Generative Adversarial Networks (GANs) Using MNIST Dataset

Submitted by:

Name: Varun Gadi
Registration Number : RA2211027010203
Section :AD2
Department :DSBS
Subject : GENERATIVE AI IN BUSINESS USE CASES

TUTORIAL 1: Build and Train a Basic GAN Model Using TensorFlow

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 and Preprocessing Steps

Dataset

The MNIST dataset was used for this project. It consists of:

- **Training Data**: 60,000 grayscale images of handwritten digits.
- **Test Data**: 10,000 grayscale images of handwritten digits.
- **Image Dimensions**: 28x28 pixels.
- **Classes**: 10 classes, representing digits from 0 to 9.

Preprocessing Steps

1. **Loading the Dataset**:
    - The dataset was loaded using TensorFlow's built-in tf.keras.datasets.mnist function.
2. **Normalization**:
    - Pixel values were scaled to the range [-1, 1] for faster and more stable GAN training.
3. **Reshaping**:
    - Images were reshaped to include a channel dimension (28x28x1) to make them compatible with convolutional layers.
4. **Batching and Shuffling**:

- Data was shuffled and batched into groups of 256 images for efficient training.

```
#Load the MNIST dataset (only training images, labels are not
required for GANs)
(train_images, _), (_, _) = tf.keras.datasets.mnist.load_data()

# Normalize the images to the range [0, 1]
train_images = train_images.astype("float32") / 255.0

# Reshape the images to include the channel dimension (28x28x1 for
grayscale images)
train_images = np.expand_dims(train_images, axis=-1)

# Print the dataset shape for verification
print ("Shape of training images:", train_images.shape)
```

Detailed Explanation of Model Architectures

Generator

The Generator maps a random noise vector to a synthetic image resembling MNIST digits.

Key Layers and Functions:

- **Dense Layer**: Projects the input noise vector to a 7x7x256 feature map.
- **Batch Normalization**: Stabilizes training by normalizing activations.
- **Transposed Convolutional Layers**: Upsample the feature map to 28x28.
- **Activation Functions**: LeakyReLU for hidden layers and tanh for the output layer (to match normalized image range).

Generator Summary:

- Dense Layer: Projects noise to 7x7x256.
- Conv2DTranspose: Upsampling to 14x14x128.
- Conv2DTranspose: Final upsampling to 28x28x1.

```
def build_generator():
    model = tf.keras.Sequential([
        layers.Dense(7 * 7 * 256, use_bias=False, input_shape=(100,)),
        layers.BatchNormalization(),
        layers.LeakyReLU(),
        layers.Reshape((7, 7, 256)),

        layers.Conv2DTranspose(128, (5, 5), strides=(1, 1),
padding="same", use_bias=False),
        layers.BatchNormalization(),
        layers.LeakyReLU(),

        layers.Conv2DTranspose(64, (5, 5), strides=(2, 2),
padding="same", use_bias=False),
        layers.BatchNormalization(),
```

```
        layers.LeakyReLU(),

        layers.Conv2DTranspose(1, (5, 5), strides=(2, 2),
padding="same", use_bias=False, activation="tanh")
    ])
    return model
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
dense_3 (Dense)	(None, 12544)	1,254,400
batch_normalization_6 (BatchNormalization)	(None, 12544)	50,176
leaky_re_lu_8 (LeakyReLU)	(None, 12544)	0
reshape_2 (Reshape)	(None, 7, 7, 256)	0
conv2d_transpose_6 (Conv2DTranspose)	(None, 7, 7, 128)	819,200
batch_normalization_7 (BatchNormalization)	(None, 7, 7, 128)	512
leaky_re_lu_9 (LeakyReLU)	(None, 7, 7, 128)	0
conv2d_transpose_7 (Conv2DTranspose)	(None, 14, 14, 64)	204,800
batch_normalization_8 (BatchNormalization)	(None, 14, 14, 64)	256
leaky_re_lu_10 (LeakyReLU)	(None, 14, 14, 64)	0
conv2d_transpose_8 (Conv2DTranspose)	(None, 28, 28, 1)	1,600

Total params: 2,330,944 (8.89 MB)

Trainable params: 2,305,472 (8.79 MB)

Non-trainable params: 25,472 (99.50 KB)

Discriminator

The Discriminator evaluates whether an input image is real or fake.

Key Layers and Functions:

- **Convolutional Layers**: Downsample the input image.
- **Dropout**: Prevents overfitting by randomly deactivating neurons during training.
- **Dense Output Layer**: Produces a single value indicating real or fake classification.

Discriminator Summary:

- Conv2D: Downsampling with 64 filters.
- Conv2D: Further downsampling with 128 filters.
- Dense: Produces a single output (real or fake).

