Introduction to GANs

GANs are a type of deep learning model used for generating new data samples that resembles given dataset. In GANs, we have two type of neural network -

1. Generator
2. Discriminator

Generator - It takes random noise as input and generates data similar to real datset. Its goal is to create samples that are indistinguishable from real data, effectively fooling the Discriminator.

Discriminator - It takes real or generated data as input and classifies them as real(from dataset) or fake(from generator). Its goal is to distinguish between real and generated data as accurately as possible.

How GANs Work

The generator and discriminator are trained simultaneously in a min-max game.

- The Generator tries to minimize the Discriminator's ability to classify generated data as fake.
- The Discriminator tries to maximize its ability to correctly predict the generated data as real or fake.

This is modeled mathematically as:

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

where,

- x : real data sample.
- z : random noise input to the Generator.
- G(z) : generated data sample.
- D(x) : Discriminator's probability that x is real.

Dataset description and preprocessing steps.

We are using MNIST dataset which contains 70,000 greyscale images of handwritten digits from 0 to 9. So discriminator have ten classes to predict from. Each image is 28*28 pixels and have corresponding digit labels (0-9). **Dataset split:**

- Training set : 60,000 images
- Test set : 10,000 images

Preprocessing Steps:

1. **Loading the dataset :** It is done using load_data() function from keras.datasets.mnist

```
(train_images, train_labels), (_, _) =
tf.keras.datasets.mnist.load_data()
```

2. **Reshaping the Images :**

```
train_images = train_images.reshape(train_images.shape[0], 28,
28, 1).astype('float32')
```

- Each image is reshaped from (28, 28) to (28, 28, 1) to explicitly include the channel dimension (grayscale has one channel).
- This ensures compatibility with the generator and discriminator models, which are designed to work with 4D tensors: (batch_size, height, width, channels).

3. **Normalizing the pixel values :**

```
train_images = (train_images - 127.5) / 127.5
```

- MNIST pixel values range from 0 to 255. They are normalized to [-1, 1]
- The generator uses a tanh activation function, which outputs values in the range of [-1, 1]
- Normalizing inputs to the same range improves model stability and convergence during training.

4. **Shuffling and Batching :**

```
BUFFER_SIZE = 60000
BATCH_SIZE = 256

train_dataset =
tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_S
IZE).batch(BATCH_SIZE)
```

- The dataset is shuffled with a buffer size of 60,000 (the total number of MNIST training images).
- The dataset is divided into batches of size 256 and each batch has a shape [256, 28, 28, 1]. Batching improves training efficiency and leverages parallel computation.

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import os
from tensorflow.keras import layers
import time

# Load the dataset
(train_images, train_labels), (_, _) =
tf.keras.datasets.mnist.load_data()
```

```
train_images = train_images.reshape(train_images.shape[0], 28, 28,
1).astype('float32') # Reshape the images from (28, 28) to (28, 28, 1)
as needed by CNNs
train_images = (train_images - 127.5) / 127.5   # Normalize to [-1, 1]

