## HCMUS - VNUHCM / FIT /

## Computer Vision & Cognitive Cybernetics Department

## Digital Image and Video Processing Application

**Student ID:** 21127690

**Student name:** Ngo Nguyen Thanh Thanh

# Report: Generative Adversarial Networks

## I. Evaluation summary:

| No | Task | Implementation | Completion (%) |
|---|---|---|---|
| 1 | Setup Google Colab for training | Configure Colab environment, install necessary libraries | 100% |
| 2 | Load and preprocess dataset | Use MNIST dataset, normalize images | 100% |
| 3 | Define Generator model | Implement a neural network to generate images | 100% |
| 4 | Define Discriminator model | Implement a classifier to distinguish real vs fake images | 100% |
| 5 | Setup optimizers for GAN training | Use Adam optimizer for both Generator and Discriminator | 100% |
| 6 | Train GAN model | Train Generator and Discriminator iteratively | 100% |
| 7 | Evaluate model performance | Create the loss curve analysis helps monitor GAN training stability | 100% |
| 8 | Generate and visualize images | Implement function to generate and display images using trained Generator | 100% |

## II. List of features and file structure:

### Functions and Methods Used

The notebook primarily utilizes the following libraries:

- torch, torch.nn, torch.optim: For defining and training neural networks.
- torchvision.transforms: For preprocessing image data.
- matplotlib.pyplot, numpy: For visualization and numerical operations.

### Main Functions

The main functions in the notebook can be categorized as follows:

**1. Model Implementation**

This section includes the definitions of the **Discriminator (D) and Generator (G)** models.

- **Discriminator (D)**
  - class Discriminator(nn.Module): Defines a neural network for distinguishing real and generated images.
  - __init__(self, inp_dim=784): Initializes the discriminator with fully connected layers.
  - forward(self, x): Passes input through the network and outputs a probability score.
- **Generator (G)**
  - class Generator(nn.Module): Defines a neural network for generating synthetic images.
  - __init__(self, z_dim=100): Initializes the generator with a latent space input.
  - forward(self, x): Transforms random noise into an image representation.

**2. Data Processing**

Handles image dataset loading and preprocessing.

- transforms.ToTensor(): Converts image data into tensors.
- transforms.Normalize((0.5,), (0.5,)): Normalizes the dataset for stable training.
- x = x.view(x.size(0), 784): Flattens the images for the neural network input.

**3. Training Functions**

Manages the training process of the GAN model.

- optimizerD = torch.optim.Adam(D.parameters(), lr=0.0002): Optimizer for the discriminator.
- optimizerG = torch.optim.Adam(G.parameters(), lr=0.0002): Optimizer for the generator.
- lossD = criterion(output, label): Computes the loss for the discriminator.
- lossG = criterion(output, label): Computes the loss for the generator.
- lossD.backward(), lossG.backward(): Performs backpropagation.
- optimizerD.step(), optimizerG.step(): Updates model weights.

**4. Result Visualization**

After training, the generated images are displayed.

- make_grid(images, nrow=8, normalize=True): Creates a grid of generated images.
- plt.imshow(...): Displays the generated images.

# How to Run the Code

1. **Setup Environment**
   - The code is designed to run in **Google Colab**.
   - Mount Google Drive using:

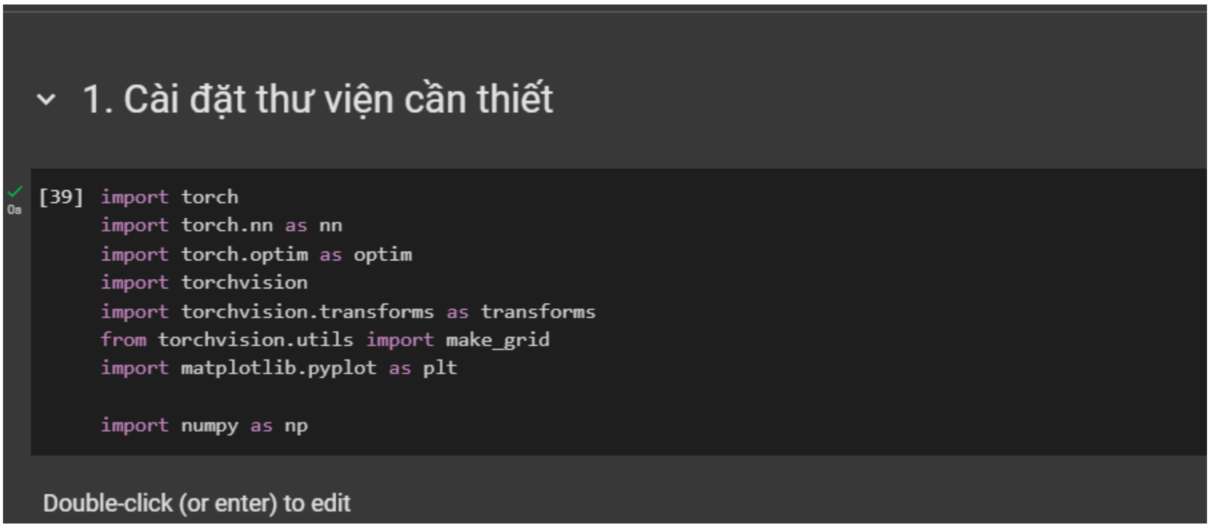
```
from google.colab import drive
drive.mount('/content/drive')
```

