EX NO: 9	Generative Adversarial Network (GAN) using TensorFlow

Aim:

To implement and train a **Generative Adversarial Network (GAN)** using **TensorFlow** for generating realistic images. In this case, the goal is to generate images that resemble a given dataset (e.g., MNIST digits, Fashion-MNIST, or CIFAR-10).

Algorithm:

1. **Import Libraries**: Import necessary TensorFlow and Keras libraries.

2. **Define the Generator Network**: Create a neural network that generates fake images starting from random noise.

3. **Define the Discriminator Network**: Create a neural network that discriminates between real and fake images.

4. **Define the GAN Model**: Combine the generator and discriminator into a GAN where the generator tries to fool the discriminator.

5. **Train the Model**: Use the adversarial loss function to train the generator and discriminator alternately.

6. **Evaluate the Results**: Visualize the generated images during the training process

Code:

```
import tensorflow as tf

import numpy as np
```

```
import matplotlib.pyplot as plt

from tensorflow.keras import layers, models

import os

tf.random.set_seed(42)

latent_dim = 100

epochs = 50

batch_size = 128

buffer_size = 60000

image_shape = (28, 28, 1)

(x_train, _), (_, _) = tf.keras.datasets.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]

train_dataset = tf.data.Dataset.from_tensor_slices(x_train).shuffle(buffer_size).batch(batch_size)


def build_generator():

    model = models.Sequential([

        layers.Dense(256, input_dim=latent_dim),

        layers.LeakyReLU(alpha=0.2),

        layers.BatchNormalization(momentum=0.8),

        layers.Dense(512),

        layers.LeakyReLU(alpha=0.2),

        layers.BatchNormalization(momentum=0.8),

        layers.Dense(1024),
```

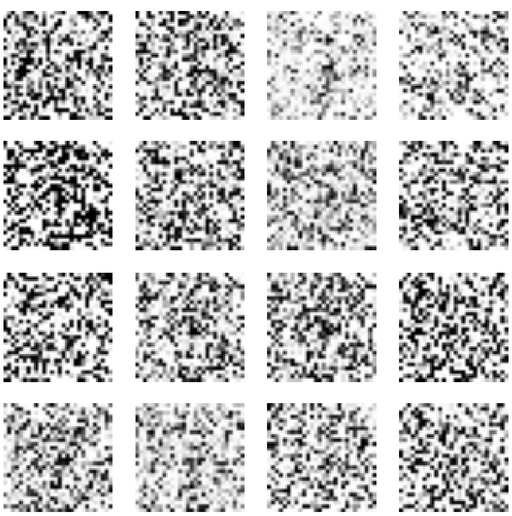
```
        layers.LeakyReLU(alpha=0.2),
        layers.BatchNormalization(momentum=0.8),
        layers.Dense(np.prod(image_shape), activation='tanh'),
        layers.Reshape(image_shape)
    ])
    return model
def build_discriminator():
    model = models.Sequential([
        layers.Flatten(input_shape=image_shape),
        layers.Dense(512),
        layers.LeakyReLU(alpha=0.2),
        layers.Dense(256),
        layers.LeakyReLU(alpha=0.2),
        layers.Dense(1, activation='sigmoid')
    ])
    return model
cross_entropy = tf.keras.losses.BinaryCrossentropy()


def discriminator_loss(real_output, fake_output):
    real_loss =-0.5 * 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 = build_generator()
discriminator = build_discriminator()
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, 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)

    gen_gradients = gen_tape.gradient(gen_loss, generator.trainable_variables)
    disc_gradients = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
```

```
    generator_optimizer.apply_gradients(zip(gen_gradients, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(disc_gradients, discriminator.trainable_variables))

    return gen_loss, disc_loss
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(16):
        plt.subplot(4, 4, i+1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
        plt.axis('off')
    plt.savefig(f'image_at_epoch_{epoch:04d}.png')
    plt.close()
seed = tf.random.normal([16, latent_dim])
for epoch in range(epochs):
    print(f'Epoch {epoch+1}/{epochs}')
    for image_batch in train_dataset:
        gen_loss, disc_loss = train_step(image_batch)
    if (epoch + 1) % 10 == 0:
        generate_and_save_images(generator, epoch + 1, seed)
        print(f'Generator Loss: {gen_loss:.4f}, Discriminator Loss: {disc_loss:.4f}')
generate_and_save_images(generator, epochs, seed)
generator.save('generator_model.h5')
```

```
predictions = generator(seed, training=False)

fig = plt.figure(figsize=(4, 4))

for i in range(16):

    plt.subplot(4, 4, i+1)

    plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')

    plt.axis('off')

plt.show()
```

Output:



Result:

The GAN successfully learns to generate realistic images after multiple epochs, with the generator improving its output and the discriminator refining its ability to distinguish real from fake. Sample images show progressively better results.