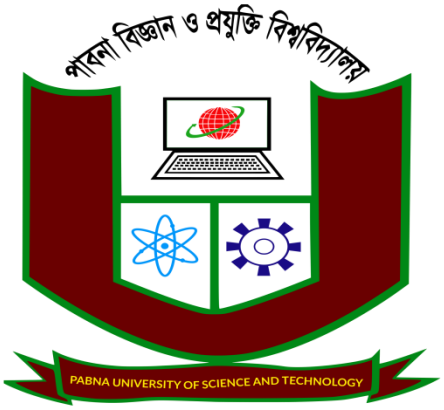

Department of Information and Communication Engineering

Pabna University of Science and Technology

Faculty of Engineering and Technology

B.Sc (Engineering) 4th Year 2nd Semester Exam-2023

Session: 2019-2020

Course Title : Neural Networks Sessional.

Course Code: ICE-4206

Lab Report

Submitted By: MD RASHED
Roll No**:** 200613
Dept. of Information and Communication Engineering
Pabna University of Science and Technology
Pabna-6600,Bangladesh

Submitted To: Dr. Md. Imran Hossain
Associate Professor
Dept. of Information and Communication Engineering.
Pabna University of Science and Technology
Pabna-6600,Bangladesh

Date of Submission: 10/04/2025 Signature

Experiment No: 02
Experiment Name: Generative Adversarial Network (GAN) for Image Generation on MNIST dataset.

Objectives:

- To understand the architecture and principles of Generative Adversarial Networks (GANs).
- To implement a GAN model for image generation.
- To train the GAN effectively.
- To evaluate the quality of generated images.
- To investigate the impact of hyperparameters.
- To demonstrate the capability of GANs in unsupervised learning.

Theory:
A **generative adversarial network (GAN)** is a machine learning (ML) model in which two neural networks compete by using deep learning methods to become more accurate in their predictions. GANs typically run unsupervised and use a cooperative zero-sum game framework to learn.

A generative adversarial network (GAN) has two parts:
1. Generator
2. Discriminator

The **generator** learns to generate plausible data. The generated instances become negative training examples for the discriminator.
The **discriminator** learns to distinguish the generator's fake data from real data. The discriminator penalizes the generator for producing implausible results.
When training begins, the generator produces obviously fake data, and the discriminator quickly learns to tell that it's fake

As training progresses, the generator gets closer to producing output that can fool the discriminator:

Finally, if generator training goes well, the discriminator gets worse at telling the difference between real and fake. It starts to classify fake data as real, and its accuracy decreases.

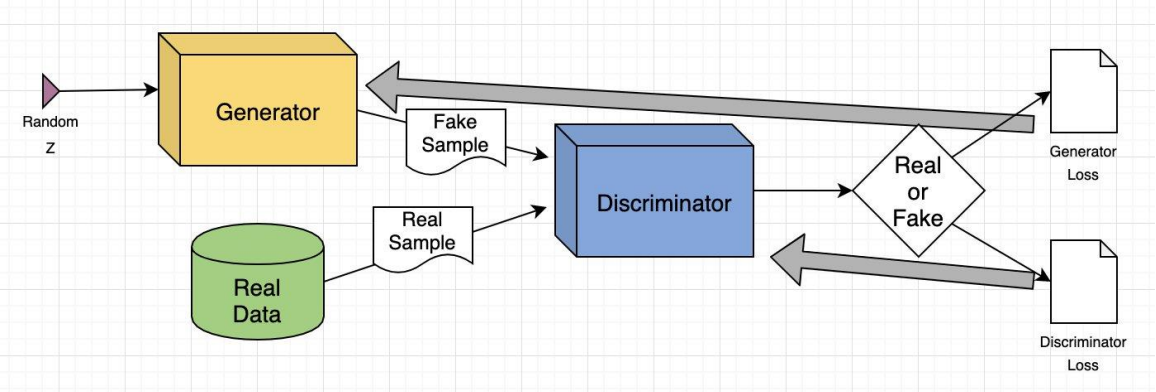

Fig. 1: General Block Diagram of Generative Adversarial Network (GAN)

Both the generator and the discriminator are neural networks. The generator output is connected directly to the discriminator input. Through backpropagation, the discriminator's classification provides a signal that the generator uses to update its weights.