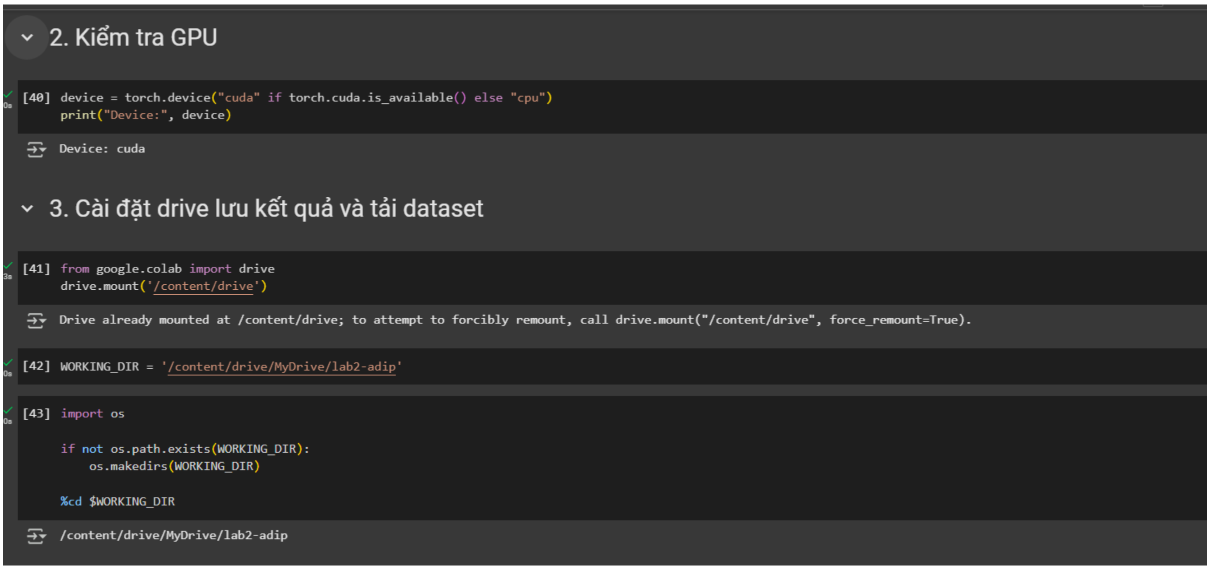
- Ensure CUDA is available for GPU acceleration:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", device)
```

2. **Execute the Notebook Cells**
   - Run all cells sequentially to:
     - Load dependencies.
     - Define and initialize the **Discriminator** and **Generator**.
     - Preprocess the dataset.
     - Train the model with **backpropagation and optimization**.
     - Visualize generated images.

# Image proof:



## 4. Tải và lưu dataset MNIST

```
# Đường dẫn lưu dataset trên Google Drive
location_path = '/content/drive/MyDrive/' + 'lab2-adip/dataset'

# Định nghĩa transform trước khi truyền vào dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Batch size mới
batch_size = 128

# Tải dataset MNIST
dataset = torchvision.datasets.MNIST(root=location_path, train=True, download=True, transform=transform)

