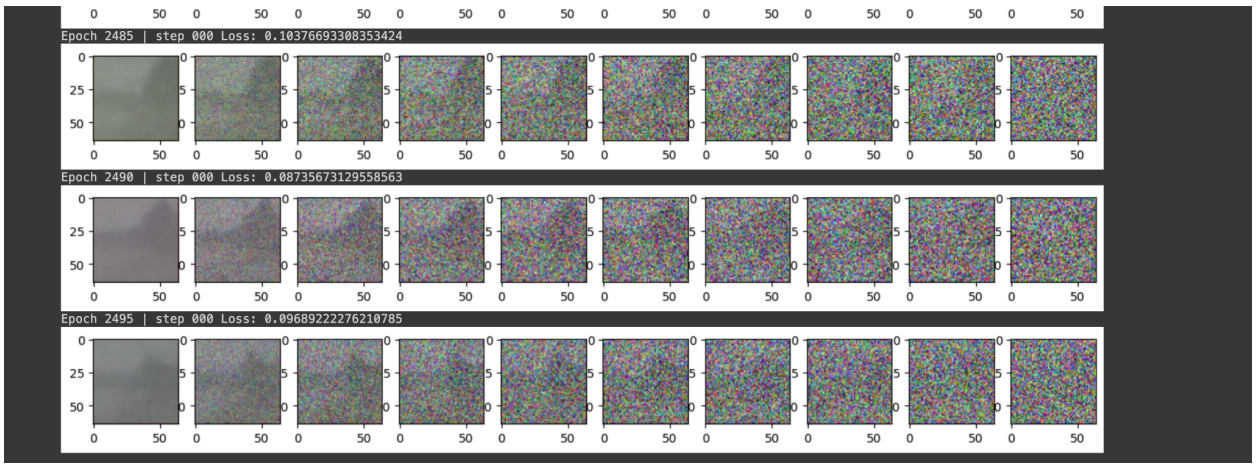
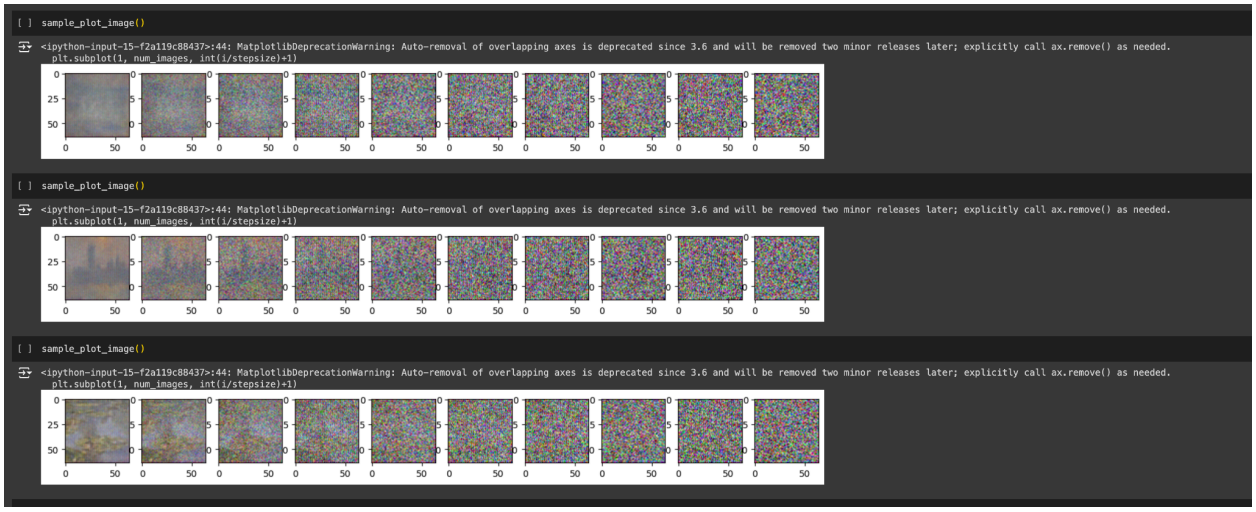
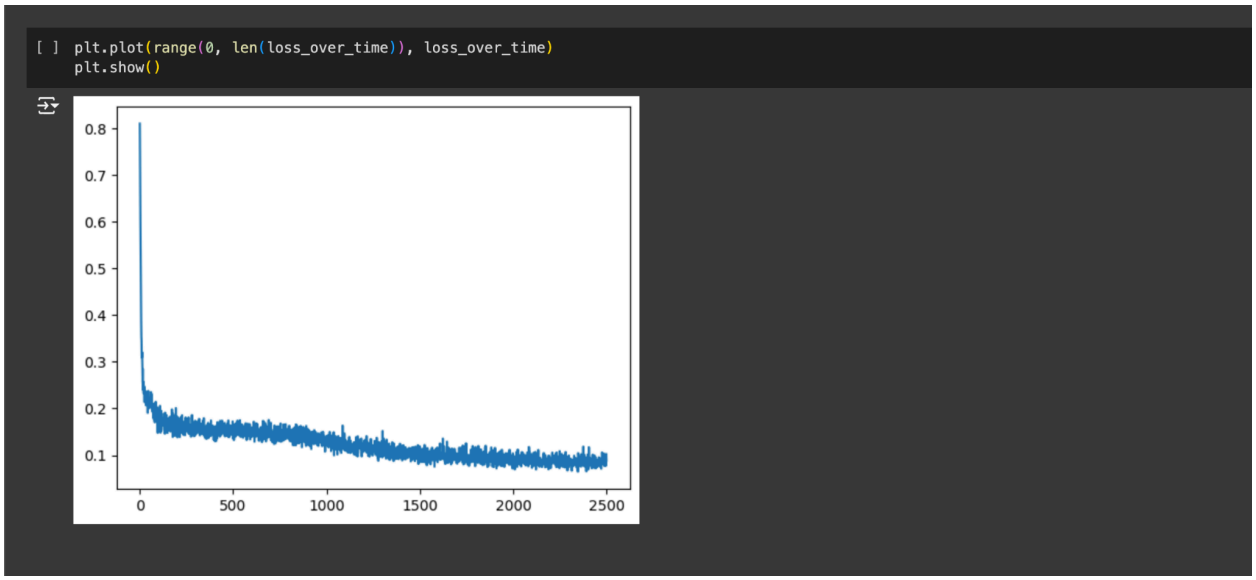


