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

EX.N0 : 9	BUILDING A SIMPLE GENERATIVE ADVERSARIAL NETWORK (GAN) USING TENSORFLOW
DATE : 17/04/2025	

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
```

```python
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)
```

```python
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')
```

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:

Generator Model Summary:
Model: "sequential_2"

Layer (type)	Output Shape	Param #
dense_2 (Dense)	(None, 12544)	1,254,400
batch_normalization_3 (BatchNormalization)	(None, 12544)	50,176
leaky_re_lu_5 (LeakyReLU)	(None, 12544)	0
reshape_1 (Reshape)	(None, 7, 7, 256)	0
conv2d_transpose_3 (Conv2DTranspose)	(None, 7, 7, 128)	819,200
batch_normalization_4 (BatchNormalization)	(None, 7, 7, 128)	512
leaky_re_lu_6 (LeakyReLU)	(None, 7, 7, 128)	0
conv2d_transpose_4 (Conv2DTranspose)	(None, 14, 14, 64)	204,800
batch_normalization_5 (BatchNormalization)	(None, 14, 14, 64)	256
leaky_re_lu_7 (LeakyReLU)	(None, 14, 14, 64)	0
conv2d_transpose_5 (Conv2DTranspose)	(None, 28, 28, 1)	1,600

Total params: 2,330,944 (8.89 MB)
Trainable params: 2,305,472 (8.79 MB)
Non-trainable params: 25,472 (99.50 KB)

Discriminator Model Summary:
Model: "sequential_3"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 14, 14, 64)	1,664
leaky_re_lu_8 (LeakyReLU)	(None, 14, 14, 64)	0
dropout_2 (Dropout)	(None, 14, 14, 64)	0
conv2d_3 (Conv2D)	(None, 7, 7, 128)	204,928
leaky_re_lu_9 (LeakyReLU)	(None, 7, 7, 128)	0
dropout_3 (Dropout)	(None, 7, 7, 128)	0
flatten_1 (Flatten)	(None, 6272)	0
dense_3 (Dense)	(None, 1)	6,273

Total params: 212,865 (831.50 KB)
Trainable params: 212,865 (831.50 KB)
Non-trainable params: 0 (0.00 B)

RESULT:

Thus the Program has been executed successfully and verified.