# Tạo DataLoader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

# 5. Định nghĩa mô hình Discriminator

## Tự code

```
[45] class Discriminator(nn.Module):
        def __init__(self, inp_dim=784):
            super(Discriminator, self).__init__()
            self.w1 = nn.Parameter(torch.randn(inp_dim, 128) * 0.02)  # Dùng nn.Parameter
            self.b1 = nn.Parameter(torch.zeros(128))
            self.w2 = nn.Parameter(torch.randn(128, 1) * 0.02)
            self.b2 = nn.Parameter(torch.zeros(1))

        def forward(self, x):
            x = x.view(x.size(0), 784)
            h = torch.matmul(x, self.w1) + self.b1
            h = torch.maximum(0.2 * h, h)  # LeakyReLU
            out = torch.matmul(h, self.w2) + self.b2
            out = torch.sigmoid(out)  # Sigmoid
            return out
```

## Tham khảo lab

```
class Discriminator(nn.Module):
    def __init__(self, inp_dim=784):
        super(Discriminator, self).__init__()
        self.fc1 = nn.Linear(inp_dim, 128)
        self.nonlin1 = nn.LeakyReLU(0.2)
        self.fc2 = nn.Linear(128, 1)

    def forward(self, x):
        x = x.view(x.size(0), 784)  # Flatten (batch_size x 1 x 28 x 28) -> (batch_size x 784)
        h = self.nonlin1(self.fc1(x))
        out = self.fc2(h)
        out = torch.sigmoid(out)
        return out
```

# 6. Định nghĩa mô hình Generator

## Tự code

[46]
```
class Generator(nn.Module):
    def __init__(self, z_dim=100):
        super(Generator, self).__init__()
        self.w1 = nn.Parameter(torch.randn(z_dim, 128) * 0.02)
        self.b1 = nn.Parameter(torch.zeros(128))
        self.w2 = nn.Parameter(torch.randn(128, 784) * 0.02)
        self.b2 = nn.Parameter(torch.zeros(784))

    def forward(self, x):
        h = torch.matmul(x, self.w1) + self.b1
        h = torch.maximum(0.2 * h, h)  # LeakyReLU
        out = torch.matmul(h, self.w2) + self.b2
        out = torch.tanh(out)  # [-1, 1]
        out = out.view(out.size(0), 1, 28, 28)
        return out
```

## Tham khảo lab

```
class Generator(nn.Module):
    def __init__(self, z_dim=100):
        super(Generator, self).__init__()
        self.fc1 = nn.Linear(z_dim, 128)
        self.nonlin1 = nn.LeakyReLU(0.2)
        self.fc2 = nn.Linear(128, 784)

    def forward(self, x):
        h = self.nonlin1(self.fc1(x))
        out = self.fc2(h)
        out = torch.tanh(out)  # Đưa giá trị về khoảng [-1, 1]
        out = out.view(out.size(0), 1, 28, 28)  # Chuyển về kích thước ảnh
        return out
```

## 7. Khởi tạo mô hình

[47]
```
# Re-initialize D, G (Assuming Discriminator and Generator classes are defined)
D = Discriminator().to(device)
G = Generator().to(device)
```

## 8. Hàm mất mát và bộ tối ưu hóa

[48]

```python
print("Discriminator parameters:", sum(p.numel() for p in D.parameters() if p.requires_grad))
print("Generator parameters:", sum(p.numel() for p in G.parameters() if p.requires_grad))

# Set up the optimizers for Discriminator and Generator
# Adam is better than SGD for this task
#optimizerD = torch.optim.SGD(D.parameters(), lr=0.03)
#optimizerG = torch.optim.SGD(G.parameters(), lr=0.03)

# Uncomment to use Adam optimizer instead
optimizerD = torch.optim.Adam(D.parameters(), lr=0.0002)
optimizerG = torch.optim.Adam(G.parameters(), lr=0.0002)

# Define the loss function BCE (Binary Cross-Entropy)
criterion = nn.BCELoss()
batch_size = 128
```

```
Discriminator parameters: 100609
Generator parameters: 114064
```