Figure 2:



Loss over the number of epochs:



Diffusion Model Library

For this particular comparison we used an already made diffusion model library called the Denoising diffusion probabilistic model through pytorch. This particular model is said to have the potential to rival GANs. It uses denoising score matching to estimate the gradient of the data distribution, followed by Langevin sampling to sample from the true distribution.

Implementations

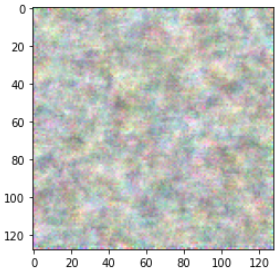
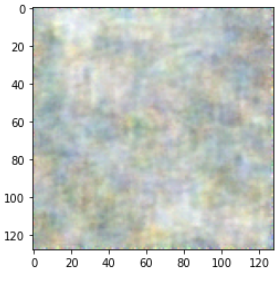
1st Implementation Of Model Library:

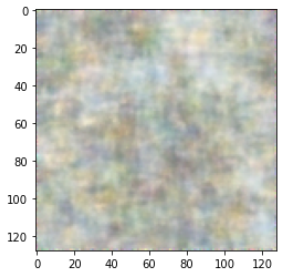
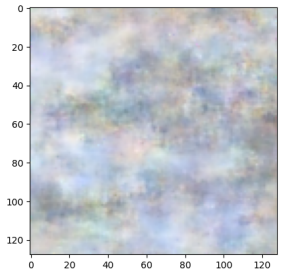
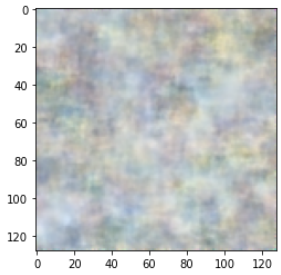
We utilized various approaches to implement this model, aiming to optimize our results. Initially, we leveraged our personal computers equipped with powerful GPUs, including the GeForce RTX 3050 and 3060, both among the latest offerings in the market. Utilizing Jupyter Notebook

on our local machines, we ensured seamless integration, requiring additional installations such as CUDA. CUDA toolkit, developed by NVIDIA, played a pivotal role in accelerating our computational processes by efficiently harnessing the capabilities of our GPUs.

