# 221501039

# 9. BUILDING A SIMPLE GENERATIVE ADVERSARIAL NETWORK (GAN) USING TENSORFLOW

| **EX.N0 : 9** | **BUILDING A SIMPLE GENERATIVE ADVERSARIAL NETWORK (GAN) USING TENSORFLOW** |
| --- | --- |
| **DATE : 25/03/2025** |

**AIM:**

To build and train a simple Generative Adversarial Network (GAN) using TensorFlow for generating images.

# ALGORITHM:

Step 1: Import required TensorFlow and data libraries. Step 2: Load and preprocess the MNIST dataset.

Step 3: Define the Generator and Discriminator models.

Step 4: Create the loss functions and optimizers for both networks.

Step 5: Train the GAN by alternating training of the discriminator and generator. Step 6: Generate and visualize synthetic digit images.

# PROGRAM:

import tensorflow as tf

from tensorflow.keras import layers import matplotlib.pyplot as plt import numpy as np

(train\_images, \_), (\_, \_) = tf.keras.datasets.mnist.load\_data()

train\_images = train\_images.reshape(train\_images.shape[0], 28, 28, 1).astype("float32")

train\_images = (train\_images - 127.5) / 127.5 # Normalize to [-1, 1]

BUFFER\_SIZE = 60000

BATCH\_SIZE = 256

221501039

train\_dataset = tf.data.Dataset.from\_tensor\_slices(train\_images).shuffle(BUFFER\_SIZE).batch(BATCH\_SIZE)

def make\_generator\_model():

model = tf.keras.Sequential([

layers.Dense(7\*7\*256, use\_bias=False, input\_shape=(100,)), layers.BatchNormalization(),

layers.LeakyReLU(), layers.Reshape((7, 7, 256)),

layers.Conv2DTranspose(128, (5,5), strides=(1,1), padding='same', use\_bias=False), layers.BatchNormalization(),

layers.LeakyReLU(),

layers.Conv2DTranspose(64, (5,5), strides=(2,2), padding='same', use\_bias=False), layers.BatchNormalization(),

layers.LeakyReLU(),

layers.Conv2DTranspose(1, (5,5), strides=(2,2), padding='same', use\_bias=False, activation='tanh')

])

return model

def make\_discriminator\_model():

model = tf.keras.Sequential([

layers.Conv2D(64, (5,5), strides=(2,2), padding='same', input\_shape=[28, 28, 1]), layers.LeakyReLU(),

layers.Dropout(0.3),

layers.Conv2D(128, (5,5), strides=(2,2), padding='same'), layers.LeakyReLU(),

layers.Dropout(0.3), layers.Flatten(), layers.Dense(1)

])

return model

cross\_entropy = tf.keras.losses.BinaryCrossentropy(from\_logits=True)

221501039

def discriminator\_loss(real\_output, fake\_output):

return cross\_entropy(tf.ones\_like(real\_output), real\_output) + \ cross\_entropy(tf.zeros\_like(fake\_output), fake\_output)

def generator\_loss(fake\_output):

return cross\_entropy(tf.ones\_like(fake\_output), fake\_output) generator = make\_generator\_model()

discriminator = make\_discriminator\_model() generator\_optimizer = tf.keras.optimizers.Adam(1e-4) discriminator\_optimizer = tf.keras.optimizers.Adam(1e-4) @tf.function

def train\_step(images):

noise = tf.random.normal([BATCH\_SIZE, 100])

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))

EPOCHS = 50

noise\_dim = 100

num\_examples\_to\_generate = 16

seed = tf.random.normal([num\_examples\_to\_generate, noise\_dim]) def generate\_and\_save\_images(model, epoch, test\_input): predictions = model(test\_input, training=False)

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')

221501039

plt.axis('off') plt.show()

def train(dataset, epochs):

for epoch in range(epochs):

for image\_batch in dataset: train\_step(image\_batch)

generate\_and\_save\_images(generator, epoch + 1, seed) train(train\_dataset, EPOCHS)

# OUTPUT:



**RESULT:**

Thus the Program has been executed successfully and verified.