Let's explain the pieces of this system in greater detail:

The Discriminator:

The discriminator in a GAN is simply a classifier. It tries to distinguish real data from the data created by the generator. It could use any network architecture appropriate to the type of data it's classifying.

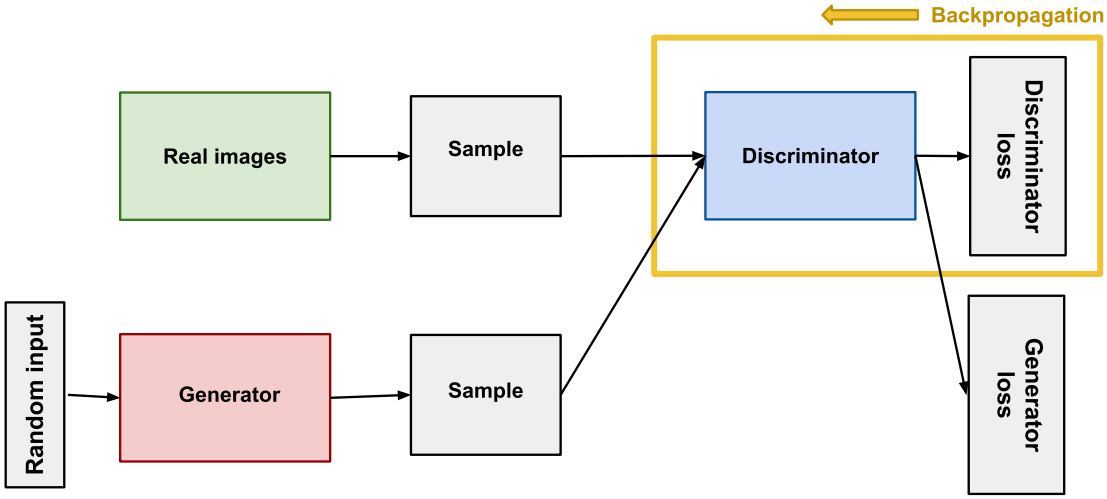

Fig.2: Backpropagation in Discriminator Training

The discriminator's training data comes from two sources:

Real data instances, such as real pictures of people. The discriminator uses these instances as positive examples during training.

Fake data instances created by the generator. The discriminator uses these instances as negative examples during training.

In Figure 2, the two "Sample" boxes represent these two data sources feeding into the discriminator. During discriminator training, the generator does not train. Its weights remain constant while it produces examples for the discriminator to train on.

Training the Discriminator

The discriminator connects to two loss functions. During discriminator training, the discriminator ignores the generator loss and just uses the discriminator loss. We use the generator loss during generator training, as described in the next section.

During discriminator training, the discriminator classifies both real data and fake data from the generator. The discriminator loss penalizes the discriminator for misclassifying a real instance as fake or a fake instance as real. The discriminator updates its weights through backpropagation from the discriminator loss through the discriminator network.

The Generator:

The generator part of a GAN learns to create fake data by incorporating feedback from the discriminator. It learns to make the discriminator classify its output as real.

Generator training requires tighter integration between the generator and the discriminator than discriminator training requires. The portion of the GAN that trains the generator includes:

- random input
- generator network, which transforms the random input into a data instance
- discriminator network, which classifies the generated data
- discriminator output
- generator loss, which penalizes the generator for failing to fool the discriminator