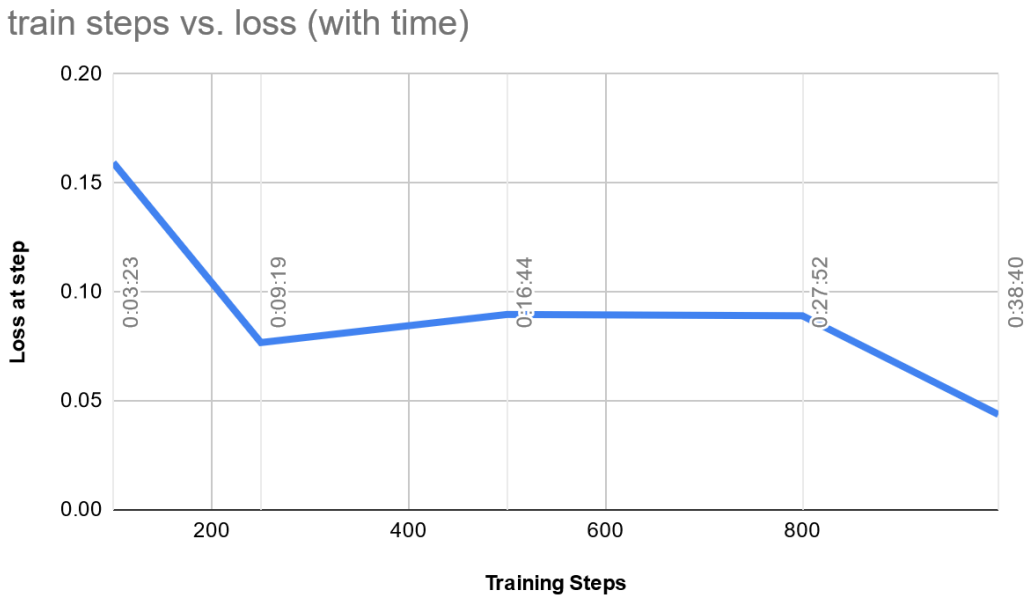
Results:

After encountering prolonged training times and sporadic errors on our computers, we made a crucial adjustment by reducing the training batch size from 32 to 16. This decision stemmed from observations that the training process was extending over several hours without yielding satisfactory results. Additionally, intermittent errors were impeding progress. By halving the batch size, we aimed to reduce computational strain and enhance the stability of the training. This adjustment not only expedited training but also helped reduce the occurrence of errors, making the progress smoother in model optimization.

TRAIN STEPS	TIME	LOSS	PRODUCED IMAGE
100	00:03:23	0.1594	
250	00:09:19	0.0768	

500	00:16:44	0.0897	
800	00:38:40	0.0891	
999	00:38:40	0.0438	

GPU = RTX 3050, **Batch_size** = 16, **train learning rate** = 2e-5



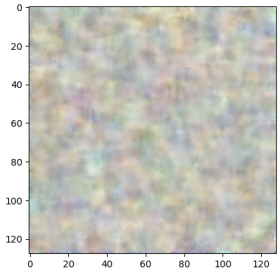
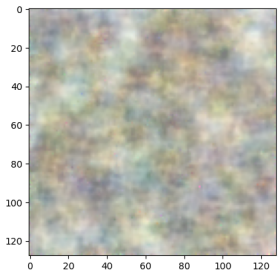
Problems Encountered:

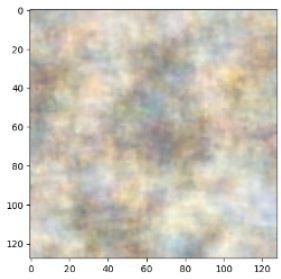
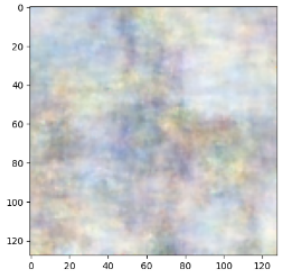
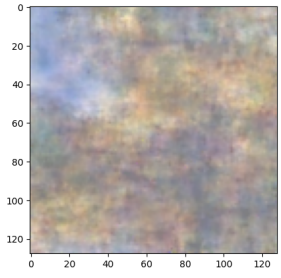
During attempts to train the model on our personal computers, we encountered significant challenges. These ranged from memory allocation issues, preventing the use of a 256-image size, requiring downsizing to 128 or 64. Furthermore, we faced constraints wherein the training process could not surpass 999 steps without kernel failures.

