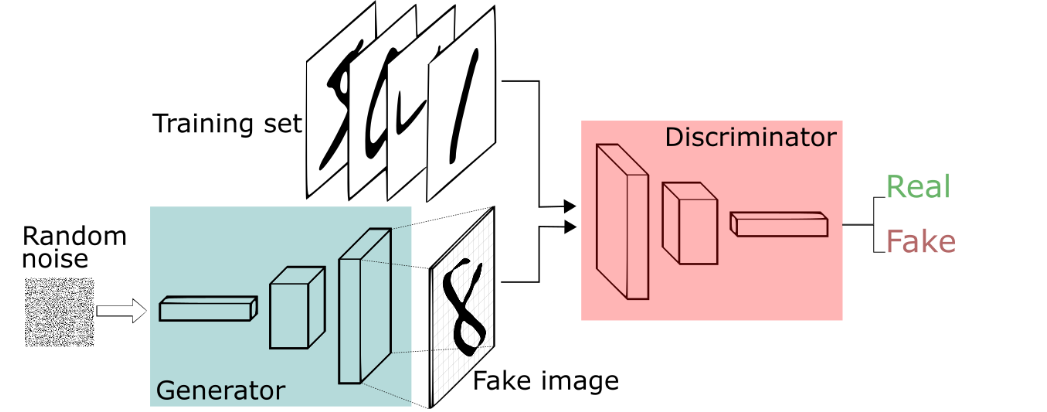
**Lab Exercise 5- Implementing GAN on MNIST in Google Colab**

**Objective:**

**Prerequisites:**

1. **Google Colab** (open Google Colab)
2. Basic knowledge of Python and TensorFlow or PyTorch.
3. Familiarity with GAN structure (Generator, Discriminator, Adversarial training process).



**Step-by-Step Exercise:**

**1. Setup the Environment**

First, import the necessary libraries: TensorFlow/Keras, NumPy, and Matplotlib.

# Import necessary libraries

import tensorflow as tf

from tensorflow.keras.layers import Dense, Reshape, Flatten, LeakyReLU, BatchNormalization, Conv2DTranspose, Conv2D

from tensorflow.keras.models import Sequential

import numpy as np

import matplotlib.pyplot as plt

# Check if GPU is available

print("Num GPUs Available: ", len(tf.config.experimental.list\_physical\_devices('GPU')))

**2. Load and Preprocess the MNIST Dataset**

We will use the MNIST dataset, which contains 28x28 grayscale images of digits from 0 to 9.

# Load the MNIST dataset

(train\_images, \_), (\_, \_) = tf.keras.datasets.mnist.load\_data()

# Normalize the images to [-1, 1] range for better GAN performance

train\_images = train\_images.reshape(train\_images.shape[0], 28, 28, 1).astype('float32')

train\_images = (train\_images - 127.5) / 127.5 # Rescale from [0, 255] to [-1, 1]

# Set buffer and batch size

BUFFER\_SIZE = 60000

BATCH\_SIZE = 256

# Create dataset

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

**3. Define the Generator**

The **generator** transforms random noise into a realistic image. It uses **Dense** and **Conv2DTranspose** layers for upsampling.

# Generator model

def build\_generator():

model = Sequential()

# First dense layer

model.add(Dense(7\*7\*256, use\_bias=False, input\_shape=(100,)))

model.add(BatchNormalization())

model.add(LeakyReLU())

# Reshape to 7x7x256

model.add(Reshape((7, 7, 256)))

# First transposed convolution (upsampling) layer

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

model.add(BatchNormalization())

model.add(LeakyReLU())

# Second transposed convolution (upsampling) layer

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

model.add(BatchNormalization())

model.add(LeakyReLU())

# Final transposed convolution to produce 28x28x1 image

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

return model

# Instantiate generator and display its architecture

generator = build\_generator()

generator.summary()

**4. Define the Discriminator**

The **discriminator** classifies images as "real" or "fake." It uses **Conv2D** layers to downsample images.

# Discriminator model

def build\_discriminator():

model = Sequential()

# First convolutional layer

model.add(Conv2D(64, (5, 5), strides=(2, 2), padding='same', input\_shape=[28, 28, 1]))

model.add(LeakyReLU())

# Second convolutional layer

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

model.add(LeakyReLU())

# Flatten and output layer

model.add(Flatten())

model.add(Dense(1, activation='sigmoid')) # Single output (real or fake)

return model

# Instantiate discriminator and display its architecture

discriminator = build\_discriminator()

discriminator.summary()

**5. Define the GAN (Combining Generator and Discriminator)**

In the **GAN**, the generator’s output is passed to the discriminator to determine if it's real or fake.

# Compile discriminator

discriminator.compile(optimizer='adam', loss='binary\_crossentropy')

# Build the GAN model by stacking generator and discriminator

def build\_gan(generator, discriminator):

model = Sequential()

model.add(generator)

model.add(discriminator)

return model

# Instantiate GAN

gan = build\_gan(generator, discriminator)

# Compile GAN (freeze the discriminator during generator training)

discriminator.trainable = False

gan.compile(optimizer='adam', loss='binary\_crossentropy')

**6. Training the GAN**

Define the training process for the GAN, including both generator and discriminator updates.

# Hyperparameters

EPOCHS = 50

NOISE\_DIM = 100

NUM\_EXAMPLES\_TO\_GENERATE = 16

# Random seed for consistent outputs

seed = tf.random.normal([NUM\_EXAMPLES\_TO\_GENERATE, NOISE\_DIM])

# Training step for GAN

def train\_step(images):

    # Get the actual batch size (might be smaller for the last batch)

    batch\_size = images.shape[0]

    # Generate fake images with the same batch size as the real images

    noise = tf.random.normal([batch\_size, NOISE\_DIM])

    generated\_images = generator(noise, training=True)

    # Labels for real (1) and fake (0) images

    real\_labels = tf.ones((batch\_size, 1))

    fake\_labels = tf.zeros((batch\_size, 1))

    # Train the discriminator on real and fake images

    discriminator.trainable = True

    d\_loss\_real = discriminator.train\_on\_batch(images, real\_labels)

    d\_loss\_fake = discriminator.train\_on\_batch(generated\_images, fake\_labels)

    d\_loss = 0.5 \* np.add(d\_loss\_real, d\_loss\_fake)

    # Train the generator

    noise = tf.random.normal([batch\_size, NOISE\_DIM])

    misleading\_labels = tf.ones((batch\_size, 1))  # Generator tries to fool the discriminator

    discriminator.trainable = False

    g\_loss = gan.train\_on\_batch(noise, misleading\_labels)

    return d\_loss, g\_loss

# Function to display generated 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(f'image\_at\_epoch\_{epoch:04d}.png')

    plt.show()

# Training loop

def train(dataset, epochs):

    for epoch in range(epochs):

        for image\_batch in dataset:

            d\_loss, g\_loss = train\_step(image\_batch)

        # Display progress

        generate\_and\_save\_images(generator, epoch + 1, seed)

        print(f'Epoch {epoch+1}, Generator Loss: {g\_loss}, Discriminator Loss: {d\_loss}')

# Start training

train(train\_dataset, EPOCHS)

**7. Visualize Results**

Once the training completes, you will see images generated by the GAN at different stages of training. This gives you a sense of how well the GAN is learning to generate realistic images.

**Key Takeaways:**

* **Generator** learns to create realistic images from random noise.
* **Discriminator** distinguishes real images from fake images generated by the generator.
* The training is adversarial, where both networks improve by challenging each other.

This basic GAN can be further extended to generate more complex images or even trained on different datasets.