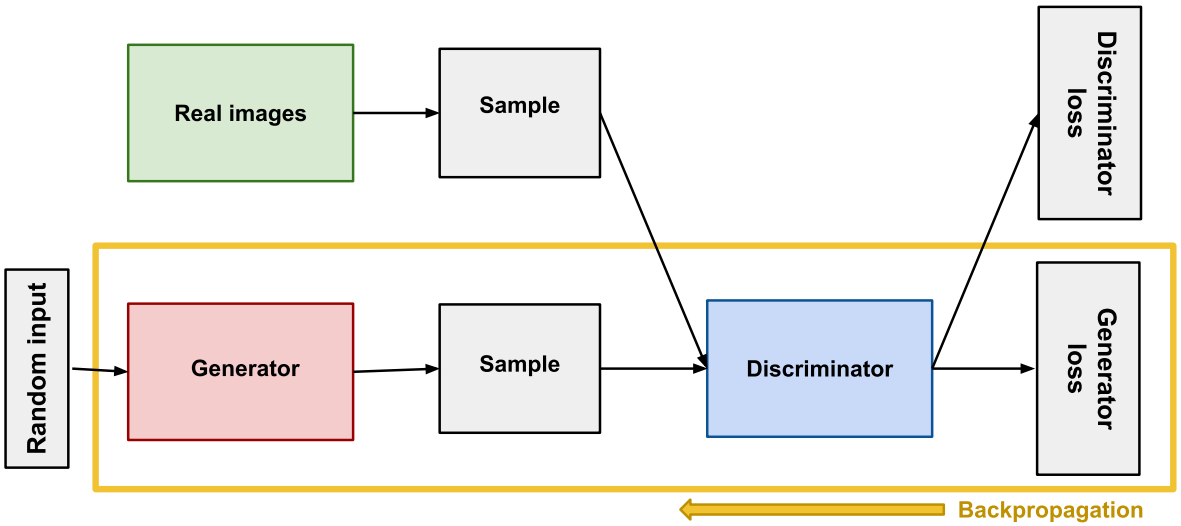

Fig.3: Backpropagation in Generator Training

Random Input

Neural networks need some form of input. Normally we input data that we want to do something with, like an instance that we want to classify or make a prediction about. But what do we use as input for a network that outputs entirely new data instances?

In its most basic form, a GAN takes random noise as its input. The generator then transforms this noise into a meaningful output. By introducing noise, we can get the GAN to produce a wide variety of data, sampling from different places in the target distribution.

Experiments suggest that the distribution of the noise doesn't matter much, so we can choose something that's easy to sample from, like a uniform distribution. For convenience the space from which the noise is sampled is usually of smaller dimension than the dimensionality of the output space.

Using the Discriminator to Train the Generator

To train a neural network, we alter the network's weights to reduce the error or loss of its output. In our GAN, however, the generator is not directly connected to the loss that we're trying to affect. The generator feeds into the discriminator net, and the discriminator produces the output we're trying to affect. The generator loss penalizes the generator for producing a sample that the discriminator network classifies as fake.

This extra chunk of the network must be included in backpropagation. Backpropagation adjusts each weight in the right direction by calculating the weight's impact on the output — how the output would change if you changed the weight. However the impact of a generator weight depends on the impact of the discriminator weights it feeds into. So backpropagation starts at the output and flows back through the discriminator into the generator. At the same time, we don't want the discriminator to change during generator training. Trying to hit a moving target would make a hard problem even harder for the generator.

So we train the generator with the following procedure:

- Sample random noise.
- Produce generator output from sampled random noise.
- Get discriminator "Real" or "Fake" classification for generator output.
- Calculate loss from discriminator classification.
- Backpropagate through both the discriminator and generator to obtain gradients.
- Use gradients to change only the generator weights.

GAN Training

GAN contains two separately trained networks, its training algorithm must address two complications:

- GANs must juggle two different kinds of training (generator and discriminator).
- GAN convergence is hard to identify.

Convergence

As the generator improves with training, the discriminator performance gets worse because the discriminator can't easily tell the difference between real and fake. If the generator succeeds perfectly, then the discriminator has a 50% accuracy. In effect, the discriminator flips a coin to make its prediction.

This progression poses a problem for convergence of the GAN as a whole: the discriminator feedback gets less meaningful over time. If the GAN continues training past the point when the discriminator is giving completely random feedback, then the generator starts to train on junk feedback, and its own quality may collapse.

For a GAN, convergence is often a fleeting, rather than stable, state.

Loss Functions

GANs try to replicate a probability distribution. They should therefore use loss functions that reflect the distance between the distribution of the data generated by the GAN and the distribution of the real data.

One Loss Function or Two?

A GAN can have two loss functions: one for generator training and one for discriminator training. How can two loss functions work together to reflect a distance measure between probability distributions?

In the loss schemes we'll look at here, the generator and discriminator losses derive from a single measure of distance between probability distributions. In both of these schemes, however, the generator can only affect

one term in the distance measure: the term that reflects the distribution of the fake data. So during generator training we drop the other term, which reflects the distribution of the real data.
The generator and discriminator losses look different in the end, even though they derive from a single formula.