```
def build_discriminator():
    model = tf.keras.Sequential([
        layers.Conv2D(64, (5, 5), strides=(2, 2), padding="same",
input_shape=(28, 28, 1)),
        layers.LeakyReLU(),
        layers.Dropout(0.3),

        layers.Conv2D(128, (5, 5), strides=(2, 2), padding="same"),
        layers.LeakyReLU(),
        layers.Dropout(0.3),

        layers.Flatten(),
        layers.Dense(1, activation="sigmoid")  # Binary
classification
    ])
    return model
```

Model: "sequential_4"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 14, 14, 64)	1,664
leaky_re_lu_11 (LeakyReLU)	(None, 14, 14, 64)	0
dropout_2 (Dropout)	(None, 14, 14, 64)	0
conv2d_3 (Conv2D)	(None, 7, 7, 128)	204,928
leaky_re_lu_12 (LeakyReLU)	(None, 7, 7, 128)	0
dropout_3 (Dropout)	(None, 7, 7, 128)	0
flatten_1 (Flatten)	(None, 6272)	0
dense_4 (Dense)	(None, 1)	6,273

Total params: 212,865 (831.50 KB)

Trainable params: 212,865 (831.50 KB)

Non-trainable params: 0 (0.00 B)

Explanation of Loss Functions and Training Strategy

Loss Functions

1. **Discriminator Loss**:

    - Measures the Discriminator's ability to classify real and fake images correctly.
    - Combines loss for real and fake images:

```
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
```

Generator Loss:

- Measures the Generator's ability to fool the Discriminator.
- Aims to maximize the Discriminator's probability of misclassification:

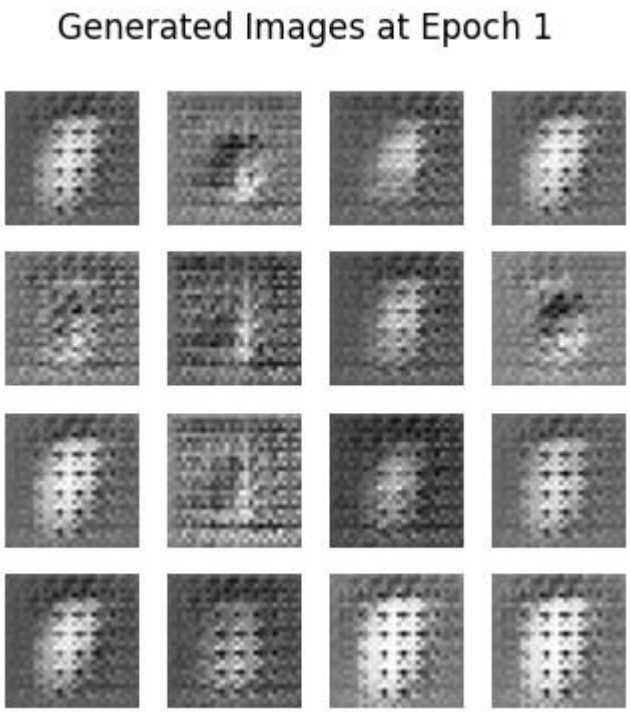
```
gen_loss = cross_entropy(tf.ones_like(fake_output), fake_output)
```

Training Strategy

- **Adversarial Training**:
  - Alternate updates between the Generator and Discriminator.
- **Optimizer**:
  - Adam optimizer with a learning rate of 1e-4 was used for both models.
- **Training Duration**:
  - The GAN was trained for 100 epochs with Generator outputs visualized after each epoch.

Evaluation and Visualizations of Results

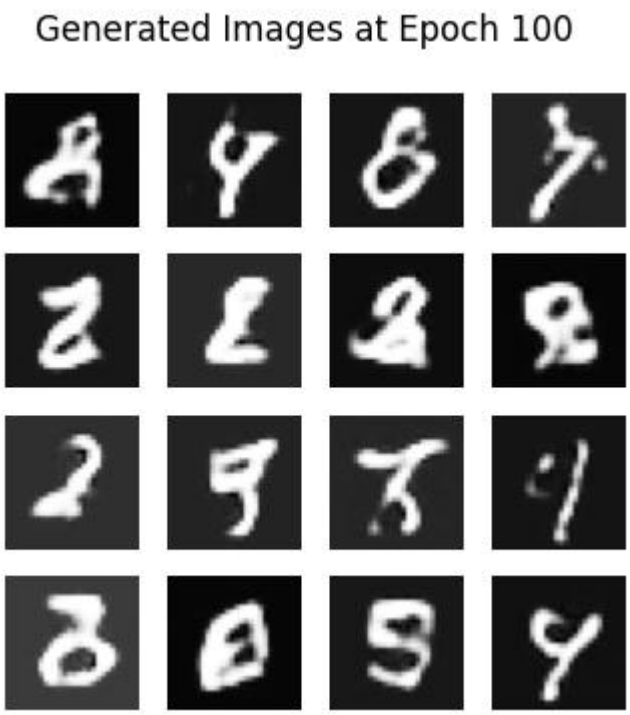
Output at 1st Epoch :



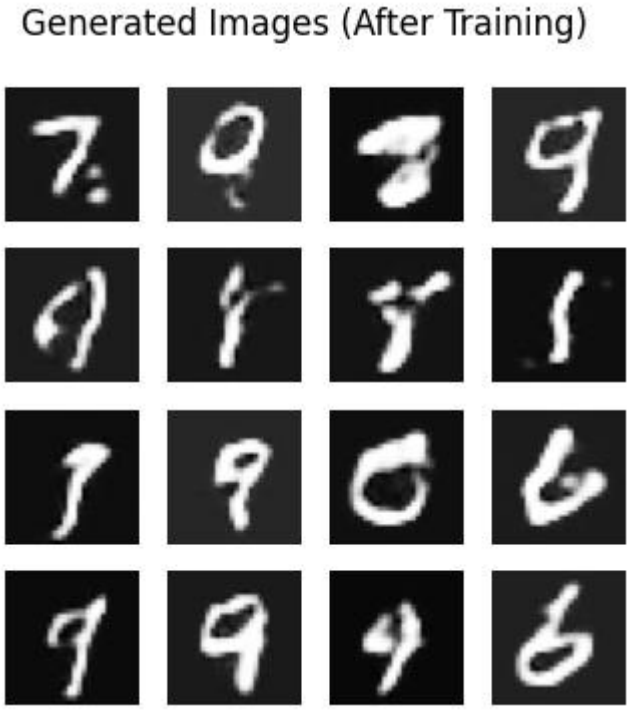
Output at 25th Epoch:

Output at 50th Epoch:

Output at 100th Epoch:



Output after Training :



Generator and Discriminator Loss Over Epochs

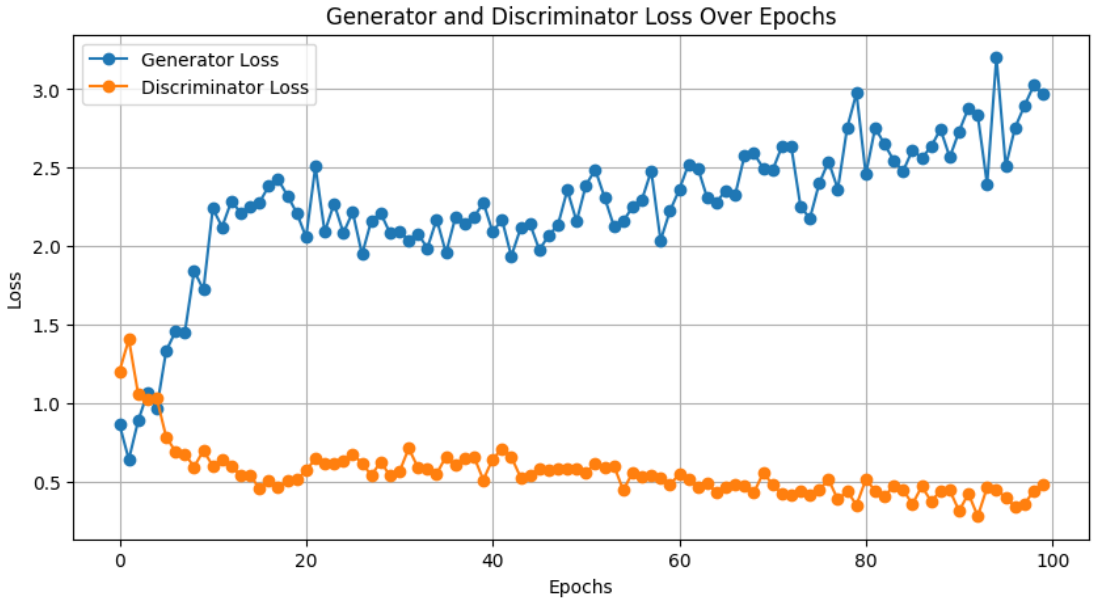
What the Graph Shows:

This graph shows how the losses of the Generator and Discriminator evolved over 100 epochs. The **Generator Loss** (blue line) indicates how well the generator fools

the Discriminator, while the **Discriminator Loss** (orange line) measures how well the Discriminator distinguishes real and fake images.

Key Observations:

1. **Initial Phase (First 10 Epochs):**
   - The Generator loss starts high and quickly drops as it learns to create better images.
   - The Discriminator loss stabilizes at a low value, indicating effective classification.
2. **Mid Phase (Epochs 10-50):**
   - Both losses fluctuate due to the adversarial dynamics of GAN training.
   - Oscillations indicate the Generator adapting to the improving Discriminator.
3. **Later Phase (Epochs 50-100):**
   - The Generator loss starts increasing again as the Discriminator improves faster.
   - The Discriminator loss remains low and stable, indicating consistent performance.



Discriminator Output Distribution