BUFFER_SIZE = 60000
BATCH_SIZE = 256

# Batch and shuffle the data
train_dataset =
tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).
batch(BATCH_SIZE)
```

Model Architecture

Generator

Dense Layer:

- The first dense layer maps the input noise into a vector of size 7×7×256, effectively shaping the foundation for the subsequent layers.
- Batch Normalization normalizes the activations to stabilize and accelerate training.
- A LeakyReLU activation function introduces non-linearity and mitigates the vanishing gradient problem.

Reshape Layer:

- Reshapes the 1D output of the dense layer into a 3D feature map of shape 7×7×256, preparing it for the transposed convolutional layers.

First Transposed Convolutional Layer:

- Output Shape: (None,7,7,128)
- Applies a 3×3 kernel with a stride of 1×1 and 'same' padding to refine the initial feature maps.
- Batch Normalization and LeakyReLU are applied for stability and activation.

Second Transposed Convolutional Layer:

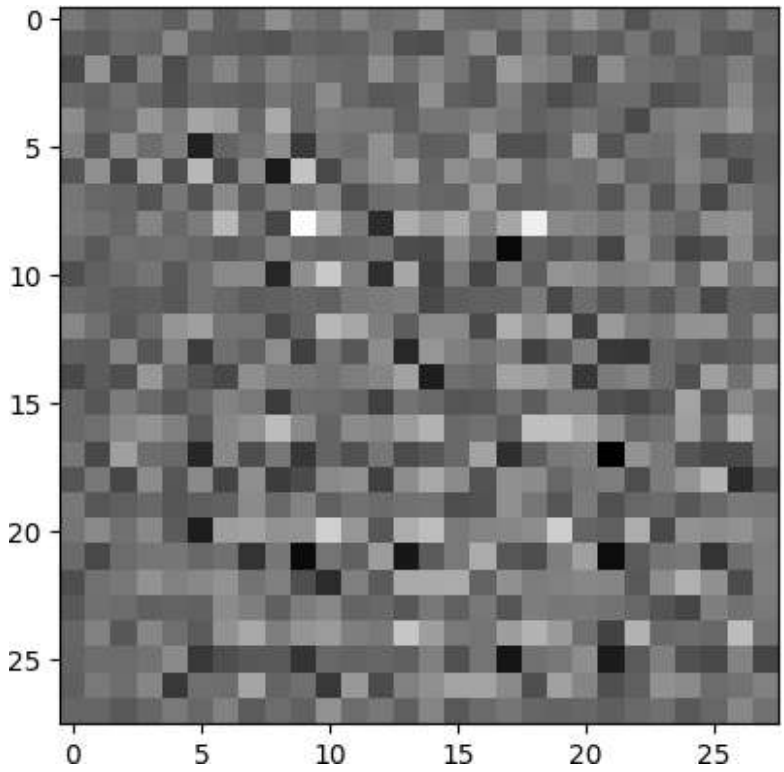
- Output Shape: (None,14,14,64)
- Upsamples the feature maps by doubling their spatial dimensions using a stride of 2×2.
- Again, Batch Normalization and LeakyReLU ensure the generator learns efficiently.

Final Transposed Convolutional Layer:

- Output Shape: (None,28,28,1)
- Produces the final grayscale image using a 2×2.
- Uses the tanh activation function to scale pixel values to the range [−1,1], matching the normalized input range of the discriminator.

```
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False,
input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256)   # None is the
batch size

    model.add(layers.Conv2DTranspose(128, (3, 3), strides=(1, 1),
padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (3, 3), strides=(2, 2),
padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (3, 3), strides=(2, 2),
padding='same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model

# Use the (as yet untrained) generator to create an image.
generator = make_generator_model()

noise = tf.random.normal([1, 100])
generated_image = generator(noise, training=False)

plt.imshow(generated_image[0, :, :, 0], cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7e2ec81b5910>
```



Discriminator

Convolutional Layers:

- Two convolutional layers are used, each with a kernel size of 3×3, stride of 2×2, and 'same' padding to maintain spatial integrity during feature extraction.
- A LeakyReLU activation function (α=0.2) ensures better gradient flow for negative activations, addressing the "dying ReLU" problem.
- Batch Normalization is applied after each convolution to stabilize and accelerate training by normalizing feature maps.

Dropout:

- A dropout rate of 0.4 is applied after each convolutional layer.
- This reduces the risk of overfitting and ensures the model generalizes well to unseen data.

Flatten Layer:

- Converts the 2D feature maps into a 1D vector, preparing the data for the dense layer.

Output Layer:

- A single unit with a sigmoid activation function outputs a scalar value.
- This scalar represents the probability that the input image is "real" (close to 1) or "fake" (close to 0).

```
def make_discriminator_model():
    model = tf.keras.Sequential([
        layers.Conv2D(64, (3, 3), strides=(2, 2), padding='same',

kernel_regularizer=tf.keras.regularizers.l2(0.001),
                      input_shape=[28, 28, 1]),
        layers.BatchNormalization(),
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.4),

        layers.Conv2D(128, (3, 3), strides=(2, 2), padding='same',

kernel_regularizer=tf.keras.regularizers.l2(0.001)),
        layers.BatchNormalization(),
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.4),

        layers.Flatten(),
        layers.Dense(1, activation='sigmoid')
    ])
    return model
```

Loss Functions

We use BinaryCrossentropy loss here because the model will predict only two values i.e real or fake

```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

Discriminator Loss

- The discriminator's loss measures its ability to classify real images as "real" and generated images as "fake."
- The discriminator tries to output 1 for real images.
- The discriminator tries to output 0 for fake images.

```
# Discriminator loss
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)
    return real_loss + fake_loss
```

Generator Loss

- The generator's loss measures its ability to fool the discriminator into classifying generated images as "real."
- The generator aims for the discriminator to output 1 for its images.