## 9. Huấn luyện mô hình

### Nhập giá trị noise vector

[49]

```python
noise_dim = int(input("Nhập giá trị noise_dim: "))
print(f"Noise dimension: {noise_dim}")
```

```
Nhập giá trị noise_dim: 100
Noise dimension: 100
```

```python
import torch
import matplotlib.pyplot as plt

# Số epochs
epochs = 50

# Danh sách lưu loss
lossD_list = []
lossG_list = []

for epoch in range(epochs):
    lossD_epoch = 0
    lossG_epoch = 0
    num_batches = 0

    for i, data in enumerate(dataloader):
        x_real, _ = data
        x_real = x_real.to(device)
```

```python
        # Labels cho dữ liệu thật và giả
        lab_real = torch.ones((x_real.size(0), 1), device=device)
        lab_fake = torch.zeros((x_real.size(0), 1), device=device)

        # ---- Training Discriminator ----
        optimizerD.zero_grad()
        lossD_real = criterion(D(x_real), lab_real)

        z = torch.randn(x_real.size(0), noise_dim, device=device)
        x_gen = G(z).detach()
        lossD_fake = criterion(D(x_gen), lab_fake)

        lossD = lossD_real + lossD_fake
        lossD.backward()
        optimizerD.step()

        # ---- Training Generator ----
        optimizerG.zero_grad()
        z = torch.randn(x_real.size(0), noise_dim, device=device)
        x_gen = G(z)
        lossG = criterion(D(x_gen), lab_real)  # Generator muốn đánh lừa D

        lossG.backward()
        optimizerG.step()

        # Tính tổng loss trong epoch
        lossD_epoch += lossD.item()
        lossG_epoch += lossG.item()
        num_batches += 1
```

```
        optimizerG.zero_grad()
        z = torch.randn(x_real.size(0), noise_dim, device=device)
        x_gen = G(z)
        lossG = criterion(D(x_gen), lab_real)  # Generator muốn đánh lừa D

        lossG.backward()
        optimizerG.step()

        # Tính tổng loss trong epoch
        lossD_epoch += lossD.item()
        lossG_epoch += lossG.item()
        num_batches += 1

    # Lưu loss trung bình vào danh sách
    lossD_list.append(lossD_epoch / num_batches)
    lossG_list.append(lossG_epoch / num_batches)

    print(f"Epoch [{epoch+1}/{epochs}] - Loss D: {lossD_list[-1]:.4f}, Loss G: {lossG_list[-1]:.4f}")

# ---- Vẽ đồ thị ----
plt.figure(figsize=(8, 5))
plt.plot(range(1, epochs+1), lossD_list, label="Discriminator Loss", color="red")
plt.plot(range(1, epochs+1), lossG_list, label="Generator Loss", color="blue")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training Loss of GAN")
plt.legend()
plt.show()
```

```
... Epoch [1/50] - Loss D: 1.1026, Loss G: 0.6367
```

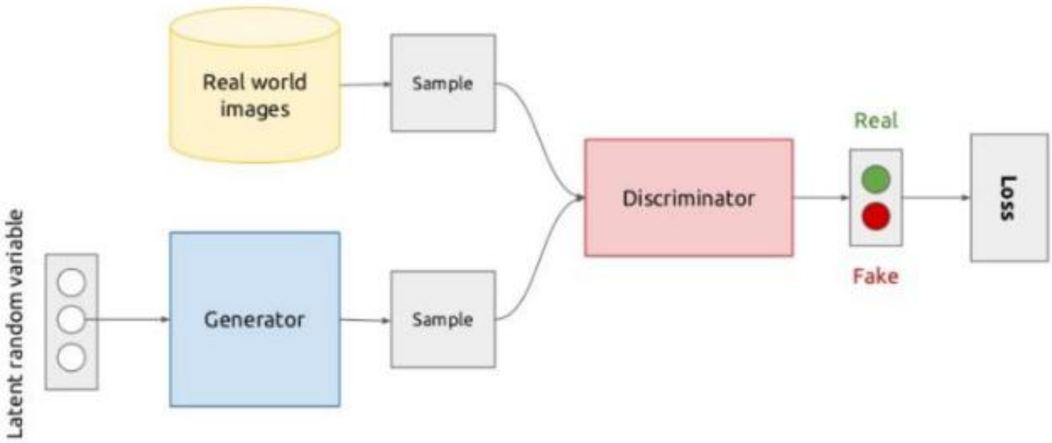
## 10. Sinh ảnh và hiển thị

```
from PIL import Image

def show_generated_images(G, num_images=16, noise_dim=100, device="cpu", save_path=None):
    G.eval()  # Đặt Generator ở chế độ đánh giá
    with torch.no_grad():
        z = torch.randn(num_images, noise_dim, device=device)
        fake_images = G(z).detach().cpu()

    grid = make_grid(fake_images, normalize=True, nrow=4)
    np_img = np.transpose(grid.numpy(), (1, 2, 0))  # Chuyển về định dạng ảnh

    plt.figure(figsize=(6, 6))
    plt.imshow(np_img)
    plt.axis("off")

    if save_path:
        image = Image.fromarray((np_img * 255).astype(np.uint8))  # Convert to image
        image.save(save_path)
        print(f"Ảnh đã lưu tại: {save_path}")

    plt.show()
    G.train()  # Đặt lại Generator về chế độ training

# Gọi hàm và lưu ảnh vào Google Drive
save_path = "/content/drive/My Drive/lab2-adip/generated_image.png"  # Thay đổi đường dẫn nếu cần
show_generated_images(G, device=device, save_path=save_path)
```

...

