Experiment: 5

Aim: To implement a GAN for generating realistic data by training a generator and discriminator in a competitive framework.

Theory:

GANs are machine learning models used for generating data similar to a given dataset. They consist of two neural networks:

Generator: Creates data from random noise, aiming to produce outputs that look real. Discriminator: Distinguishes between real data and generated (fake) data, providing feedback to improve the generator. The training process involves a game where the generator tries to "fool" the discriminator, and the discriminator aims to correctly classify real vs. fake data. This iterative process helps GANs generate realistic images, audio, or other types of data.

Importing Necessary libraries

In [ ]:
```
from tensorflow.keras.layers import (Dense,
                                     BatchNormalization,
                                     LeakyReLU,
                                     Reshape,
                                     Conv2DTranspose,
                                     Conv2D,
                                     Dropout,
                                     Flatten)
import tensorflow as tf
import matplotlib.pyplot as plt
```

Preparing the data

In [ ]:
```
# underscore to omit the label arrays
(train_images, train_labels), (_, _) = 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 the images to [-1, 1]

BUFFER_SIZE = 60000
BATCH_SIZE = 256

# Batch and shuffle the data
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch
```

```
2024-07-08 08:27:01.770745: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1926] Create
d device /job:localhost/replica:0/task:0/device:GPU:0 with 17947 MB memory:  -> device: 0,
name: NVIDIA A100-SXM4-40GB MIG 3g.20gb, pci bus id: 0000:b7:00.0, compute capability: 8.0
```

Dimension of the Noise

In [ ]:
```
# Set the dimensions of the noise
z_dim = 100
```

Functional Code Generator

In [ ]:
```
# nch = 200
# g_input = Input(shape=[100])
# H = Dense(nch*14*14, kernel_initializer='glorot_normal')(g_input)
# H = BatchNormalization()(H)
# H = Activation('relu')(H)
# H = Reshape( [nch, 14, 14] )(H)
# H = UpSampling2D(size=(2, 2))(H)
# H = Convolution2D(int(nch/2), 3, 3, padding='same', kernel_initializer='glorot_uniform')(H
# H = BatchNormalization()(H)
# H = Activation('relu')(H)
# H = Convolution2D(int(nch/4), 3, 3, padding='same', kernel_initializer='glorot_uniform')(H
# H = BatchNormalization()(H)
# H = Activation('relu')(H)
# H = Convolution2D(1, 1, 1, padding='same', kernel_initializer='glorot_uniform')(H)
# g_V = Activation('sigmoid')(H)
# generator = Model(g_input,g_V)
# generator.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0002, beta_1=0.5))
# generator.summary()
```

Building Generator

In [ ]:
```
def generator_model():
    model = tf.keras.Sequential()
    model.add(Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(BatchNormalization())
    model.add(LeakyReLU())

    model.add(Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch size

    model.add(Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(BatchNormalization())
    model.add(LeakyReLU())

    model.add(Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(BatchNormalization())
    model.add(LeakyReLU())

    model.add(Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, ac
    assert model.output_shape == (None, 28, 28, 1)

    print(model.summary())

    return model
```

In [ ]:
```
generator = generator_model()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do
not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, pr
efer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 12544)	1,254,400
batch_normalization (BatchNormalization)	(None, 12544)	50,176
leaky_re_lu (LeakyReLU)	(None, 12544)	0
reshape (Reshape)	(None, 7, 7, 256)	0
conv2d_transpose (Conv2DTranspose)	(None, 7, 7, 128)	819,200
batch_normalization_1 (BatchNormalization)	(None, 7, 7, 128)	512
leaky_re_lu_1 (LeakyReLU)	(None, 7, 7, 128)	0
conv2d_transpose_1 (Conv2DTranspose)	(None, 14, 14, 64)	204,800
batch_normalization_2 (BatchNormalization)	(None, 14, 14, 64)	256
leaky_re_lu_2 (LeakyReLU)	(None, 14, 14, 64)	0
conv2d_transpose_2 (Conv2DTranspose)	(None, 28, 28, 1)	1,600

Total params: 2,330,944 (8.89 MB)

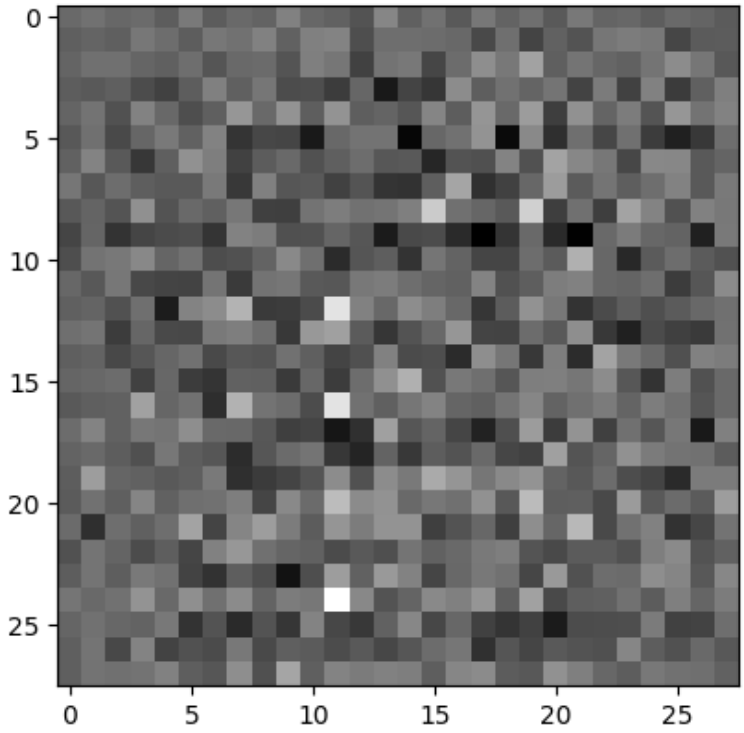
Trainable params: 2,305,472 (8.79 MB)

Non-trainable params: 25,472 (99.50 KB)

Generate a Sample Image

In [ ]:
```
# Create a random noise and generate a sample
noise = tf.random.normal([1, 100])
generated_image = generator(noise, training=False)
# Visualize the generated sample
plt.imshow(generated_image[0, :, :, 0], cmap='gray')
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7f476254b310>
```



Discriminative Model

In [ ]:
```
def discriminator_model():
    model = tf.keras.Sequential()

    model.add(Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=[28, 28, 1]))
    model.add(LeakyReLU())
    model.add(Dropout(0.3))

    model.add(Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(LeakyReLU())
    model.add(Dropout(0.3))

    model.add(Flatten())
    model.add(Dense(1))

    print(model.summary())

    return model
```

In [ ]:
```
discriminator = discriminator_model()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 14, 14, 64)	1,664
leaky_re_lu_3 (LeakyReLU)	(None, 14, 14, 64)	0
dropout (Dropout)	(None, 14, 14, 64)	0
conv2d_1 (Conv2D)	(None, 7, 7, 128)	204,928
leaky_re_lu_4 (LeakyReLU)	(None, 7, 7, 128)	0
dropout_1 (Dropout)	(None, 7, 7, 128)	0
flatten (Flatten)	(None, 6272)	0
dense_1 (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)

Configure the Model

In [ ]:
```
# This method returns a helper function to compute cross entropy loss
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

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)
    total_loss = real_loss + fake_loss
    return total_loss

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

generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

Training

In [ ]:
```
import os
checkpoint_dir = '/content/drive/MyDrive/AMITY/Deep Learning (codes)/GAN/'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)
```

In [ ]:
```
EPOCHS = 100
# We will reuse this seed overtime (so it's easier)
# to visualize progress in the animated GIF)
num_examples_to_generate = 16
noise_dim = 100
seed = tf.random.normal([num_examples_to_generate, noise_dim])
```

Training Steps

In [ ]:
```
# tf.function annotation causes the function
# to be "compiled" as part of the training
@tf.function
def train_step(images):

    # 1 - Create a random noise to feed it into the model
    # for the image generation
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    # 2 - Generate images and calculate loss values
    # GradientTape method records operations for automatic differentiation.
    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)

    # 3 - Calculate gradients using loss values and model variables
    # "gradient" method computes the gradient using
    # operations recorded in context of this tape (gen_tape and disc_tape).

    # It accepts a target (e.g., gen_loss) variable and
    # a source variable (e.g.,generator.trainable_variables)
    # target --> a list or nested structure of Tensors or Variables to be differentiated.
    # source --> a list or nested structure of Tensors or Variables.
    # target will be differentiated against elements in sources.

    # "gradient" method returns a list or nested structure of Tensors
    # (or IndexedSlices, or None), one for each element in sources.
    # Returned structure is the same as the structure of sources.
    gradients_of_generator = gen_tape.gradient(gen_loss,
                                               generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
                                                discriminator.trainable_variables)

    # 4 - Process  Gradients and Run the Optimizer
    # "apply_gradients" method processes aggregated gradients.
    # ex: optimizer.apply_gradients(zip(grads, vars))
    """
    Example use of apply_gradients:
    grads = tape.gradient(loss, vars)
    grads = tf.distribute.get_replica_context().all_reduce('sum', grads)
    # Processing aggregated gradients.
    optimizer.apply_gradients(zip(grads, vars), experimental_aggregate_gradients=False)
    """
    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_var
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.t
```

Image Generation Function

In [ ]:
```
def generate_and_save_images(model, epoch, test_input):
  # Notice `training` is set to False.
  # This is so all layers run in inference mode (batchnorm).
  # 1 - Generate images
  predictions = model(test_input, training=False)
  # 2 - Plot the generated images
  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')
  # 3 - Save the generated images
```

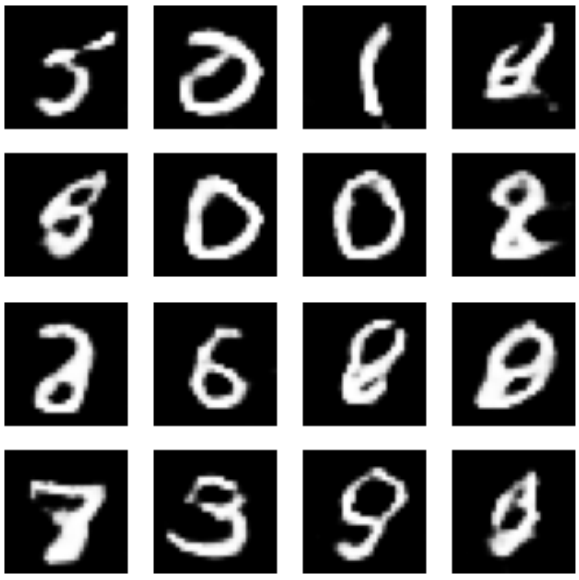
```
  plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
  plt.show()
```

Train GAN

In [ ]:
```
import time
from IPython import display # A command shell for interactive computing in Python.

def train(dataset, epochs):
  # A. For each epoch, do the following:
  for epoch in range(epochs):
    start = time.time()
    # 1 - For each batch of the epoch,
    for image_batch in dataset:
      # 1.a - run the custom "train_step" function
      # we just declared above
      train_step(image_batch)

    # 2 - Produce images for the GIF as we go
    display.clear_output(wait=True)
    generate_and_save_images(generator,
                             epoch + 1,
                             seed)

    # 3 - Save the model every 5 epochs as
    # a checkpoint, which we will use later
    if (epoch + 1) % 5 == 0:
      checkpoint.save(file_prefix = checkpoint_prefix)

    # 4 - Print out the completed epoch no. and the time spent
    print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

  # B. Generate a final image after the training is completed
  display.clear_output(wait=True)
  generate_and_save_images(generator,
                           epochs,
                           seed)
```

Start Training

In [ ]:
```
train(train_dataset, EPOCHS)
```



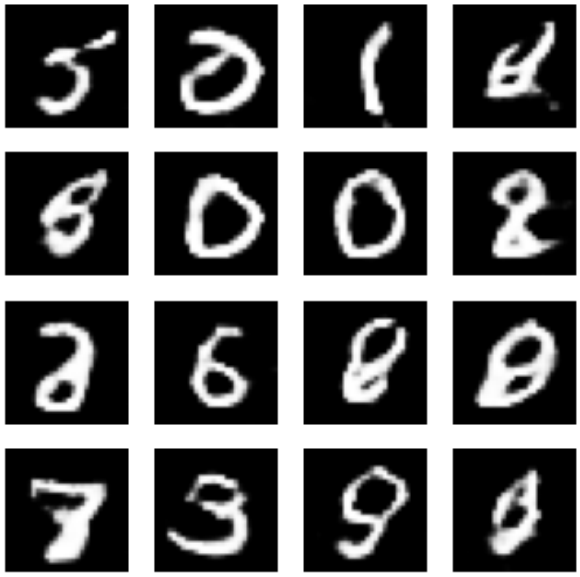
Generated Digits

In [ ]:
```
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
```

Out[ ]:
```
<tensorflow.python.checkpoint.checkpoint.CheckpointLoadStatus at 0x7fe1513039a0>
```

In [ ]:
```
# PIL is a library which may open different image file formats
import PIL
# Display a single image using the epoch number
def display_image(epoch_no):
  return PIL.Image.open('image_at_epoch_{:04d}.png'.format(epoch_no))
display_image(EPOCHS)
```

Out[ ]:



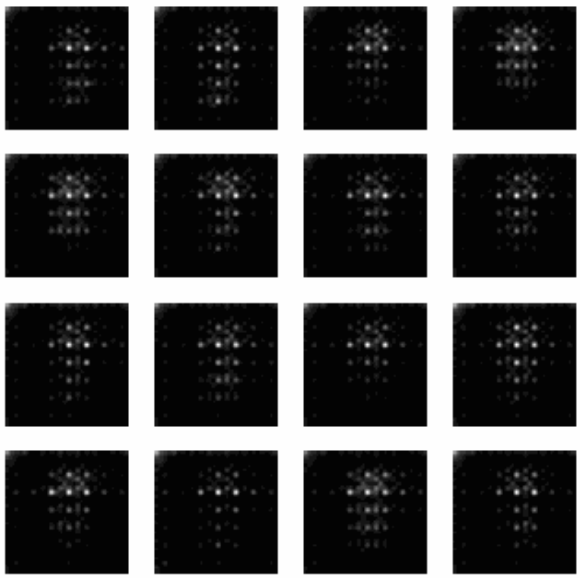
In [ ]:
```
pip install imageio
```

```
Requirement already satisfied: imageio in /usr/local/lib/python3.10/dist-packages (2.34.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from image
io) (1.24.4)
Requirement already satisfied: pillow>=8.3.2 in /usr/local/lib/python3.10/dist-packages (fr
om imageio) (10.2.0)
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting be
haviour with the system package manager. It is recommended to use a virtual environment ins
tead: https://pip.pypa.io/warnings/venv
Note: you may need to restart the kernel to use updated packages.
```

In [ ]:
```
import glob # The glob module is used for Unix style pathname pattern expansion.
import imageio # The library that provides an easy interface to read and write a wide range

anim_file = 'dcgan.gif'

with imageio.get_writer(anim_file, mode='I') as writer:
  filenames = glob.glob('image*.png')
  filenames = sorted(filenames)
  for filename in filenames:
    image = imageio.imread(filename)
    writer.append_data(image)
  # image = imageio.imread(filename)
  # writer.append_data(image)

display.Image(open('dcgan.gif','rb').read())
```

```
/tmp/ipykernel_1192389/2115569125.py:10: DeprecationWarning: Starting with ImageIO v3 the b
ehavior of this function will switch to that of iio.v3.imread. To keep the current behavior
(and make this warning disappear) use `import imageio.v2 as imageio` or call `imageio.v2.im
read` directly.
  image = imageio.imread(filename)
```

Out[ ]:



In [ ]: