# Generative Adversarial Networks (GANs) Using MNIST Dataset

## ****Introduction to GANs****

Generative Adversarial Networks (GANs) are a class of deep learning models introduced by Ian Goodfellow in 2014. GANs consist of two neural networks, the Generator and the Discriminator, that compete against each other in a zero-sum game framework. The Generator aims to create realistic data (such as images) from random noise, while the Discriminator's goal is to differentiate between real and fake data. Over successive training iterations, the Generator improves its ability to produce realistic data, while the Discriminator becomes better at distinguishing fake data from real.

Key advantages of GANs include their ability to generate high-quality synthetic data and their applications in fields such as image synthesis, data augmentation, and unsupervised learning.

## ****Dataset Description and Preprocessing Steps****

### ****Dataset Description****

The MNIST dataset consists of 70,000 grayscale images of handwritten digits (0-9), where each image has dimensions of 28x28 pixels. This dataset is widely used for benchmarking machine learning models and serves as an excellent choice for training a GAN to generate digit-like images.

### ****Preprocessing Steps****

1. **Normalization:** The pixel values of the MNIST images, originally in the range [0, 255], are normalized to the range [-1, 1]. This normalization ensures faster convergence and stable training of the GAN because the Generator's output uses the tanh activation function, which outputs values in this range.
2. **Adding Channel Dimension:** Each image is reshaped to include a channel dimension to match the expected input shape for convolutional layers in the models (28x28x1).
3. **Batching and Shuffling:** The dataset is divided into batches of size 128 and shuffled to ensure the model sees diverse examples during training.

# Load MNIST dataset

(x\_train, \_), (\_, \_) = mnist.load\_data()

# Normalize data to range [-1, 1]

x\_train = x\_train.astype("float32") / 127.5 - 1.0

# Add channel dimension

x\_train = np.expand\_dims(x\_train, axis=-1)

# Create batches

batch\_size = 128

dataset = tf.data.Dataset.from\_tensor\_slices(x\_train).shuffle(60000).batch(batch\_size)

## ****Detailed Explanation of Model Architectures****

### ****Generator Architecture****

The Generator takes a random noise vector as input and transforms it into a 28x28 grayscale image using a series of fully connected and convolutional layers.

#### ****Layers in the Generator:****

1. **Input Layer:** A dense layer maps the 100-dimensional noise vector to 256 units.
2. **Hidden Layers:**

* Three fully connected layers progressively increase the feature size (256 -> 512 -> 1024).
* Batch normalization layers stabilize training by normalizing activations.
* LeakyReLU activation introduces non-linearity and avoids dying neurons.

1. **Output Layer:** A dense layer reshapes the output to 28x28x1 using a tanh activation to match the pixel range of the MNIST images.

def build\_generator():

model = tf.keras.Sequential([

layers.Dense(256, input\_shape=(100,)),

layers.LeakyReLU(alpha=0.2),

layers.BatchNormalization(momentum=0.8),

layers.Dense(512),

layers.LeakyReLU(alpha=0.2),

layers.BatchNormalization(momentum=0.8),

layers.Dense(1024),

layers.LeakyReLU(alpha=0.2),

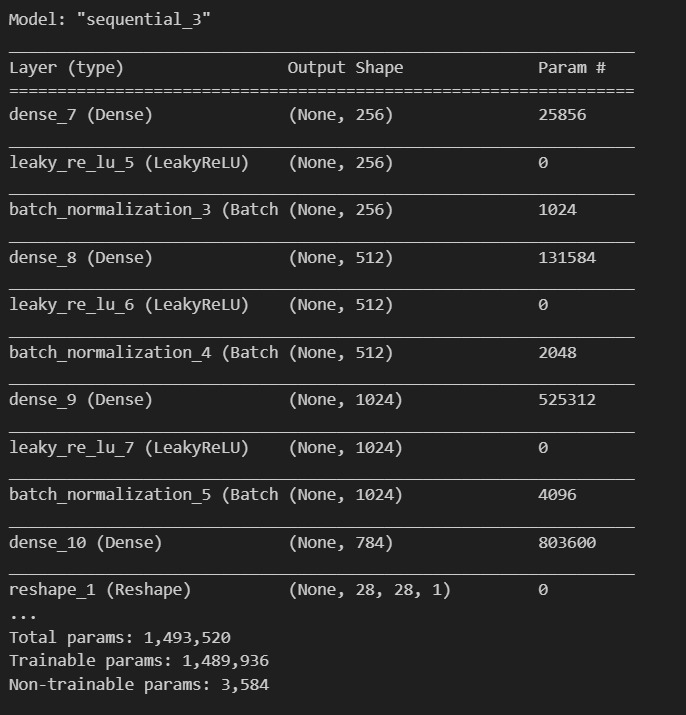
layers.BatchNormalization(momentum=0.8),

layers.Dense(28 \* 28 \* 1, activation='tanh'),

layers.Reshape((28, 28, 1))

])

return model



### ****Discriminator Architecture****

The Discriminator is a binary classifier that determines whether an input image is real or fake.

#### ****Layers in the Discriminator:****

1. **Input Layer:** Flattens the 28x28x1 image into a 784-dimensional vector.
2. **Hidden Layers:**

* Two dense layers progressively reduce the feature size (512 -> 256).
* LeakyReLU activation introduces non-linearity.
* Dropout layers (rate = 0.3) help prevent overfitting.

