Ex No: 9

BUILD GENERATIVE ADVERSARIAL NEURAL NETWORK

Aim:

To build a generative adversarial neural network using Keras/TensorFlow.

Procedure:

1. Load and preprocess the dataset by importing the CIFAR-10 dataset, selecting images of a specific class, and normalizing the pixel values to a range of -1 to 1.
2. Define the generator model, a neural network that takes random noise as input and generates images using dense layers, reshaping, upsampling, convolutional layers, batch normalization, and activation functions.
3. Define the discriminator model, a convolutional neural network to classify whether an image is real or fake, with layers such as convolutional layers, dropout, batch normalization, and a final dense layer with a sigmoid activation.
4. Compile the discriminator model using binary cross-entropy loss and an optimizer to help it learn to differentiate real images from generated ones.
5. Combine the generator and discriminator by freezing the discriminator's weights and creating a single model where the generator produces images classified as real by the discriminator.
6. Set training parameters, including the number of epochs, batch size, and intervals for displaying generated images. Define ground truth labels for real and fake images, adding small noise for robustness.
7. Train the discriminator by sampling real images and generating fake ones with the generator, then training the discriminator on these to classify them more accurately.
8. Train the generator by feeding random noise into it and updating its weights through the combined model to produce images that the discriminator classifies as real.
9. Monitor training progress by tracking generator and discriminator losses and displaying generated images at regular intervals to observe improvement.
10. Use the trained generator to produce new images from random noise, scale them to a range of 0 to 1 for visualization, and display them in a grid.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
import keras
from keras.layers import Input, Dense, Reshape, Flatten, Dropout
from keras.layers import BatchNormalization, Activation, ZeroPadding2D
from tensorflow.keras.layers import LeakyReLU
from tensorflow.keras.layers import UpSampling2D, Conv2D
from keras.models import Sequential, Model
from keras.optimizers import Adam,SGD
#Loading the CIFAR10 data
(X, y), (_, _) = keras.datasets.cifar10.load_data()
#Selecting a single class images
#The number was randomly chosen and any number
#between 1 to 10 can be chosen
X = X[y.flatten() == 8]
#Defining the Input shape
image_shape = (32, 32, 3)
latent_dimensions = 100
def build_generator():
  model = Sequential()
  #Building the input layer
  model.add(Dense(128 * 8 * 8, activation="relu",
      input_dim=latent_dimensions))
  model.add(Reshape((8, 8, 128)))
  model.add(UpSampling2D())
  model.add(Conv2D(128, kernel_size=3, padding="same"))
  model.add(BatchNormalization(momentum=0.78))
  model.add(Activation("relu"))
  model.add(UpSampling2D())
```

```
  model.add(Conv2D(64, kernel_size=3, padding="same"))
  model.add(BatchNormalization(momentum=0.78))
  model.add(Activation("relu"))
  model.add(Conv2D(3, kernel_size=3, padding="same"))
  model.add(Activation("tanh"))
  #Generating the output image
  noise = Input(shape=(latent_dimensions,))
  image = model(noise)
  return Model(noise, image)
def build_discriminator():
  #Building the convolutional layers
  #to classify whether an image is real or fake
  model = Sequential()
  model.add(Conv2D(32, kernel_size=3, strides=2,
      input_shape=image_shape, padding="same"))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.25))

  model.add(Conv2D(64, kernel_size=3, strides=2, padding="same"))
  model.add(ZeroPadding2D(padding=((0,1),(0,1))))
  model.add(BatchNormalization(momentum=0.82))
  model.add(LeakyReLU(alpha=0.25))
  model.add(Dropout(0.25))

  model.add(Conv2D(128, kernel_size=3, strides=2, padding="same"))
  model.add(BatchNormalization(momentum=0.82))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.25))

