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
```

```
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)
```

```
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:

RESULT:

Thus the Program has been executed successfully and verified.