# III. Summarization of the usage

## Framework:




A generative adversarial network (GAN) uses two neural networks to compete with each other like in a game, one known as a "discriminator" and the other known as the "generator".

▪ The Generator wants to learn to generate realistic images that are indistinguishable from the real data. The input of the Generator is a Gaussian noise random sample, and its output is a generated data point

▪ The Discriminator wants to tell the real & fake images apart. The input of the Discriminator is a datapoint or an image, and its output is a probability assigned to the datapoint being real. It can be seen as a binary classifier

## Detailed Usage and Algorithm Explanation:

### 1. Generative Adversarial Network (GAN) Overview

GAN consists of two models:

- **Discriminator D(x):** Learns to classify real and fake images.
- **Generator G(z):** Learns to generate realistic images from random noise z.

The objective function of GAN is:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

where:

- $p_{\text{data}}(x)$ is the real data distribution.
- $p_z(z)$ is the noise distribution used to generate fake samples.

## 2. Discriminator (D) Implementation and Explanation

**Mathematical Formulation**

The discriminator is a binary classifier that takes an input xxx and outputs a probability D(x)∈[0,1] representing the likelihood that xxx is real. It is trained using **binary cross-entropy loss**:

$$\mathcal{L}_D = -\mathbb{E}_{x \sim p_{\text{data}}(x)}[\log D(x)] - \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

**Pseudo Code for Discriminator Training**

```
# Forward pass real images through Discriminator
real_output = D(real_images)
real_loss = criterion(real_output, torch.ones_like(real_output))

# Forward pass generated images through Discriminator
fake_images = G(noise)
fake_output = D(fake_images.detach())
fake_loss = criterion(fake_output, torch.zeros_like(fake_output))

# Compute total loss and backpropagate
lossD = real_loss + fake_loss
optimizerD.zero_grad()
lossD.backward()
optimizerD.step()
```

## 3. Generator (G) Implementation and Explanation

**Mathematical Formulation**

The generator learns to transform random noise zzz into realistic images. It aims to maximize D(G(z)) so that the discriminator classifies fake images as real:

$$\mathcal{L}_G = -\mathbb{E}_{z \sim p_z(z)}[\log D(G(z))]$$

To improve stability, we use the alternative loss:

$$\mathcal{L}_G = \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

**Pseudo Code for Generator Training**

```
# Generate fake images
fake_images = G(noise)
```

```
# Forward pass fake images through Discriminator
fake_output = D(fake_images)
lossG = criterion(fake_output, torch.ones_like(fake_output))  # Fool the discriminator

# Backpropagate
optimizerG.zero_grad()
lossG.backward()
optimizerG.step()
```

## 4. Training Process

**Overall Explanation**

The training process follows the standard GAN approach, where we alternately update the **Discriminator (D)** and **Generator (G)** in each iteration.

- **Step 1**: Train **Discriminator (D)** to distinguish real and fake images.
- **Step 2**: Train **Generator (G)** to generate realistic images and fool the discriminator.

This min-max game continues until the generator produces high-quality images.

**Step 1: Training the Discriminator**

**Goal:** The discriminator is trained to assign a high probability to real images and a low probability to generated images.

1. **Process real images**
   - Pass real images xxx through D(x)
   - Compute loss using **binary cross-entropy (BCE)** with label y=1 (real).
2. **Process fake images**
   - Generate fake images G(z) from random noise z
   - Pass them through D(G(z)
   - Compute loss using **BCE** with label y=0 (fake).
3. **Update Discriminator**
   - Compute total loss $\mathcal{L}_D.$
   - Perform **backpropagation** and update DDD parameters.