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

Optimizers

- Adam optimizer is used in both descriminator and generator
- Learning rate is $1*10^-4$
- Low learning rate ensures stable training
- beta helps us smooth out oscillations during training

```
# Set up the optimizers
generator_optimizer = tf.keras.optimizers.Adam(1e-4, beta_1=0.5)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4, beta_1=0.5)

generator = make_generator_model()
discriminator = make_discriminator_model()

# Save checkpoints
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint =
tf.train.Checkpoint(generator_optimizer=generator_optimizer,

discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)

# Creates a fixed noise vector to generate images for consistent
evaluation during training.
EPOCHS = 100
noise_dim = 100
num_examples_to_generate = 16 # for a 4x4 grid

seed = tf.random.normal([num_examples_to_generate, noise_dim])
```

Training Strategy

- Firstly, noise vector of shape [BATCH_SIZE, noise_dim] is created.
- gen_tape and disc_tape tracks operations for generator and discriminator simultaneously so there gradient can be calculated.
- Then, discriminator is evaluated on real images from the dataset with Gaussian noise of mean = 0 and standard deviation = 0.05, to prevent overfitting and make the discriminator robust
- Similarly, discriminator is also evaluated on fake images produced by generator.
- The generator loss encourages the generator to produce images that the discriminator classifies as real.
- The discriminator loss measures how well the discriminator can distinguish real images from fake ones.

- Computes the gradients of the generator's loss and discriminator's loss with respect to its trainable variables.
- Updates the generator's weights and discriminator's weights using its gradients.
- Return the generator and discriminator loss for monitoring.

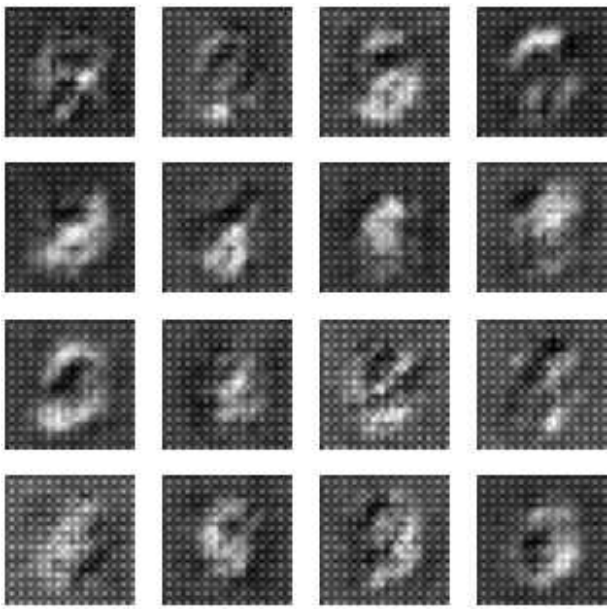
```
gen_losses = []
disc_losses = []

@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

# forward pass
    with tf.GradientTape() as gen_tape, tf.GradientTape() as
disc_tape:
        generated_images = generator(noise, training=True)
        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

# backward pass
    gradients_of_generator = gen_tape.gradient(gen_loss,
generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))

discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator
, discriminator.trainable_variables))

    return gen_loss, disc_loss

# Generate and save a grid of images to monitor the progress of the
generator during training
def generate_and_save_images(model, epoch, test_input):
    predictions = model(test_input, training=False)
    os.makedirs('generated_images', exist_ok=True)

    fig = plt.figure(figsize=(4, 4))
    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5,
cmap='gray')
        plt.axis('off')

    plt.savefig(f'generated_images/image_at_epoch_{epoch:04d}.png')
    plt.show()
```