2nd Implementation Of Model:

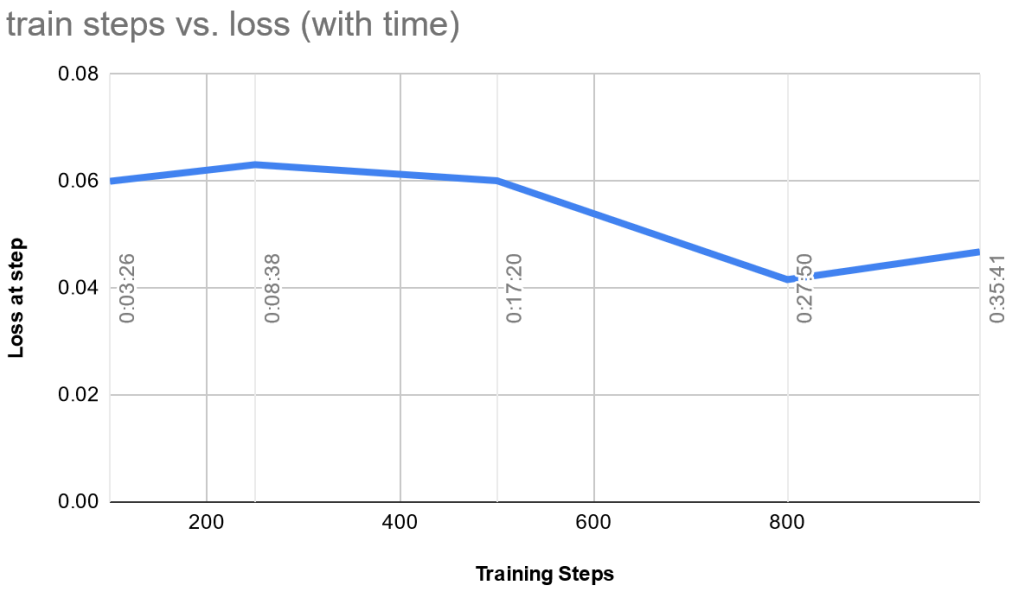
We also deployed the model through Google Colab Pro, so we could have access to more powerful GPUs through Google's resources to process the data faster and hopefully more accurately. With gained access to an L4 Tensor Core GPU, we not only achieved faster execution but also had the capability to run the library at the 32 batch size and the learning rate of 8e-5.

Results:

Train Steps	Time	Loss	Produced Image
100	00:03:26	0.0600	
250	00:08:38	0.0631	

500	00:17:20	0.0601	
800	00:27:50	0.0416	
999	00:35:41	0.0468	

GPU = L4 Tensor Core GPU (Google Colab), **Batch_size** = 32, **train learning rate** = 8e-5



Problems Encountered:

Additionally, we encountered several challenges when utilizing Google Colab for our project. Despite its convenience and accessibility, one significant issue arose from the limited availability of computing units, often resulting in delays or interruptions during model training sessions. Furthermore, we faced a critical error related to the computation of the Fréchet distance, specifically due to an unexpected imaginary component present in the covariance matrix calculation. This anomaly led to a value error, hindering our ability to accurately produced the results we wanted.

Overall Results & Problems Between Both Implementations:

Despite similar time results, employing the L4 Tensor GPU via Google Colab yielded clearer images compared to using the RTX 3050 on a personal computer. This improvement was attributed to the ability to utilize larger learning rates and training batch sizes, resulting in reduced loss throughout the training process. However, both implementations encountered limitations, restricting training to no more than 1000 steps due to recurring errors. On the personal computer, these errors often manifested as kernel failures, while within the library on Google Colab, inconsistencies in computation led to errors. Subsequent analysis indicated that the dataset size likely contributed to these issues, prompting the need for further investigation and optimization.