Minimax Loss
In the paper that introduced GANs, the generator tries to minimize the following function while the discriminator tries to maximize it:

$$E_x[log(D(x))] + E_z[log(1 - D(G(z)))]$$

In this function:

- D(x) is the discriminator's estimate of the probability that real data instance x is real.
- Ex is the expected value over all real data instances.
- G(z) is the generator's output when given noise z.
- D(G(z)) is the discriminator's estimate of the probability that a fake instance is real.
- Ez is the expected value over all random inputs to the generator (in effect, the expected value over all generated fake instances G(z)).
- The formula derives from the cross-entropy between the real and generated distributions.
- The generator can't directly affect the log(D(x)) term in the function, so, for the generator, minimizing the loss is equivalent to minimizing log(1 - D(G(z))).

Wasserstein Loss
This loss function depends on a modification of the GAN scheme (called "Wasserstein GAN" or "WGAN") in which the discriminator does not actually classify instances. For each instance it outputs a number. This number does not have to be less than one or greater than 0, so we can't use 0.5 as a threshold to decide whether an instance is real or fake. Discriminator training just tries to make the output bigger for real instances than for fake instances.
Because it can't really discriminate between real and fake, the WGAN discriminator is actually called a "critic" instead of a "discriminator". This distinction has theoretical importance, but for practical purposes we can treat it as an acknowledgement that the inputs to the loss functions don't have to be probabilities.

MNIST Dataset Description
The MNIST (Modified National Institute of Standards and Technology) handwritten digit dataset is one of the most widely used benchmark datasets in the field of machine learning and computer vision. It contains a total of 70,000 grayscale images of handwritten digits from 0 to 9, with 60,000 images designated for training and 10,000 for testing. Each image is a 28x28 pixel square, resulting in a total of 784 pixels per image. The pixel values range from 0 (white background) to 255 (black ink), and each image is labeled with a corresponding digit class (0 through 9).
The dataset was created by combining samples from American Census Bureau employees and high school students, offering a diverse representation of handwriting styles. It is particularly valuable for beginners in machine learning, as it provides a simple yet effective platform to practice classification techniques, neural network implementation, and image processing. Due to its clean and well-structured format, MNIST has

become a standard benchmark for evaluating and comparing the performance of different algorithms, including logistic regression, support vector machines (SVM), decision trees, and deep learning models such as convolutional neural networks (CNNs).

One of the key advantages of the MNIST dataset is its accessibility and ease of use. It is available in many popular machine learning libraries, such as TensorFlow, Keras, PyTorch, and Scikit-learn, often with just a single line of code. Despite its simplicity, achieving near-perfect accuracy on MNIST requires thoughtful model design and training strategies, making it an excellent platform for experimenting with optimization techniques, regularization methods, and hyperparameter tuning.

Python Source Code:

```
import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np
import matplotlib.pyplot as plt

# Load and preprocess MNIST dataset
(train_images, _), (_, _) = tf.keras.datasets.mnist.load_data()
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype('float32')
train_images = (train_images - 127.5) / 127.5  # Normalize to [-1, 1]
BUFFER_SIZE = 60000
BATCH_SIZE = 256

# Create a tf.data.Dataset for training
train_dataset =
tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)

# Define the Generator model
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)))
    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    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'))
    return model

# Define the Discriminator model
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

# Define loss functions
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

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

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

# Optimizers
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

# Initialize models
generator = make_generator_model()
discriminator = make_discriminator_model()

# Training step
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, 100])

    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))
    return gen_loss, disc_loss

# Training loop
def train(dataset, epochs):
    for epoch in range(epochs):
        for image_batch in dataset:
            gen_loss, disc_loss = train_step(image_batch)
```

```
        # Print loss every epoch
        print(f'Epoch {epoch + 1}, Generator Loss: {gen_loss:.4f}, Discriminator Loss: {disc_loss:.4f}')

        # Generate and save images every 5 epochs
        if (epoch + 1) % 5 == 0:
            generate_and_save_images(generator, epoch + 1, seed)

# Generate and visualize test 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()

# Set random seed for reproducibility
seed = tf.random.normal([16, 100])  # 16 images for visualization

# Train the GAN
EPOCHS = 50
train(train_dataset, EPOCHS)

# Generate final test images after training
generate_and_save_images(generator, EPOCHS, seed)
```

Output:

Epoch 50, Generator Loss: 0.8876, Discriminator Loss: 1.2997