Training and vizualization

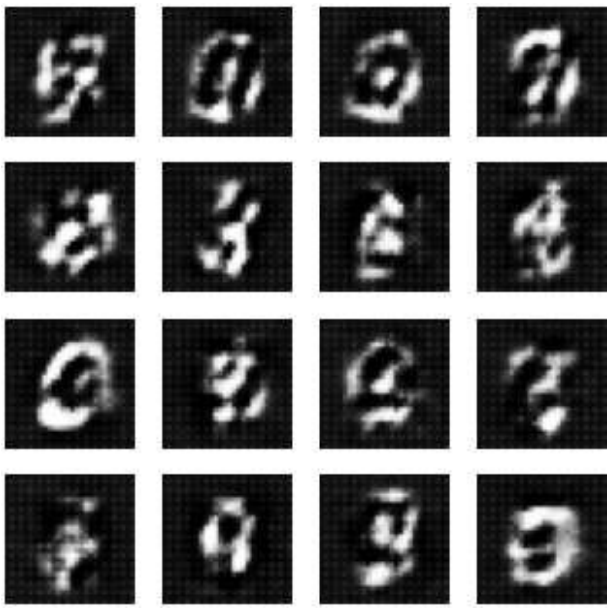
- Outer loop iterates for 100 epochs and start records the start time of each epoch
- For each image_batch g_loss and d_loss is calculated and appened to the respective lists
- We call the generate_and_save_images function to produce and save images from the generator
- The model is saved after every 15 epochs
- Display the time taken for each epoch
- Save the model after all the epochs are completed
- Plot the generator and discriminator loss

```
def train(dataset, epochs):
    for epoch in range(epochs):
        start = time.time()

        epoch_gen_loss = []
        epoch_disc_loss = []

        for image_batch in dataset:
            g_loss, d_loss = train_step(image_batch)
            epoch_gen_loss.append(g_loss)
            epoch_disc_loss.append(d_loss)

        # Calculate mean losses for this epoch
        gen_losses.append(np.mean(epoch_gen_loss))
        disc_losses.append(np.mean(epoch_disc_loss))

        # Display results
        generate_and_save_images(generator, epoch + 1, seed)
        print(f"Epoch {epoch + 1}/{epochs}")
        print(f"Generator Loss: {gen_losses[-1]:.4f}, Discriminator
Loss: {disc_losses[-1]:.4f}")
        print(f"Time for epoch {epoch + 1} is {time.time() -
start:.2f} sec\n")

        # Save the model every 15 epochs
        if (epoch + 1) % 15 == 0:
            checkpoint.save(file_prefix=checkpoint_prefix)

    # Plot the generator and discriminator loss
    plt.figure(figsize=(10, 5))
    plt.plot(gen_losses, label='Generator Loss')
    plt.plot(disc_losses, label='Discriminator Loss')
    plt.title('Loss During Training')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

train(train_dataset, EPOCHS)
```

Before trained the model :-

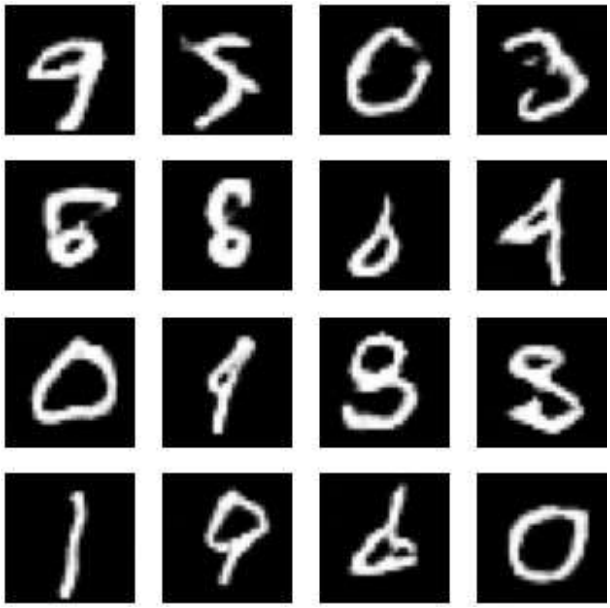


```
Epoch 1/100
Generator Loss: 1.4130, Discriminator Loss: 0.8617
Time for epoch 1 is 15.39 sec
```