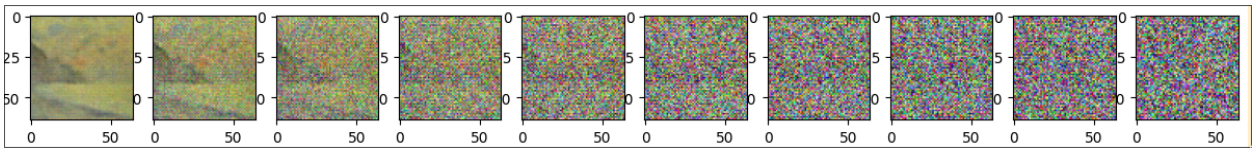
Conclusion

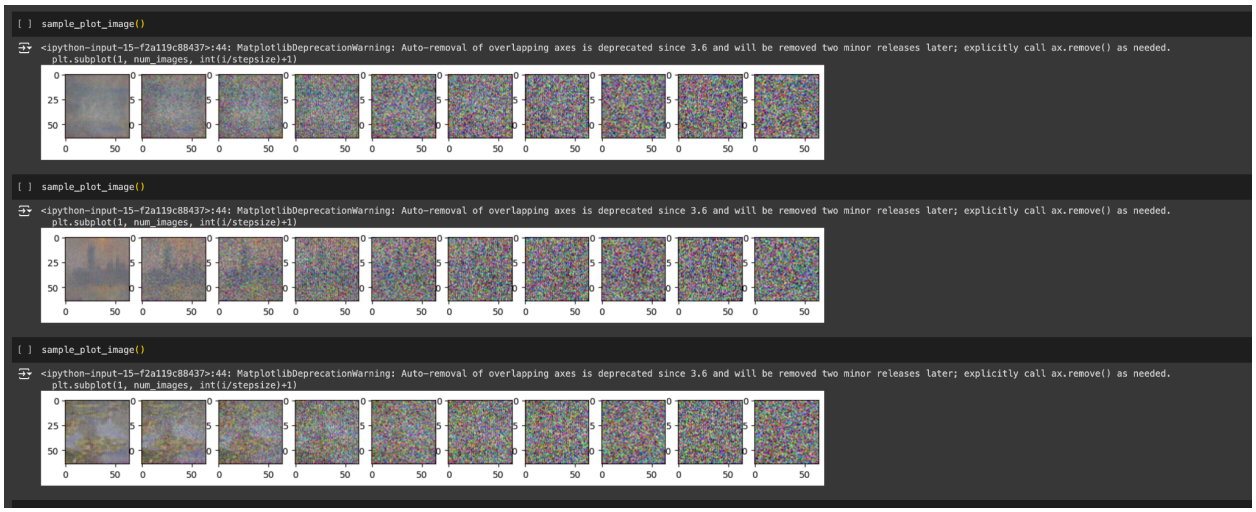
GANS and Diffusion models stand as the pinnacle of image generation technology, boasting unparalleled capabilities in their ability to craft hyperrealistic images. Today, cutting-edge image generators such as DALL-E 2, DALL-E 3, and MidJourney harness the power of diffusion or GAN models to bring to life their visually stunning creations. Moreover, Stable Diffusion models have emerged as the go-to choice for seamlessly translating text into images, a feat that was once considered science fiction. The advent of both GANS and Diffusion models has heralded a paradigm shift in the realm of image generation, revolutionizing how we perceive and interact with visual content. Notably, Diffusion models have transcended static imagery and have been integrated into OpenAI's latest groundbreaking model, SORA. Leveraging the transformative capabilities of the transformer-diffusion architecture, SORA represents a pioneering leap forward in video generation technology, showcasing the versatility and potency of Diffusion models.

Despite the initial shortcomings encountered with DCGAN, primarily attributed to mode collapse, the integration of the Wasserstein distance within our WGAN-GP models marked a significant improvement. Notably, the images produced by the WGAN-GP model bore a striking resemblance to the paintings of Monet, who’s use of colors surpassed those generated by the diffusion model. Such findings were unexpected, considering the modernity of the Diffusion model and its widespread adoption as a cornerstone for image generation technology.

