|  |  |
| --- | --- |
| **Ex No: 7**  **Date: 18-9-2024** | **Lab 7: GANs** |

**Objective:**

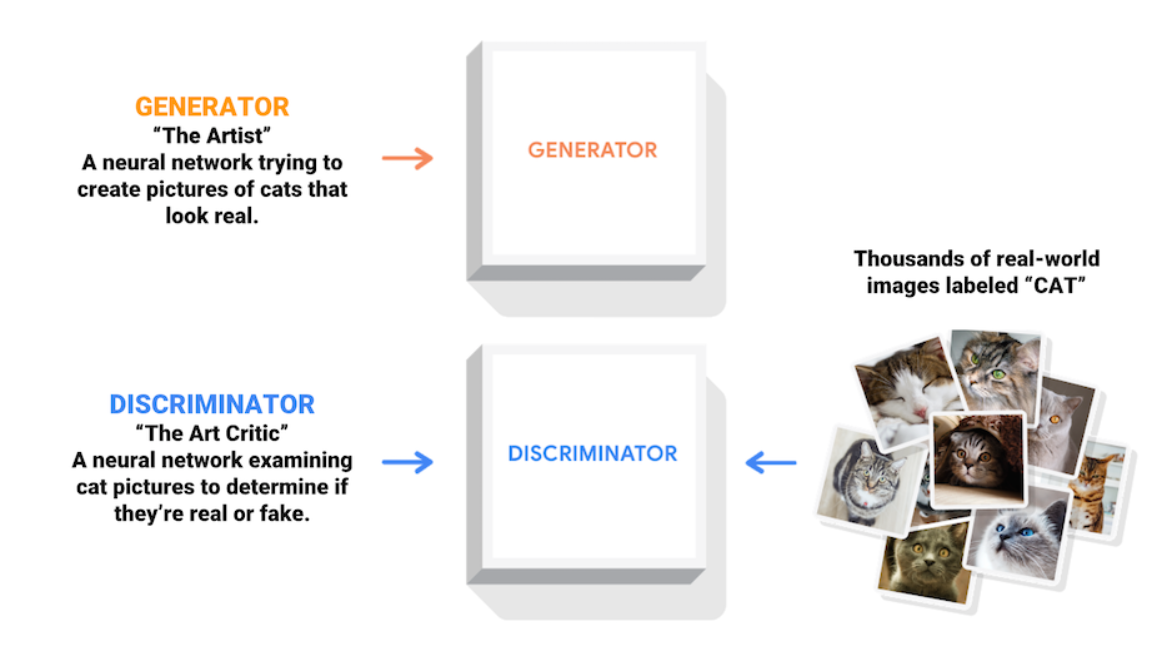
The objective of this project is to build a Generative Adversarial Network that can generate realistic 28x28 grayscale images resembling the MNIST handwritten digits. The generator creates images from random noise, while the discriminator tries to differentiate between real images and those generated by the generator. Both models are trained in opposition to improve the generator's ability to produce real like images over time.

**Descriptions:**

This **lab** focuses on creating a Generative Adversarial Networkto make realistic images of handwritten digits using the MNIST dataset. The GAN has two main parts: a generator, which makes new images from random noise, and a discriminator, which tries to decide if an image is real or fake. The generator learns by trying to trick the discriminator, while the discriminator gets better at spotting fake images. Both parts get better over time through repeated training.

As they train, the generator becomes better at making images that look like real handwritten digits, making it harder for the discriminator to tell them apart. The goal is to have the generator produce images that are almost indistinguishable from real ones. This lab shows how GANs can be used not only to create realistic images but also for other creative tasks like art and style transfer. The interaction between the generator and discriminator makes both models improve, making GANs to produce high quality real like images from random inputs.

**Model:**

****

**Building the parts of algorithm**

1) Data Preparation:

* Load and preprocess the MNIST dataset.
* Normalize the images to a range of [-1, 1] and reshape them for the GAN.

2) Model Architecture:

* Generator:

a.Use dense layers to start with random noise.

b.Apply upsampling with convolutional layers to generate images.

* Discriminator:

a.Use convolutional layers to analyze and classify images as real or fake.

b.Flatten the output and use a dense layer to produce a single score.

3) Training:

* Define loss functions for both the generator and the discriminator.
* Use the Adam optimizer to update the model weights.
* Alternate between training the discriminator (to correctly classify real and fake images) and the generator (to produce more convincing fake images).

4) Evaluation:

* Generate images from random noise using the trained generator.
* Assess the quality of the generated images and how well the discriminator distinguishes between real and fake images.
* Visualize and compare the generated images with real MNIST images to evaluate performance.

Code explanation:

import glob

import imageio

import matplotlib.pyplot as plt

import numpy as np

import os

import PIL

from tensorflow.keras import layers

import time

from IPython import display

1. TensorFlow and Keras: Used for building and training neural networks, providing tools like layers and optimizers.
2. NumPy: Handles numerical operations and array manipulations required for data processing.
3. Matplotlib: Creates visualizations for displaying generated images and monitoring training progress.
4. Imageio and PIL: Facilitate image reading, writing, and manipulation for saving and loading images.

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 size

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

    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', use\_bias=False))

    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', use\_bias=False, activation='tanh'))

    assert model.output\_shape == (None, 28, 28, 1)

    return model

1. Dense Layer: Starts with a dense layer that takes a 100-dimensional noise vector and transforms it into a large tensor (7x7x256).
2. BatchNormalization and LeakyReLU: These layers help stabilize and speed up training by normalizing activations and adding non-linearity.
3. Reshape: Converts the output into a 3D tensor suitable for upsampling.

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

1. Conv2D Layer: Extracts features from the images using convolutional layers.
2. LeakyReLU: Adds non-linearity.
3. Dropout: Helps prevent overfitting by randomly dropping some units during training.

train(train\_dataset, EPOCHS)

This line starts the training process for the GAN. It loops through the dataset for a specified number of epochs (EPOCHS), processing each batch of images to update the model weights. After each epoch, it prints the time taken to complete that epoch.

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

Discriminator Loss: Measures how well the discriminator can distinguish real images from fake ones. It calculates losses for real and fake images and sums them up.

def generator\_loss(fake\_output):

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

Generator Loss: Measures how well the generator can fool the discriminator. It calculates how close the fake images are to being classified as real by the discriminator.

**GitHub Link:**

**https://github.com/amruthaa-m/DL-Lab1/tree/main/Unit-2/lab7**