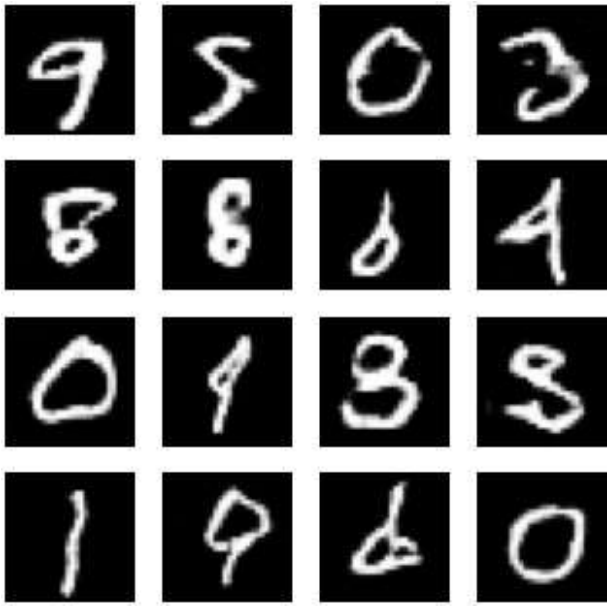


```
Epoch 2/100
Generator Loss: 1.4394, Discriminator Loss: 0.8765
Time for epoch 2 is 9.11 sec
```

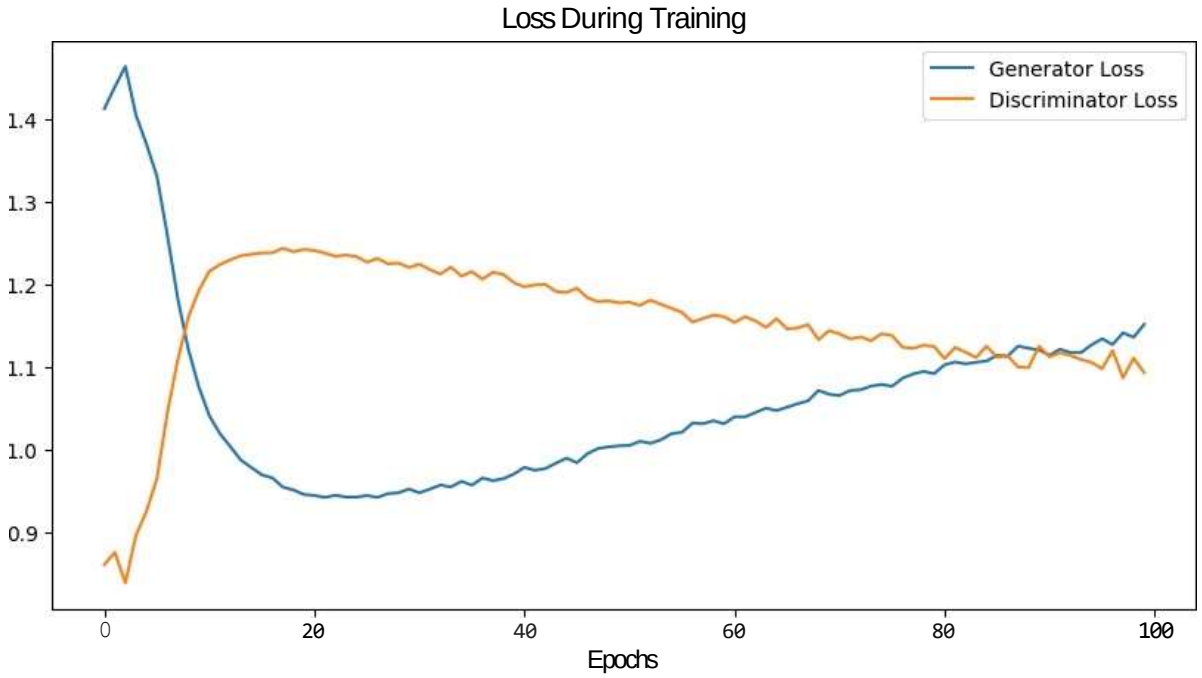
After trained the model:-



```
Epoch 99/100
Generator Loss: 1.1366, Discriminator Loss: 1.1114
Time for epoch 99 is 9.16 sec
```



```
Epoch 100/100
Generator Loss: 1.1521, Discriminator Loss: 1.0938
Time for epoch 100 is 9.41 sec
```



Conclusion

This project demonstrated the implementation and training of a GAN to generate realistic handwritten digits using the MNIST dataset. The generator and discriminator architectures were carefully designed and trained in an adversarial setup. Results showcased the generator's ability to improve over epochs, producing increasingly realistic images. While the model performed well, certain challenges like mode collapse and variability in generated images were noted, indicating areas for further optimization. Overall, this project highlights the potential and intricacies of GANs in generative modeling tasks.