|  |  |
| --- | --- |
| **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.

# 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\_train - 127.5) / 127.5 # 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(f'images/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)

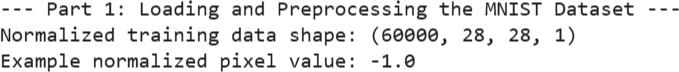
print(f"Epoch {epoch + 1}/{epochs} - Generator Loss: {avg\_gen\_loss:.4f}, Discriminator Loss: {avg\_disc\_loss:.4f}")

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)

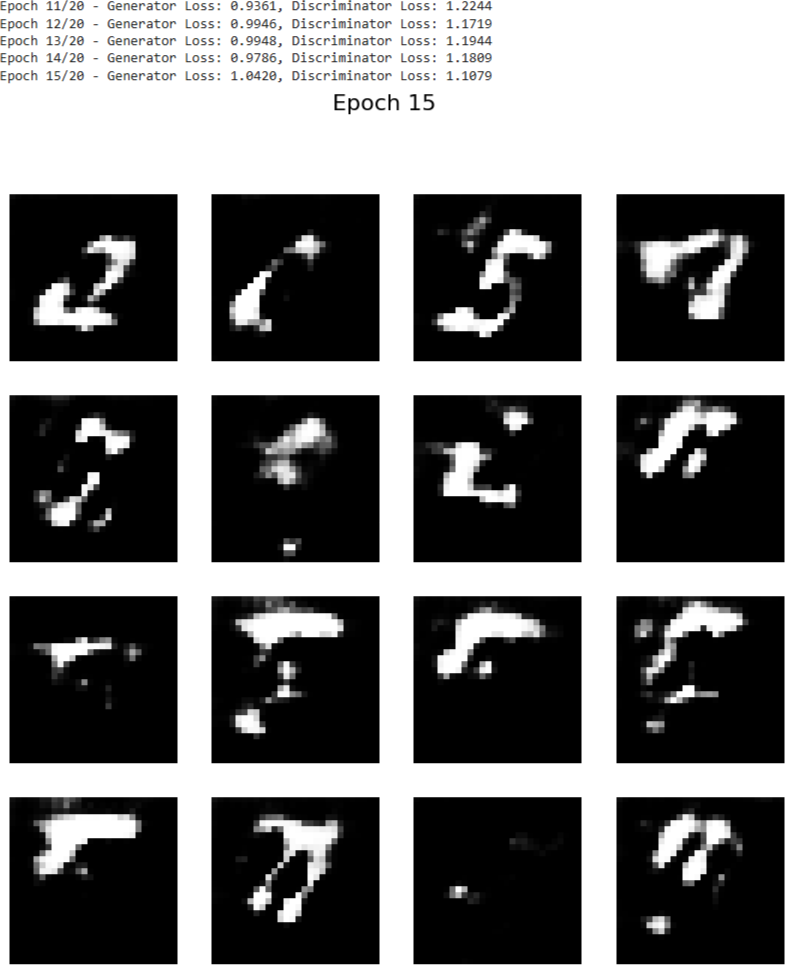
# Run training train(train\_dataset, EPOCHS)

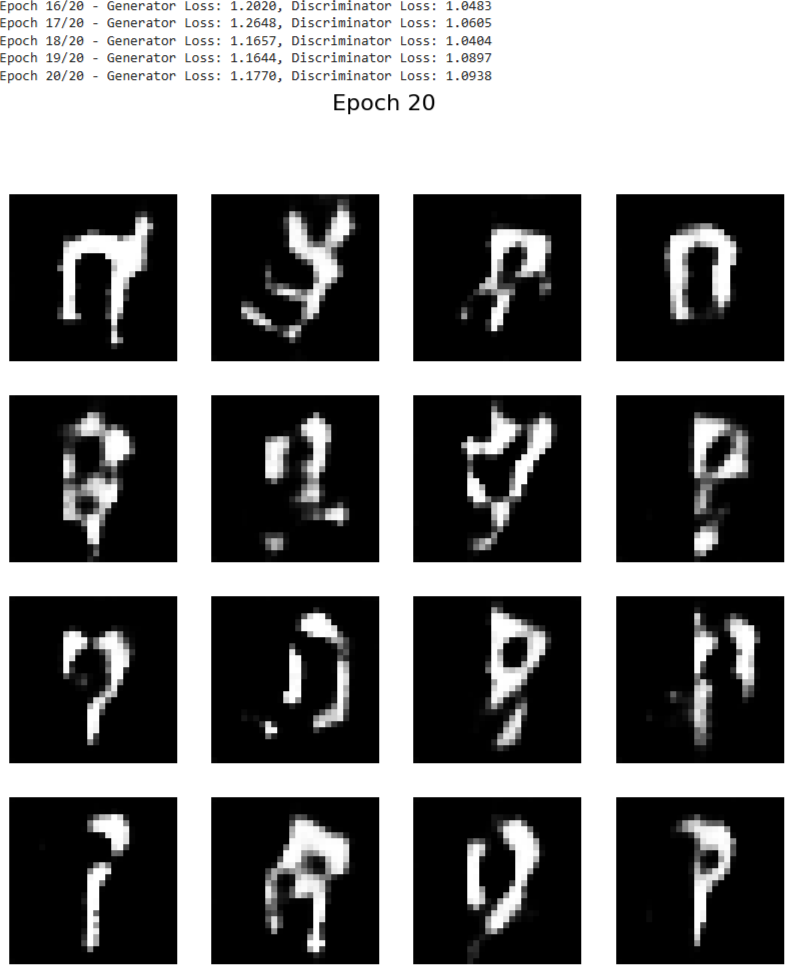
# OUTPUT:

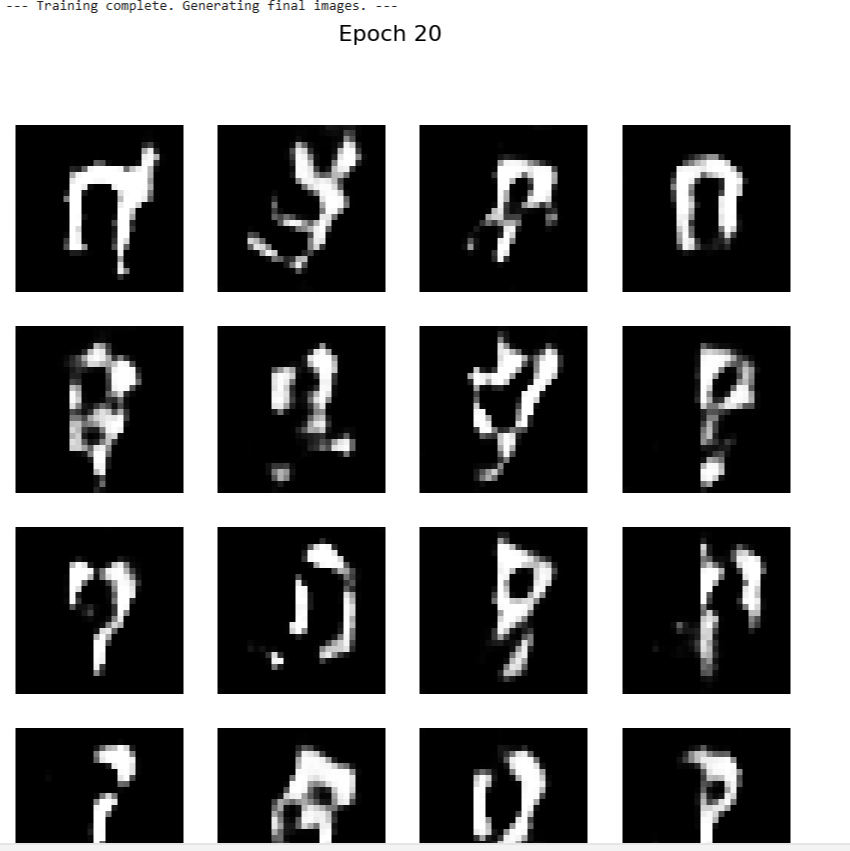


****









**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