EX 10 IMAGE GENERATION USING GENERATIVE ADVERSARIAL NETWORK (GAN)

DATE: 16/10/2025

Problem Statement:

Train a Generative Adversarial Network (GAN) using the CIFAR-10 dataset to generate new synthetic images. Evaluate the generated outputs through visual inspection to understand the training behavior and realism of generated samples.

Objectives:

- ☐ Understand the architecture of a simple Deep Convolutional GAN (DCGAN).
- ☐ Implement Generator and Discriminator networks using TensorFlow and Keras.
- ☐ Train the GAN using adversarial learning principles.
- ☐ Generate new images from random noise vectors.
- ☐ Visually evaluate the quality and diversity of generated images.

Scope:

GANs are powerful models for data generation, capable of synthesizing realistic images after learning from real samples. This experiment provides hands-on experience with adversarial training dynamics and the generator-discriminator framework.

Tools and Libraries Used:

- ☐ Python 3.x
- ☐ TensorFlow / Keras
- ☐ NumPy
- ☐ Matplotlib

Implementation Steps:

Step 1: Load and Preprocess CIFAR-10 Dataset

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

(x_train, _), (_, _) = tf.keras.datasets.cifar10.load_data()
x_train = (x_train.astype("float32") - 127.5) / 127.5
x_train = tf.data.Dataset.from_tensor_slices(x_train).shuffle(60000).batch(128)
```

Step 2: Define the Generator Network

```
def make_generator():
    model = tf.keras.Sequential([
    layers.Dense(8*8*256, use_bias=False, input_shape=(100,)),
    layers.BatchNormalization(),
    layers.LeakyReLU(),
    layers.Reshape((8, 8, 256)),
    layers.Conv2DTranspose(128, (5,5), strides=(2,2), 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(3, (5,5), strides=(1,1), padding='same', use_bias=False,
activation='tanh')
])
return model
```

Step 3: Define the Discriminator Network

```
def make_discriminator():
  model = tf.keras.Sequential([
    layers.Conv2D(64, (5,5), strides=(2,2), padding='same', input_shape=[32,32,3]),
    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
```

Step 4: Initialize Models, Loss, and Optimizers

```
generator = make_generator()
discriminator = make_discriminator()

cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
gen_optimizer = tf.keras.optimizers.Adam(1e-4)
disc_optimizer = tf.keras.optimizers.Adam(1e-4)
```

Step 5: Define Generator and Discriminator Loss Functions

```
def generator_loss(fake_output):
  return cross_entropy(tf.ones_like(fake_output), fake_output)

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
```

Step 6: Define Training Step Function

```
@tf.function
def train_step(images):
    noise = tf.random.normal([128, 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)
    gen_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))
    disc_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))
```

Step 7: Train the GAN

```
EPOCHS = 3
for epoch in range(EPOCHS):
    for image_batch in x_train:
        train_step(image_batch)
    print(f"Epoch {epoch+1}/{EPOCHS} completed.")
```

Step 8: Generate and Visualize New Images

```
noise = tf.random.normal([16, 100])
generated_images = generator(noise, training=False)

plt.figure(figsize=(8,8))
for i in range(16):
    plt.subplot(4,4,i+1)
    plt.imshow((generated_images[i] + 1) / 2)
    plt.axis('off')
plt.show()
```

Output: