EXPERIMENT 9 : DCGAN FOR MNIST HAND WRITTEN NUMBER DATASET

1 AIM :

To implement a Deep Convolution Generative Adversarial Network (DCGAN) for generating realistic images resembling the MNIST dataset.

OBJECTIVE:

The objective is to train a generator network to create handwritten text images that are indistinguishable from real MNIST images, while simultaneously training a discriminator network to correctly classify real and handwritten text images.

METHODOLOGY:

1. **Data Preparation**:
    - Load the MNIST dataset and preprocess the images to normalize them between -1 and 1.
2. **Generator Network**:
    - Build a generator model using convolutional and transpose convolutional layers.

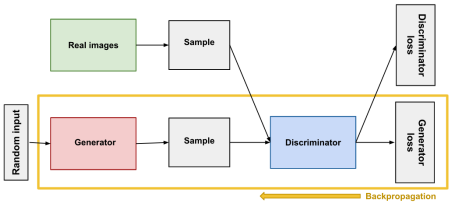


Figure 1: DCGAN ARCHITECTURE

- The generator takes random noise as input and produces handwritten text images.

3. **Discriminator Network**:
   - Construct a discriminator model that discriminates between real and handwritten text images.
   - It is trained to output high values for real images and low values for fake ones.
4. **Loss Functions**:
   - Define the loss functions for both the generator and discriminator.
   - Use binary cross-entropy loss to train the discriminator to distinguish real and fake images.
   - Train the generator to fool the discriminator by maximizing its output.
5. **Training**:
   - Train the generator and discriminator networks simultaneously in alternating steps.
   - Update the weights of both networks using Adam optimizer.
6. **Evaluation**:
   - Generate handwritten text images at regular intervals during training to visualize the learning progress.
   - Save checkpoints of the models for later use.
7. **Results**:
   - Display the final generated images and create an animated GIF to visualize the training progress.

THEORY:

The DCGAN architecture consists of two neural networks: the generator and the discriminator. The generator takes random noise as input and learns to generate realistic images, while the discriminator learns to classify between real and fake images. Through adversarial training, both networks improve over time, with the generator learning to produce more realistic images and the discriminator becoming better at distinguishing real from fake.

The generator is trained to minimize the probability that the discriminator correctly classifies its outputs as fake. Conversely, the discriminator is trained to maximize this probability. This adversarial process leads to a Nash equilibrium

where the generator produces realistic images and the discriminator is unable to differentiate between real and fake with high confidence.

1. Generator Network:

Table 1: Generator Model

Layer	Output Shape
Dense	(None, 7*7*256)
BatchNormalization	(None, 7*7*256)
LeakyReLU	(None, 7*7*256)
Reshape	(None, 7, 7, 256)
Conv2DTranspose	(None, 7, 7, 128)
BatchNormalization	(None, 7, 7, 128)
LeakyReLU	(None, 7, 7, 128)
Conv2DTranspose	(None, 14, 14, 64)
BatchNormalization	(None, 14, 14, 64)
LeakyReLU	(None, 14, 14, 64)
Conv2DTranspose	(None, 28, 28, 1)

The generator network takes a random noise vector z as input and generates fake images.

The input noise vector z is typically drawn from a simple distribution like a Gaussian distribution.

The generator produces a fake image $G(z)$.

The generator network can be represented as a function $G(z; \theta_g)$, where θ_g are the parameters of the generator network.

The loss function for the generator is typically the binary cross-entropy loss, which measures how well the generator fools the discriminator.

So, the generator's objective is to minimize the following loss function:

$$\min_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(z^{(i)})))$$

where:

- m is the batch size,
- $z^{(i)}$ is the noise vector for the i-th sample,
- $G(z^{(i)})$ is the generated fake image for the i-th sample,
- $D(G(z^{(i)}))$ is the discriminator's output when given the fake image as input.

2. Discriminator Network:

Table 2: Discriminator Model

Layer	Output Shape
Conv2D	(None, 14, 14, 64)
LeakyReLU	(None, 14, 14, 64)
Dropout	(None, 14, 14, 64)
Conv2D	(None, 7, 7, 128)
LeakyReLU	(None, 7, 7, 128)
Dropout	(None, 7, 7, 128)
Flatten	(None, 6272)
Dense	(None, 1)

The discriminator network takes an image x as input and predicts whether it is real or fake.

The discriminator produces a probability $D(x)$ representing the likelihood that the input image is real.

The discriminator network can be represented as a function $D(x; \theta_d)$, where θ_d are the parameters of the discriminator network.

The loss function for the discriminator is typically the binary cross-entropy loss, which measures how well the discriminator distinguishes between real and fake images.

So, the discriminator's objective is to minimize the following loss function:

$$\min_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D(x^{(i)}) + \log(1 - D(G(z^{(i)}))) \right]$$

where:

- $x^{(i)}$ is a real image sample,
- $D(x^{(i)})$ is the discriminator's output when given the real image as input.

This loss function is a combination of two terms: the first term encourages the discriminator to correctly classify real images as real, and the second term encourages it to correctly classify fake images as fake.

Discriminator Loss:

discriminator_loss = binary_crossentropy(ones, real_output) + binary_crossentropy(zeros, fake_output)

Generator Loss:

generator_loss = binary_crossentropy(ones, fake_output)

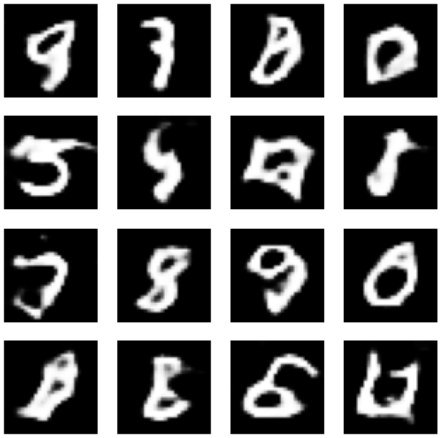


Figure 2: handwritten image generation

RESULT:

The DCGAN successfully generates realistic images resembling handwritten digits from the MNIST dataset. The generated images improve in quality as the training progresses, eventually becoming visually indistinguishable from real MNIST digits.

CONCLUSION:

The implementation of the Deep Convolutional Generative Adversarial Network (DCGAN) demonstrates the effectiveness of adversarial training in generating realistic images. By training the generator and discriminator networks in tandem, the model learns to produce high-quality C text images, showing potential applications in image generation and data augmentation tasks.

Deep Convolutional Generative Adversarial Network

In [ ]:
```
import glob
import imageio
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
from tensorflow.keras import layers
import time
import tensorflow as tf
from IPython import display
```

In [ ]:
```
(train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dataset
s/mnist.npz
11490434/11490434 [==============================] - 2s 0us/step
```

In [ ]:
```
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype('fl
train_images = (train_images - 127.5) / 127.5  # Normalize the images to [-1, 1]
```

In [ ]:
```
BUFFER_SIZE = 60000
BATCH_SIZE = 256
```

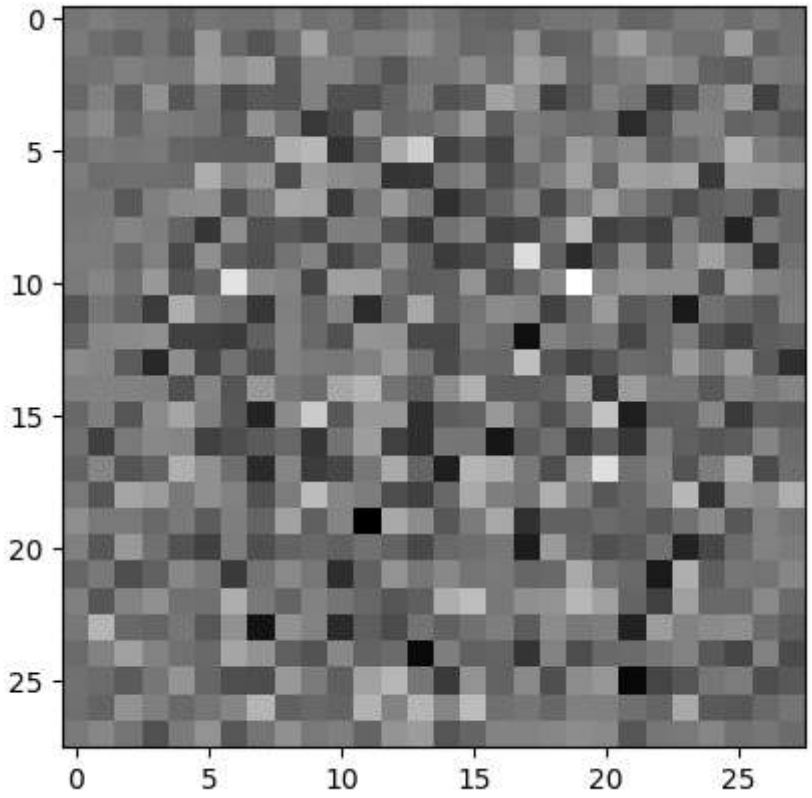
In [ ]:
```
# Batch and shuffle the data
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_
```

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

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

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

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

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

    return model
```

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

```
noise = tf.random.normal([1, 100])
generated_image = generator(noise, training=False)

plt.imshow(generated_image[0, :, :, 0], cmap='gray')
```

Out[ ]: <matplotlib.image.AxesImage at 0x794d914bc6a0>



In [ ]:
```
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                                     input_shape=[28, 28, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

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

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

    return model
```

In [ ]:
```
discriminator = make_discriminator_model()
decision = discriminator(generated_image)
print (decision)
```

```
tf.Tensor([[0.00107551]], shape=(1, 1), dtype=float32)
```

In [ ]:
```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

In [ ]:
```
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
```

In [ ]:
```
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)
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer
                                 generator=generator,
                                 discriminator=discriminator)
```

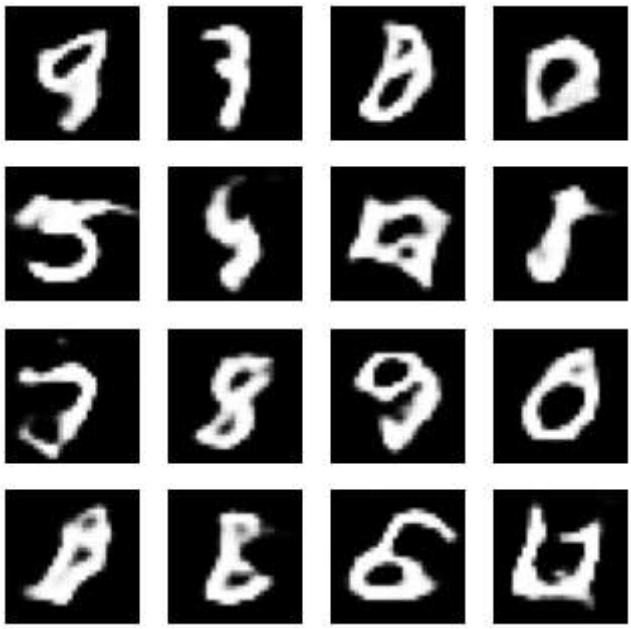
In [ ]:
```
EPOCHS = 50
noise_dim = 100
num_examples_to_generate = 16

seed = tf.random.normal([num_examples_to_generate, noise_dim])

# Notice the use of `tf.function`
# This annotation causes the function to be "compiled".
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_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)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_var
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.tra

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.tr
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, disc
```

In [ ]:
```
def train(dataset, epochs):
  for epoch in range(epochs):
    start = time.time()

    for image_batch in dataset:
      train_step(image_batch)

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

    # Save the model every 15 epochs
    if (epoch + 1) % 15 == 0:
      checkpoint.save(file_prefix = checkpoint_prefix)

    print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))
```

```
  # Generate after the final epoch
  display.clear_output(wait=True)
  generate_and_save_images(generator,
                           epochs,
                           seed)
```

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

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



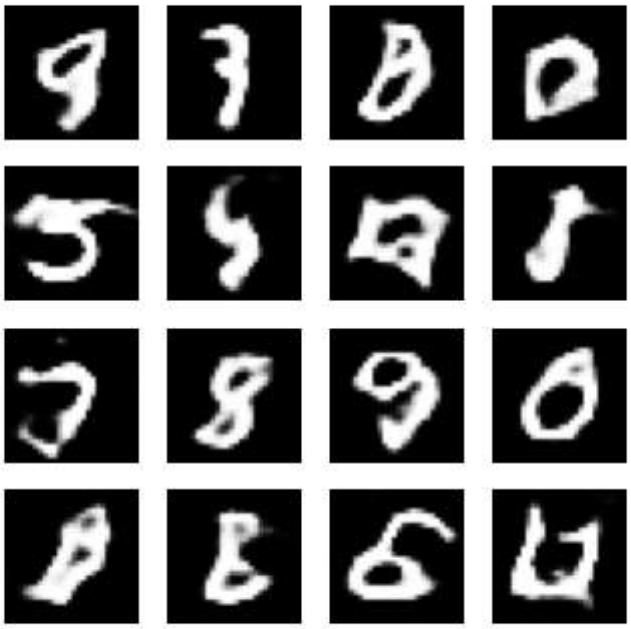
In [ ]:
```
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
```

Out[ ]:
```
<tensorflow.python.checkpoint.checkpoint.CheckpointLoadStatus at 0x794d8a10dff0
>
```

In [ ]:
```
# 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))
```

In [ ]:
```
display_image(EPOCHS)
```

Out[ ]:



In [ ]:
```
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)
```