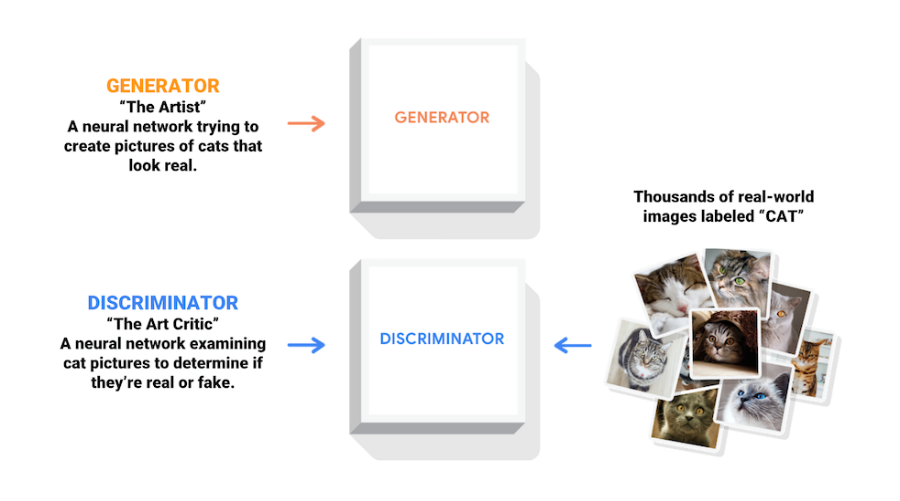
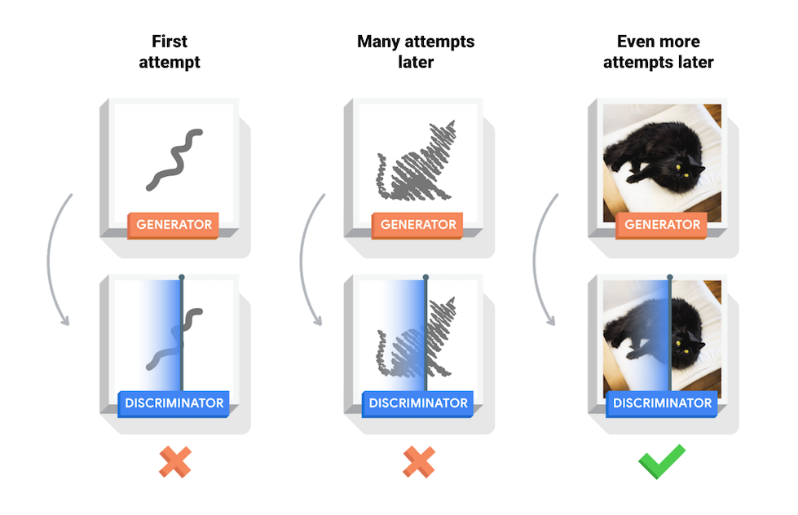
|  |  |
| --- | --- |
| **Ex No: 7**  **Date: 18/09/2024** | **Lab Record: Generative Adversarial Networks (GANs)** |

**Objective:**

The objective of this lab is to implement and understand a Deep Convolutional Generative Adversarial Network (DCGAN) using TensorFlow and Keras to generate images of handwritten digits similar to those in the MNIST dataset. We will explain the main code snippets line by line to understand how each component contributes to the overall functionality.

**Description:**

Generative Adversarial Networks (GANs) are a type of neural network architecture consisting of two models:

1. **Generator**: Generates new data (e.g., images) based on random noise.
2. **Discriminator**: Tries to distinguish between real data and the data generated by the Generator.

The **Generator** and **Discriminator** are trained together in an adversarial process, where the Generator learns to generate increasingly realistic data to fool the Discriminator, and the Discriminator gets better at distinguishing real data from fakes.

**Code Line by Line Explanation:**

**Setup**

- TensorFlow: Deep learning framework used to build, train, and evaluate models.

- NumPy: For handling data arrays.

- Matplotlib: For visualizing the results.

**Loading the MNIST Dataset:**

(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

train\_dataset = tf.data.Dataset.from\_tensor\_slices(train\_images).shuffle(BUFFER\_SIZE).batch(BATCH\_SIZE)

- Loading MNIST: The dataset contains 60,000 images of handwritten digits.

- Reshape & Normalize: Each image is reshaped to 28x28x1 (grayscale) and normalized to the range [-1, 1].

- Batching: The dataset is shuffled and divided into batches for training.

**Building the Generator**

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)))

model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use\_bias=False))

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU())

model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use\_bias=False))

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU())

model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use\_bias=False, activation='tanh'))

return model

- Dense Layer: First, a fully connected layer reshapes the input noise (100 random values) into a 7x7x256 array.

- Conv2DTranspose: Upsamples the image to 28x28x1 (size of MNIST digits) using transposed convolutional layers.

- BatchNormalization: Normalizes the outputs for faster convergence.

- LeakyReLU Activation: Provides non-linearity, helping the network learn complex patterns.

- Tanh Activation: The final layer uses `tanh` because the output images are normalized between [-1, 1].

**Building the Discriminator**

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

- Conv2D Layers: Extracts features from the image by downsampling using 2D convolution.

- LeakyReLU Activation: Allows some gradients to pass even for negative inputs, improving learning.

- Dropout: Helps prevent overfitting by randomly turning off neurons during training.

- Flatten: Converts the 2D feature maps to a 1D vector.

- Dense Layer: Outputs a single scalar indicating whether the input is real or fake.

**Loss Functions**

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)

return real\_loss + fake\_loss

def generator\_loss(fake\_output):

return cross\_entropy(tf.ones\_like(fake\_output), fake\_output)

- Discriminator Loss: Compares the discriminator’s prediction for real images to 1 and fake images to 0.

- Generator Loss: Compares the discriminator's prediction for generated (fake) images to 1, as the goal is to trick the discriminator.

**Optimizers**

generator\_optimizer = tf.keras.optimizers.Adam(1e-4)

discriminator\_optimizer = tf.keras.optimizers.Adam(1e-4)

- Adam Optimizer: Used to optimize both the generator and discriminator.

**Training Loop**

@tf.function

def train\_step(images):

noise = tf.random.normal([BATCH\_SIZE, 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)

generator\_optimizer.apply\_gradients(zip(gradients\_of\_generator, generator.trainable\_variables))

discriminator\_optimizer.apply\_gradients(zip(gradients\_of\_discriminator, discriminator.trainable\_variables))

- train\_step: This function performs a single step of training, updating both the generator and the discriminator. Gradients are calculated and applied using `GradientTape`.

**Training the Model**

def train(dataset, epochs):

for epoch in range(epochs):

start = time.time()

for image\_batch in dataset:

train\_step(image\_batch)

# Output the progress of the model every epoch

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

- train: Trains the model for a specified number of epochs. It iterates over the dataset, calling `train\_step` for each batch of images.

**Generating and Saving Images**

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(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()

- generate\_and\_save\_images: This function generates and saves images during training to visualize the progress.

After running the training for 50 epochs, the generator is able to produce images that closely resemble the handwritten digits from the MNIST dataset.

This lab demonstrated how to implement and train a DCGAN to generate realistic images of handwritten digits. Key concepts covered include:

- Defining a Generator and Discriminator model.

- Implementing loss functions and optimizers.

- Creating a training loop to progressively improve the performance of the models.

GANs are a powerful tool for generating new data, and this notebook shows their potential in image generation tasks.

**GitHub Link:** https://github.com/dyuthiramesh/Deep\_Learning\_Elective/blob/main/Sem5/Lab7/dcgan\_DISTRI.ipynb