**Generative Adversarial Network (GAN) with MNIST Dataset**

**Introduction to GANs**

Generative Adversarial Networks (GANs) are a class of deep learning models introduced by Ian Goodfellow et al. in 2014. GANs consist of two neural networks—a Generator and a Discriminator—that compete against each other in a zero-sum game. The Generator aims to produce realistic data samples, while the Discriminator attempts to distinguish between real and generated samples. This adversarial process drives both networks to improve, enabling the Generator to produce highly realistic samples.

Applications of GANs include image generation, style transfer, data augmentation, and super-resolution.

**Dataset Description and Preprocessing Steps**

**Dataset: MNIST**

The MNIST dataset consists of 70,000 grayscale images of handwritten digits (0-9), each of size 28x28 pixels. It is commonly used for benchmarking machine learning models.

**Preprocessing Steps:**

1. **Normalization:** Pixel values are normalized to the range [0, 1] to improve model convergence.
2. **Batching:** The dataset is divided into mini-batches for efficient training.
3. **Shuffling:** Data is shuffled to ensure diverse input samples during training.

import tensorflow as tf

from tensorflow.keras.datasets import mnist

import numpy as np

# Load MNIST dataset

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

# Normalize images to [0, 1] range

x\_train = x\_train / 255.0

# Reshape to include channel dimension

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

# Batch and shuffle dataset

BUFFER\_SIZE = 60000

BATCH\_SIZE = 128

dataset = tf.data.Dataset.from\_tensor\_slices(x\_train)

dataset = dataset.shuffle(BUFFER\_SIZE).batch(BATCH\_SIZE)

**Model Architectures**

**Generator**

The Generator creates synthetic images from random noise. Its architecture consists of:

* Dense layers to transform noise into a higher-dimensional representation.
* Transposed convolutional layers for upsampling.
* Activation functions like ReLU and Tanh.

from tensorflow.keras import layers

def build\_generator():

model = tf.keras.Sequential([

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

layers.BatchNormalization(),

layers.ReLU(),

layers.Reshape((7, 7, 256)),

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

layers.BatchNormalization(),

layers.ReLU(),

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

layers.BatchNormalization(),

layers.ReLU(),

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

])

return model

generator = build\_generator()

**Discriminator**

The Discriminator evaluates whether an input image is real or generated. Its architecture consists of:

* Convolutional layers for feature extraction.
* LeakyReLU activation to handle sparse gradients.
* A Dense layer for binary classification.

def build\_discriminator():

model = tf.keras.Sequential([

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

layers.LeakyReLU(alpha=0.2),

layers.Dropout(0.3),

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

layers.LeakyReLU(alpha=0.2),

layers.Dropout(0.3),

layers.Flatten(),

layers.Dense(1)

])

return model

discriminator = build\_discriminator()

**Loss Functions and Training Strategy**

**Loss Functions**

* **Generator Loss:** Measures how well the Generator fools the Discriminator. Uses Binary Cross-Entropy (BCE) loss.
* **Discriminator Loss:** Measures the Discriminator's ability to distinguish real from fake images. Also uses BCE loss.

cross\_entropy = tf.keras.losses.BinaryCrossentropy(from\_logits=True)

def generator\_loss(fake\_output):

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

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

**Optimizers**

Adam optimizers are used for both networks with appropriate learning rates and beta values.

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

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

**Training Loop**

1. Sample random noise as input for the Generator.
2. Generate fake images.
3. Compute Discriminator loss on real and fake images.
4. Compute Generator loss.
5. Update both networks using their respective optimizers.

EPOCHS = 50

noise\_dim = 100

num\_examples\_to\_generate = 16

seed = tf.random.normal([num\_examples\_to\_generate, noise\_dim])

@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\_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))

def train(dataset, epochs):

for epoch in range(epochs):

for image\_batch in dataset:

train\_step(image\_batch)

**Evaluation and Visualizations**

During training, generated images are saved and visualized to monitor the Generator's progress.

import matplotlib.pyplot as plt

def generate\_and\_save\_images(model, epoch, test\_input):

predictions = model(test\_input, training=False)

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()

**Results**

* The Generator successfully learned to produce realistic MNIST digits over 50 epochs.
* Example images generated during training are shown below:

**Conclusion**

This tutorial demonstrated how to build and train a GAN to generate handwritten digits using the MNIST dataset. The adversarial training process enabled the Generator to improve iteratively, producing realistic outputs. This framework can be extended to more complex datasets and applications.