1. **Output Layer:** A dense layer with a sigmoid activation outputs a probability indicating whether the input is real or fake.

def build\_discriminator():

model = tf.keras.Sequential([

layers.Flatten(input\_shape=(28, 28, 1)),

layers.Dense(512),

layers.LeakyReLU(alpha=0.2),

layers.Dropout(0.3),

layers.Dense(256),

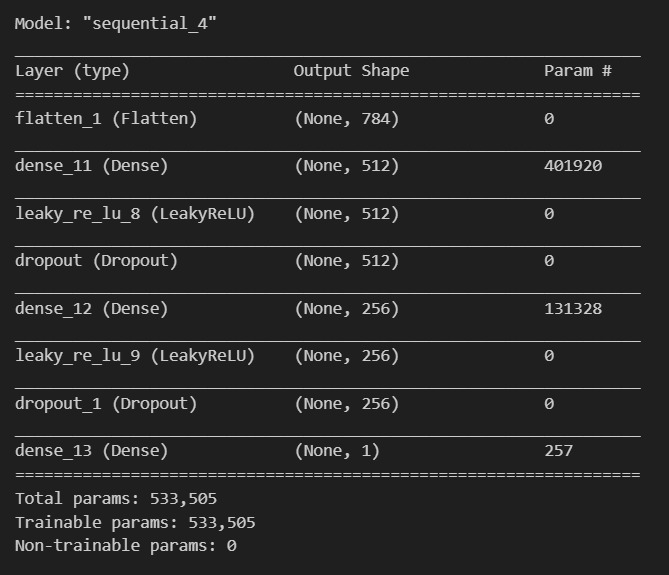
layers.LeakyReLU(alpha=0.2),

layers.Dropout(0.3),

layers.Dense(1, activation='sigmoid')

])

return model



### ****GAN Architecture****

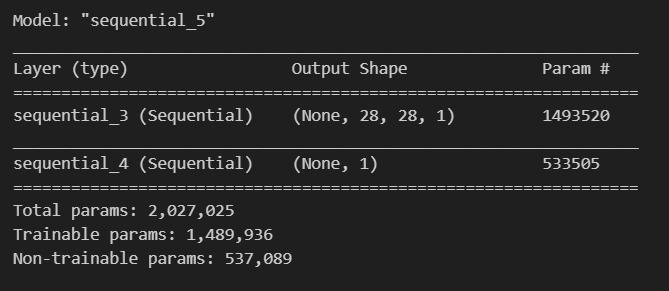
The GAN combines the Generator and Discriminator into a single model. During training, the Discriminator is frozen to ensure gradients flow only through the Generator.

def build\_gan(generator, discriminator):

discriminator.trainable = False

model = tf.keras.Sequential([generator, discriminator])

return model



## ****Explanation of Loss Functions and Training Strategy****

### ****Loss Functions****

1. **Discriminator Loss:** Measures how well the Discriminator distinguishes between real and fake images. It is the sum of:

* **Real Loss:** Binary crossentropy loss for real images (label = 1).
* **Fake Loss:** Binary crossentropy loss for fake images (label = 0).

real\_loss = loss\_fn(tf.ones\_like(real\_output), real\_output)

fake\_loss = loss\_fn(tf.zeros\_like(fake\_output), fake\_output)

disc\_loss = real\_loss + fake\_loss

1. **Generator Loss:** Measures how well the Generator fools the Discriminator. The goal is to maximize the Discriminator's error on fake images (label = 1).

gen\_loss = loss\_fn(tf.ones\_like(fake\_output), fake\_output)

### ****Training Strategy****

1. **Training the Discriminator:**

* Use real images from the dataset and fake images generated by the Generator.
* Compute real and fake losses and update the Discriminator's weights.

1. **Training the Generator:**

* Generate fake images from random noise.
* Pass these images through the Discriminator and compute the Generator loss.
* Update the Generator's weights to improve its ability to fool the Discriminator.

1. **Iterative Training:**

* Alternate between training the Discriminator and Generator.
* Repeat for multiple epochs until the Generator produces realistic images.

@tf.function

def train\_step(real\_images):

batch\_size = tf.shape(real\_images)[0]

random\_noise = tf.random.normal([batch\_size, 100])

# Train Discriminator

with tf.GradientTape() as disc\_tape:

fake\_images = generator(random\_noise, training=True)

real\_output = discriminator(real\_images, training=True)

fake\_output = discriminator(fake\_images, training=True)

real\_loss = loss\_fn(tf.ones\_like(real\_output), real\_output)

fake\_loss = loss\_fn(tf.zeros\_like(fake\_output), fake\_output)

disc\_loss = real\_loss + fake\_loss

disc\_grads = disc\_tape.gradient(disc\_loss, discriminator.trainable\_variables)

discriminator\_optimizer.apply\_gradients(zip(disc\_grads, discriminator.trainable\_variables))

# Train Generator

with tf.GradientTape() as gen\_tape:

generated\_images = generator(random\_noise, training=True)

fake\_output = discriminator(generated\_images, training=True)

gen\_loss = loss\_fn(tf.ones\_like(fake\_output), fake\_output)

gen\_grads = gen\_tape.gradient(gen\_loss, generator.trainable\_variables)

generator\_optimizer.apply\_gradients(zip(gen\_grads, generator.trainable\_variables))

return disc\_loss, gen\_loss

## ****Evaluation and Visualizations of Results****