  model.add(Conv2D(256, kernel_size=3, strides=1, padding="same"))
```

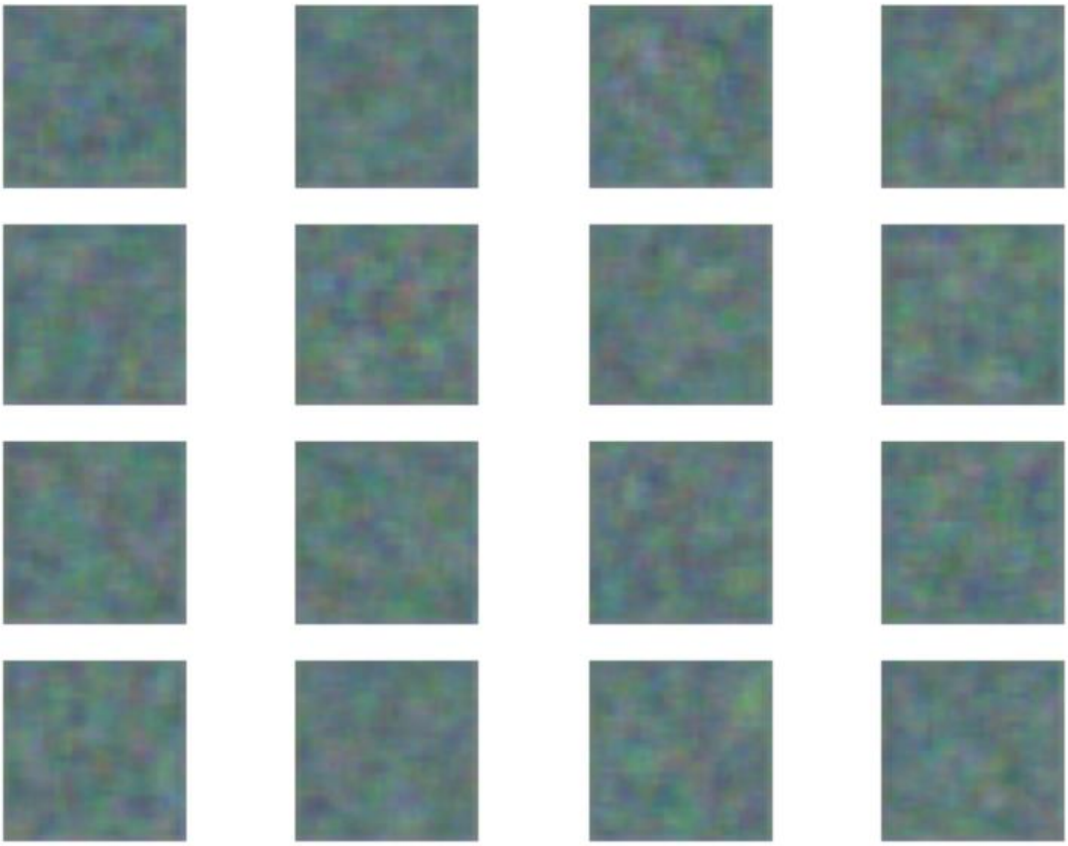
```
  model.add(BatchNormalization(momentum=0.8))
  model.add(LeakyReLU(alpha=0.25))
  model.add(Dropout(0.25))

  #Building the output layer
  model.add(Flatten())
  model.add(Dense(1, activation='sigmoid'))
  image = Input(shape=image_shape)
  validity = model(image)
  return Model(image, validity)

def display_images():
  r, c = 4,4
  noise = np.random.normal(0, 1, (r * c,latent_dimensions))
  generated_images = generator.predict(noise)
  #Scaling the generated images
  generated_images = 0.5 * generated_images + 0.5
  fig, axs = plt.subplots(r, c)
  count = 0
  for i in range(r):
   for j in range(c):
    axs[i,j].imshow(generated_images[count, :,:,])
    axs[i,j].axis('off')
    count += 1
  plt.show()
  plt.close()

# Building and compiling the discriminator
discriminator = build_discriminator()
discriminator.compile(loss='binary_crossentropy',
```

```
     optimizer=Adam(0.0002,0.5),
     metrics=['accuracy'])

#Making the Discriminator untrainable
#so that the generator can learn from fixed gradient
discriminator.trainable = False
# Building the generator
generator = build_generator()

#Defining the input for the generator
#and generating the images
z = Input(shape=(latent_dimensions,))
image = generator(z)
#Checking the validity of the generated image
valid = discriminator(image)

#Defining the combined model of the Generator and the Discriminator
combined_network = Model(z, valid)
combined_network.compile(loss='binary_crossentropy',
     optimizer=Adam(0.0002,0.5))
num_epochs=10
batch_size=32
display_interval=5
losses=[]
#Normalizing the input
X = (X / 127.5) - 1.
#Defining the Adversarial ground truths
valid = np.ones((batch_size, 1))

#Adding some noise
```

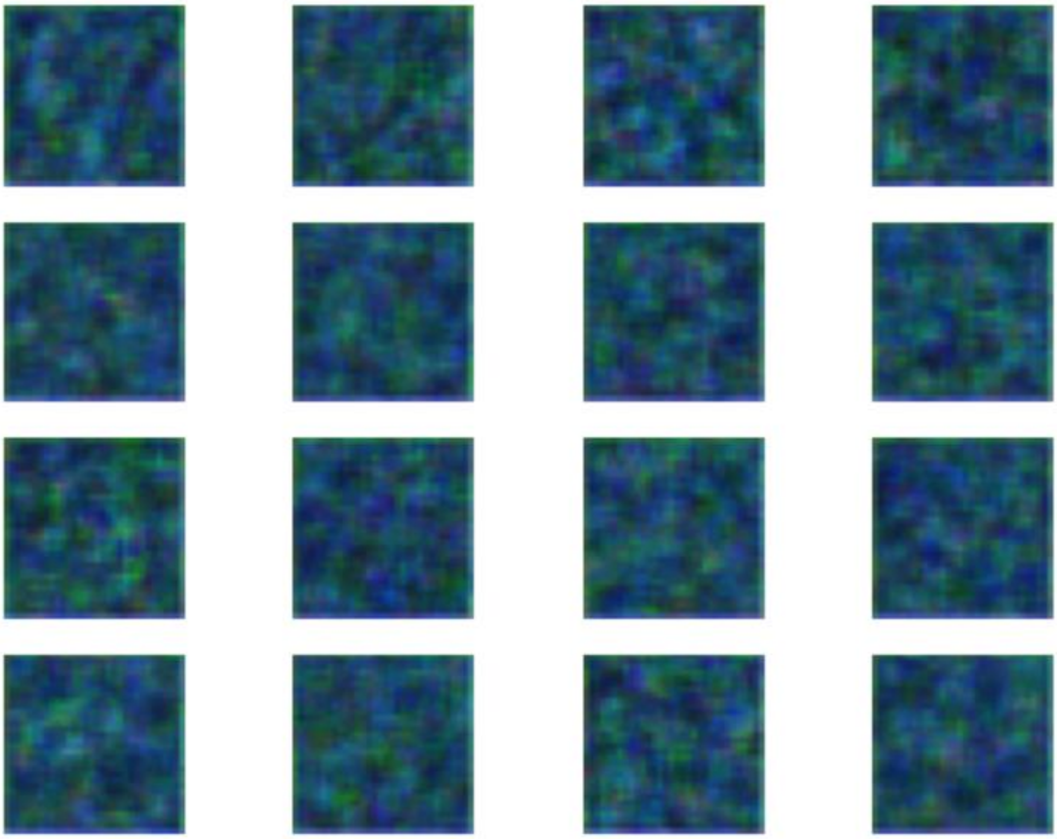
```
valid += 0.05 * np.random.random(valid.shape)
fake = np.zeros((batch_size, 1))
fake += 0.05 * np.random.random(fake.shape)
for epoch in range(num_epochs):
  #Training the Discriminator , Sampling a random half of images
  index = np.random.randint(0, X.shape[0], batch_size)
  images = X[index]
  #Sampling noise and generating a batch of new images
  noise = np.random.normal(0, 1, (batch_size, latent_dimensions))
  generated_images = generator.predict(noise)
  #Training the discriminator to detect more accurately
  #whether a generated image is real or fake
  discm_loss_real = discriminator.train_on_batch(images, valid)
  discm_loss_fake = discriminator.train_on_batch(generated_images, fake)
  discm_loss = 0.5 * np.add(discm_loss_real, discm_loss_fake)
  genr_loss = combined_network.train_on_batch(noise, valid)
  #Tracking the progress
  if epoch % display_interval == 0:
   display_images()
```

Output:

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 387ms/step
```



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 96ms/step
WARNING:tensorflow:5 out of the last 5 calls to <function TensorFlowTrainer.make_train_function.<locals>.one_step_on_iterator at 0x000001F7F18562A0> tri
gered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop,
(2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop.
or (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/fu
ction#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for  more details.
WARNING:tensorflow:6 out of the last 6 calls to <function TensorFlowTrainer.make_train_function.<locals>.one_step_on_iterator at 0x000001F780030860> tri
gered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop,
(2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop.
or (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/fu
ction#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for  more details.
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 94ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 108ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 85ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 80ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 67ms/step
```



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 99ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 90ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 87ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step
```

Result:

Thus the program to build generative adversarial neural network with Keras/TensorFlow has been executed successfully.