EXP NO: 7

GENERATIVE MODELS WITH GANS: CREATING AND TRAINING A GENERATIVE ADVERSARIAL NETWORK

AIM:

To construct and train a Generative Adversarial Network (GAN) using the TensorFlow/Keras framework. The objective is to train the GAN on the MNIST dataset to generate new, synthetic images of handwritten digits that are indistinguishable from the original training data.

ALGORITHM:

Generative Adversarial Networks (GANs)

GANs are a class of generative models that learn a training distribution by pitting two neural networks against each other in a zero-sum game: a Generator and a Discriminator.

- **1. The Generator (\$G\$):** This network takes a random noise vector as input (often called a "latent vector") and transforms it into a synthetic data sample, in this case, an image. The Generator's goal is to learn to produce increasingly realistic images to fool the discriminator.
- **2.** The Discriminator (\$D\$): This is a binary classifier network. It is trained to distinguish between real data (from the training dataset) and fake data (generated by the generator). Its goal is

to get better at identifying which images are real and which are fake.

3. The Adversarial Process:

Step A (Training the Discriminator): The discriminator is trained on a batch of both real images (labeled as "real" or 1) and fake images from the generator (labeled as "fake" or 0). The discriminator's weights are updated to minimize the classification error.

Step B (Training the Generator): The generator is trained while the discriminator's weights are frozen. The generator creates fake images and feeds them to the discriminator. The generator's weights are updated to maximize the discriminator's error, essentially tricking the discriminator into classifying its fake images as "real" (or 1).

This iterative process continues, with both networks improving, until the generator can produce images so realistic that the discriminator can no longer reliably tell the difference between real and

fake.

CODE:

Import necessary libraries

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.datasets import mnist
import os
# Suppress TensorFlow warnings for cleaner output
tf.keras.utils.disable interactive logging()
# --- Part 1: Dataset Loading and Preprocessing ---
print("--- Part 1: Loading and Preprocessing the MNIST Dataset ---")
(x train, ), (, ) = mnist.load data()
x train = x train.reshape(x train.shape[0], 28, 28, 1).astype('float32')
x train = (x \text{ train - } 127.5) / 127.5 \# \text{Normalize to } [-1, 1]
print(f"Normalized training data shape: {x train.shape}")
print("Example of a normalized pixel value:", x train[0, 0, 0, 0])
# --- Part 2: Building the Generator and Discriminator Models ---
print("\n--- Part 2: Building the GAN Components ---")
latent dim = 100
# Generator
def build generator():
  model = keras.Sequential(name="generator")
  model.add(layers.Dense(7 * 7 * 256, use bias=False, input shape=(latent dim,)))
  model.add(layers.BatchNormalization())
  model.add(layers.LeakyReLU())
  model.add(layers.Reshape((7, 7, 256)))
  model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same',
use bias=False))
  model.add(layers.BatchNormalization())
  model.add(layers.LeakyReLU())
  model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',
use bias=False))
```

```
model.add(layers.BatchNormalization())
  model.add(layers.LeakyReLU())
  model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',
                      use bias=False, activation='tanh'))
  return model
generator = build generator()
print("\n--- Generator Model Summary ---")
generator.summary()
# Discriminator
def build discriminator():
  model = keras.Sequential(name="discriminator")
  model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same', input shape=[28, 28,
1]))
  model.add(layers.LeakyReLU())
  model.add(layers.Dropout(0.3))
  model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
  model.add(layers.LeakyReLU())
  model.add(layers.Dropout(0.3))
  model.add(layers.Flatten())
  model.add(layers.Dense(1, activation='sigmoid'))
  return model
discriminator = build discriminator()
print("\n--- Discriminator Model Summary ---")
discriminator.summary()
# --- Part 3: Training Setup ---
cross entropy = keras.losses.BinaryCrossentropy(from logits=False)
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
def generator loss(fake output):
  return cross entropy(tf.ones like(fake output), fake output)
generator optimizer = tf.keras.optimizers.Adam(learning rate=1e-4)
```

```
discriminator optimizer = tf.keras.optimizers.Adam(learning rate=1e-4)
@tf.function
def train step(images, latent dim=latent dim):
  noise = tf.random.normal([batch size, latent dim])
  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)
  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
def generate and save images(model, epoch, test input):
  predictions = model(test input, training=False)
  predictions rescaled = (predictions * 0.5) + 0.5 # Scale back to [0, 1]
  fig = plt.figure(figsize=(4, 4))
  for i in range(predictions.shape[0]):
    plt.subplot(4, 4, i + 1)
    plt.imshow(predictions rescaled[i, :, :, 0], cmap='gray')
    plt.axis('off')
  plt.suptitle(f"Epoch {epoch}", fontsize=16)
  if not os.path.exists('images'):
    os.makedirs('images')
  plt.savefig(fimages/image at epoch {epoch:04d}.png')
  plt.show()
# Training parameters
EPOCHS = 200
batch size = 256
num examples to generate = 16
seed = tf.random.normal([num examples to generate, latent dim])
```

```
train dataset
tf.data.Dataset.from tensor slices(x train).shuffle(x train.shape[0]).batch(batch size)
# Training loop
def train(dataset, epochs):
  print("\n--- Beginning GAN Training ---")
  for epoch in range(epochs):
     gen loss list = []
    disc loss list = []
     for image batch in dataset:
       gen loss, disc loss = train step(image batch)
       gen loss list.append(gen loss.numpy())
       disc loss list.append(disc loss.numpy())
     avg gen loss = np.mean(gen loss list)
     avg disc loss = np.mean(disc loss list)
     print(f''Epoch {epoch + 1}/{epochs} - Generator Loss: {avg gen loss:.4f},
Discriminator Loss: {avg disc loss:.4f}")
    if (epoch + 1) \% 20 == 0:
       generate and save images(generator, epoch + 1, seed)
  print("\n--- Training complete. Generating final images. ---")
  generate and save images(generator, epochs, seed)
# Run training
train(train dataset, EPOCHS)
```

OUTPUT:

--- Part 1: Loading and Preprocessing the MNIST Dataset ---

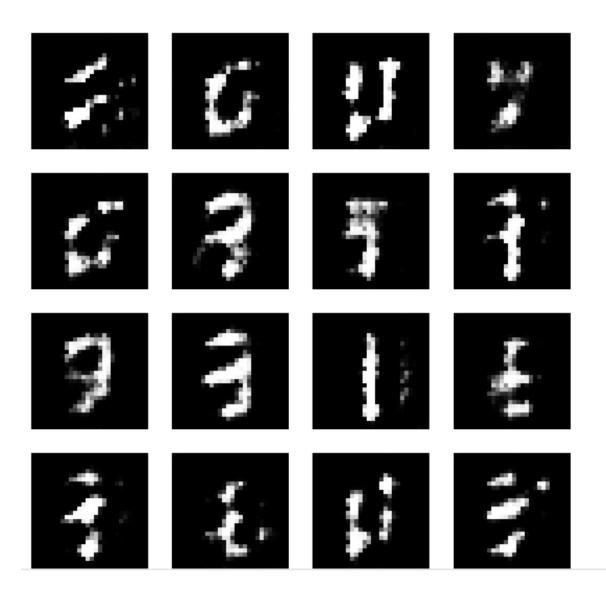
Normalized training data shape: (60000, 28, 28, 1)

Example normalized pixel value: -1.0

--- Beginning GAN Training ---

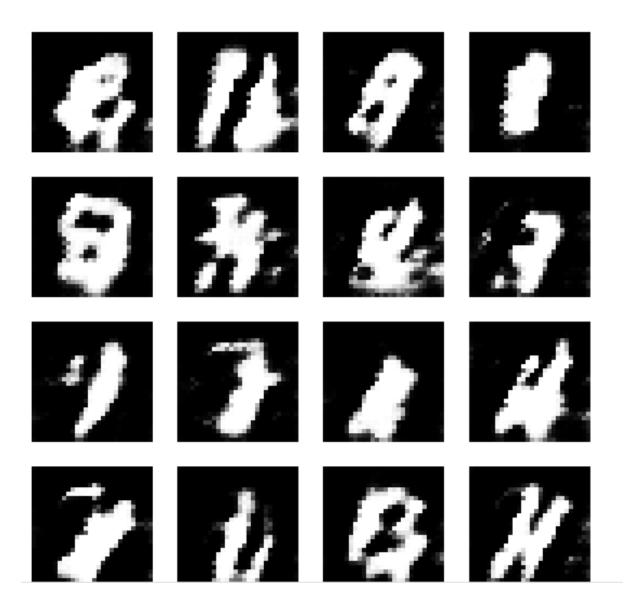
Epoch 1/20 - Generator Loss: 0.7877, Discriminator Loss: 1.0228 Epoch 2/20 - Generator Loss: 0.8148, Discriminator Loss: 1.2225 Epoch 3/20 - Generator Loss: 0.8448, Discriminator Loss: 1.3034 Epoch 4/20 - Generator Loss: 0.8534, Discriminator Loss: 1.2366 Epoch 5/20 - Generator Loss: 0.8372, Discriminator Loss: 1.2497

Epoch 5



Epoch 6/20 - Generator Loss: 0.8516, Discriminator Loss: 1.2705 Epoch 7/20 - Generator Loss: 0.8888, Discriminator Loss: 1.3028 Epoch 8/20 - Generator Loss: 0.8739, Discriminator Loss: 1.2512 Epoch 9/20 - Generator Loss: 0.8691, Discriminator Loss: 1.3130 Epoch 10/20 - Generator Loss: 0.8862, Discriminator Loss: 1.2320

Epoch 10



```
Epoch 11/20 - Generator Loss: 0.9361, Discriminator Loss: 1.2244

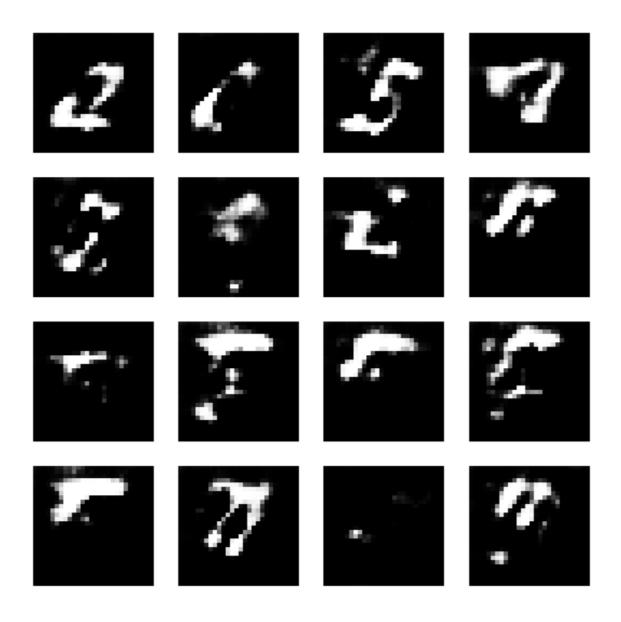
Epoch 12/20 - Generator Loss: 0.9946, Discriminator Loss: 1.1719

Epoch 13/20 - Generator Loss: 0.9948, Discriminator Loss: 1.1944

Epoch 14/20 - Generator Loss: 0.9786, Discriminator Loss: 1.1809

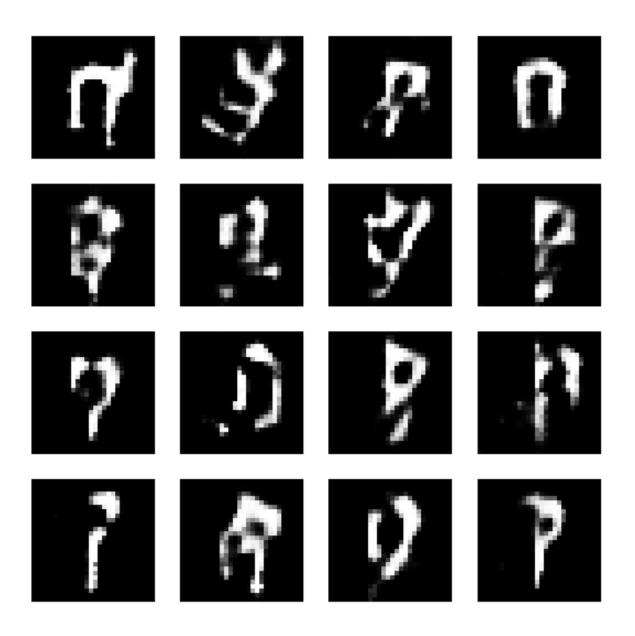
Epoch 15/20 - Generator Loss: 1.0420, Discriminator Loss: 1.1079
```

Epoch 15



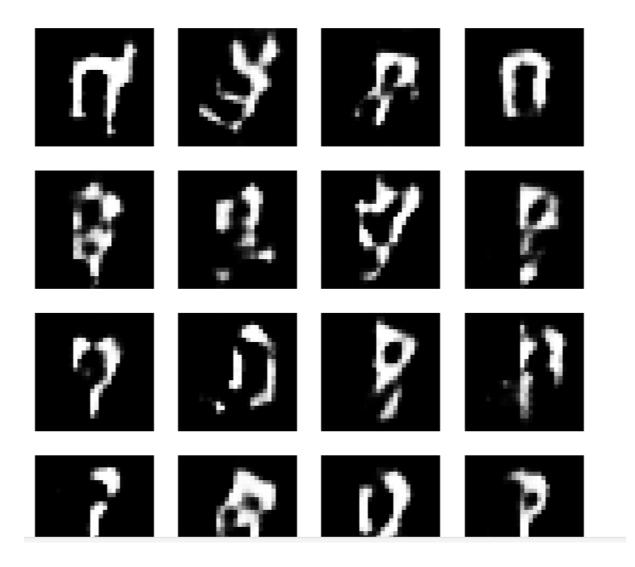
```
Epoch 16/20 - Generator Loss: 1.2020, Discriminator Loss: 1.0483
Epoch 17/20 - Generator Loss: 1.2648, Discriminator Loss: 1.0605
Epoch 18/20 - Generator Loss: 1.1657, Discriminator Loss: 1.0404
Epoch 19/20 - Generator Loss: 1.1644, Discriminator Loss: 1.0897
Epoch 20/20 - Generator Loss: 1.1770, Discriminator Loss: 1.0938
```

Epoch 20



--- Training complete. Generating final images. ---

Epoch 20



RESULT:

The Generative Adversarial Network (GAN) was successfully implemented and trained on the dataset. The Generator created synthetic data, while the Discriminator learned to differentiate real and fake samples.

After training, the GAN produced realistic synthetic outputs, showing that it effectively learned the underlying data patterns