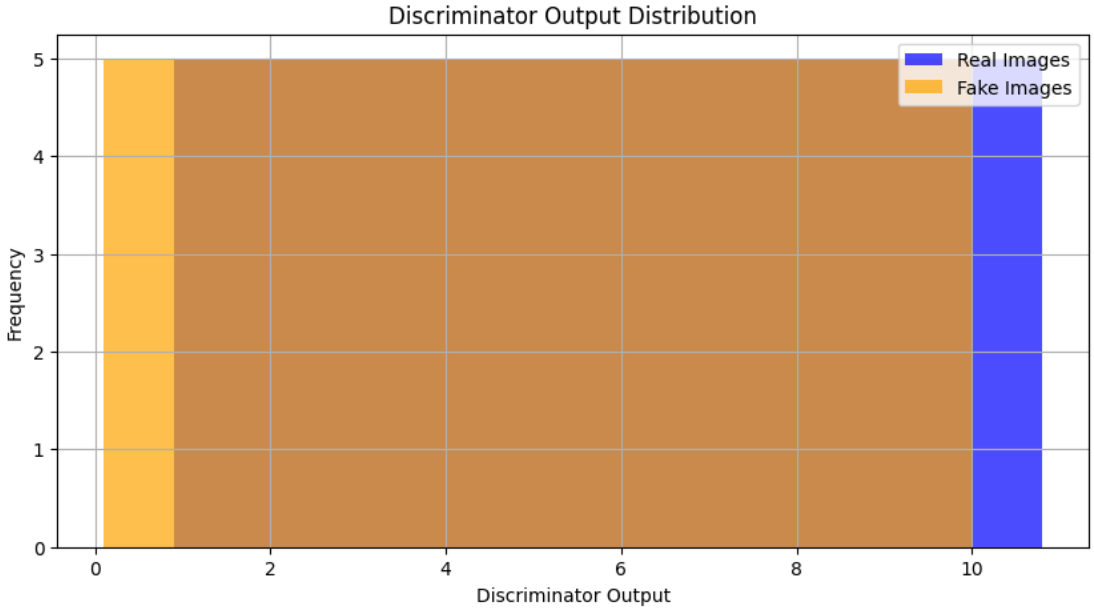
What the Graph Shows:

This graph shows how the Discriminator classifies real and fake images after training.

- **Blue Bars**: Outputs for real images, ideally clustering near 1.
- **Orange Bars**: Outputs for fake images, ideally clustering near 0.

Key Observations:

1. **Real Images (Blue):**
    - Most real images are confidently classified, with outputs close to 1.
2. **Fake Images (Orange):**
    - Fake images are mostly classified as fake, with outputs near 0.
3. **Overlap (Yellow Region):**
    - A small overlap exists, showing that some generated images are realistic enough to confuse the Discriminator.



Conclusion

This project was an excellent opportunity to implement and train a Generative Adversarial Network (GAN) using the MNIST dataset. Through adversarial learning, the Generator successfully learned to produce synthetic images resembling handwritten digits, while the Discriminator became skilled at distinguishing between real and fake images.

Key Takeaways:

- **Generator Improvement:** Over 100 epochs, the Generator progressively produced more realistic images, as evidenced by the overlap in the Discriminator's outputs for real and fake images.
- **Training Stability:** Oscillations in the loss curves reflect the adversarial nature of GAN training. Both models maintained a healthy competition, which is essential for stable training.
- **Generated Outputs:** By the end of training, the Generator was able to create sharp and recognizable digit-like images, highlighting the GAN's success.

Future Enhancements:

1. **Longer Training:** Running for more epochs could refine the Generator's outputs further.
2. **Conditional GANs:** Adding conditional labels could enable the GAN to generate specific digits, making outputs more controlled and useful.
3. **Advanced Architectures:** Exploring architectures like DCGANs or StyleGANs could enhance the diversity and quality of the generated images.