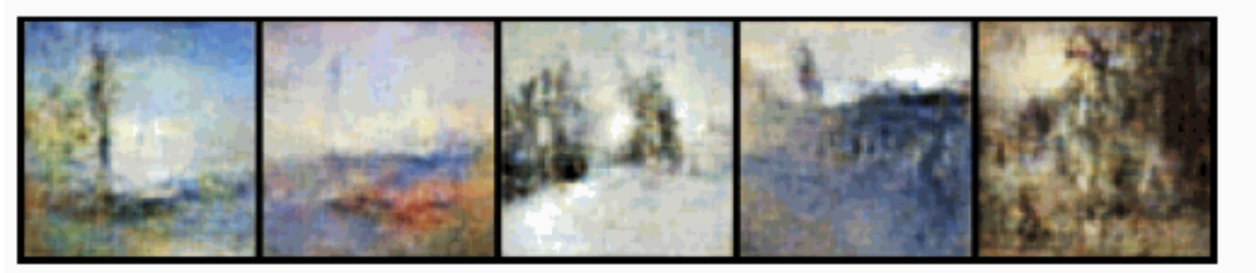
Therefore, in our renewed endeavor with this project, we intend to leverage a substantially larger dataset and extend the training duration of the diffusion model significantly. Through these adjustments, our aim is to elevate the performance of the diffusion model to surpass that of the WGAN-GP.

End Results for Diffusion:





End Results with WGAN-GP:



References

- Arjovsky, M., Chintala, S., & Bottou, L. (2017, December 6). *Wasserstein Gan*. arXiv.org. https://arxiv.org/abs/1701.07875

- Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., & Courville, A. (2017). *Improved Training of Wasserstein GAN.* arXiv.org, https://arxiv.org/abs/1704.00028

- Ho, J., Jain, A., & Abbeel, P. (2020, December 16). *Denoising Diffusion Probabilistic models*. arXiv.org. https://arxiv.org/abs/2006.11239

- Radford, A., Metz, L., & Chintala, S. (2016, January 7). *Unsupervised representation learning with deep convolutional generative Adversarial Networks*. arXiv.org. https://arxiv.org/abs/1511.06434

- Ronneberger, O., Fischer, P., & Brox, T. (2015, May 18). *U-Net: Convolutional Networks for Biomedical Image Segmentation.* arXiv.org. https://arxiv.org/abs/1505.04597

- Wang, Q., Li, H., Zhang, Q., Wang, P., & Zhu, Y. (2019, June 21). *Single Image Super-Resolution via Dense Blended Attention Generative Adversarial Network for Clinical Diagnosis.* ResearchGate. https://www.researchgate.net/publication/333841926_single_image_super-resolution_via_dense_blended_attention_generative_adversarial_network_for_clinical_diagnosis

- O'Connor, Ryan (2022, May 12). *Introduction to Diffusion Models for Machine Learning*. AssemblyAI. https://www.assemblyai.com/blog/diffusion-models-for-machine-learning-introduction/

Responsibilities

Ly Jacky Nhiayi: Created Github, Researched Diffusion, Created Diffusion Model, Improved U-net Models with Cosine Scheduling, and Improved U-net Model with Linear Scheduling.

Erica Payne: Ran Diffusion Library Models through PC and Google Colab. Handled errors / researched diffusion library improvements and limitations. Recorded samples, times, and loss throughout the model on Google Colab.

Leonardo Ramirez: research alternate methodologies.

Leonard Garcia: Researched diffusion theory and implementations, contributed to the project presentation and report

Jackson Bentley: Contributed to the project presentation, edited report.

Angel Villalobos: Trained and debugged Diffusion Models, Researched diffusion models.

Jesus Cruz: Researched possible GAN models for implementation and metrics for success, helped debug and and run GAN models .

David Camacho: Trained GANs on Colab, Augmented dataset, Improved Mode Collapse, GIF outputs

Youssef Elzein: Trained models on Colab

Matthew Johnson: Ran Diffusion Library Models through PC. Recorded samples, times, and loss throughout the model on PC.