**Pseudo Code for Discriminator Training**

```
# Set Discriminator to training mode
```

```
D.train()

# Get real images from dataset
real_images, _ = next(iter(dataloader))
real_images = real_images.view(real_images.size(0), -1).to(device)

# Compute output for real images
real_output = D(real_images)
real_labels = torch.ones_like(real_output)  # Real label = 1
real_loss = criterion(real_output, real_labels)

# Generate fake images
noise = torch.randn(batch_size, z_dim).to(device)
fake_images = G(noise).detach()  # Stop gradient propagation to G
fake_output = D(fake_images)
fake_labels = torch.zeros_like(fake_output)  # Fake label = 0
fake_loss = criterion(fake_output, fake_labels)

# Compute total Discriminator loss
lossD = real_loss + fake_loss

# Backpropagate and update D
optimizerD.zero_grad()
lossD.backward()
optimizerD.step()
```

**Step 2: Training the Generator**

**Goal:** The generator learns to create realistic images so that the discriminator classifies them as real.

1. **Generate fake images**
   - Sample random noise zzz and pass it through G(z)
2. **Trick the Discriminator**
   - Pass generated images through D(G(z)
   - Instead of using label y=0, we use y=1 (pretend fake images are real).
   - Compute loss using **BCE** with label y=1 (fooling D).
3. **Update Generator**
   - Compute total loss $\mathcal{L}_G$.
   - Perform **backpropagation** and update GGG parameters.

**Pseudo Code for Generator Training**

```
# Set Generator to training mode
G.train()
# Generate fake images
noise = torch.randn(batch_size, z_dim).to(device)
fake_images = G(noise)
# Compute Discriminator's response to fake images
fake_output = D(fake_images)
fake_labels = torch.ones_like(fake_output)  # Fool D into thinking fake images are real
lossG = criterion(fake_output, fake_labels)
# Backpropagate and update G
optimizerG.zero_grad()
lossG.backward()
optimizerG.step()
```

**Step 3: Complete Training Loop**

**Goal:** Alternate between updating **D** and **G** for multiple epochs.

**Full Training Loop Pseudo Code**

```
for epoch in range(num_epochs):
    for real_images, _ in dataloader:
        # Train Discriminator
        real_images = real_images.view(real_images.size(0), -1).to(device)
        noise = torch.randn(batch_size, z_dim).to(device)
        fake_images = G(noise).detach()
        real_output = D(real_images)
        fake_output = D(fake_images)
        lossD_real = criterion(real_output, torch.ones_like(real_output))
        lossD_fake = criterion(fake_output, torch.zeros_like(fake_output))
        lossD = lossD_real + lossD_fake
        optimizerD.zero_grad()
        lossD.backward()
        optimizerD.step()
        # Train Generator
        noise = torch.randn(batch_size, z_dim).to(device)
        fake_images = G(noise)
```

```
        fake_output = D(fake_images)
        lossG = criterion(fake_output, torch.ones_like(fake_output)
        optimizerG.zero_grad()
        lossG.backward()
        optimizerG.step()
    # Print progress
    print(f"Epoch [{epoch+1}/{num_epochs}], LossD: {lossD.item()}, LossG: {lossG.item()}")
```

## 5. Result Visualization

**Goal:** After training, visualize the **generated images** to assess the quality of the generator.

1. **Generate images** from random noise zzz.
2. **Convert tensor images** to a grid for visualization.
3. **Display the images** using matplotlib.

**Summary of the Training Process**

1. **Training the Discriminator (D)**
    - Process real images and compute loss with y=1
    - Generate fake images and compute loss with y=0
    - Backpropagate and update D.
2. **Training the Generator (G)**
    - Generate fake images.
    - Compute loss by fooling D with y=1
    - Backpropagate and update G
3. **Repeat the process** for multiple epochs until the generator produces realistic images.
4. **Visualize generated images** to evaluate the performance.

## Dataset: MNIST

**1. Overview**

The **MNIST (Modified National Institute of Standards and Technology) dataset** is a widely used benchmark dataset for handwritten digit recognition. It consists of **70,000 grayscale images** of handwritten digits from **0 to 9**, where:

- **60,000 images** are used for training.
- **10,000 images** are used for testing.

Each image has a resolution of **28 × 28 pixels** and is stored in **grayscale (1 channel)**. The pixel values range from **0 (black) to 255 (white)**.

**2. Dataset Structure**

- **Image Size**: 28×2828 \times 2828×28 pixels
- **Number of Classes**: 10 (digits 0-9)
- **Color Mode**: Grayscale
- **Pixel Intensity Range**: 0-255 (normalized to 0-1 or -1 to 1 during preprocessing)
- **Data Split**:
    - **Training set**: 60,000 images
    - **Test set**: 10,000 images

**3. Dataset Preprocessing**

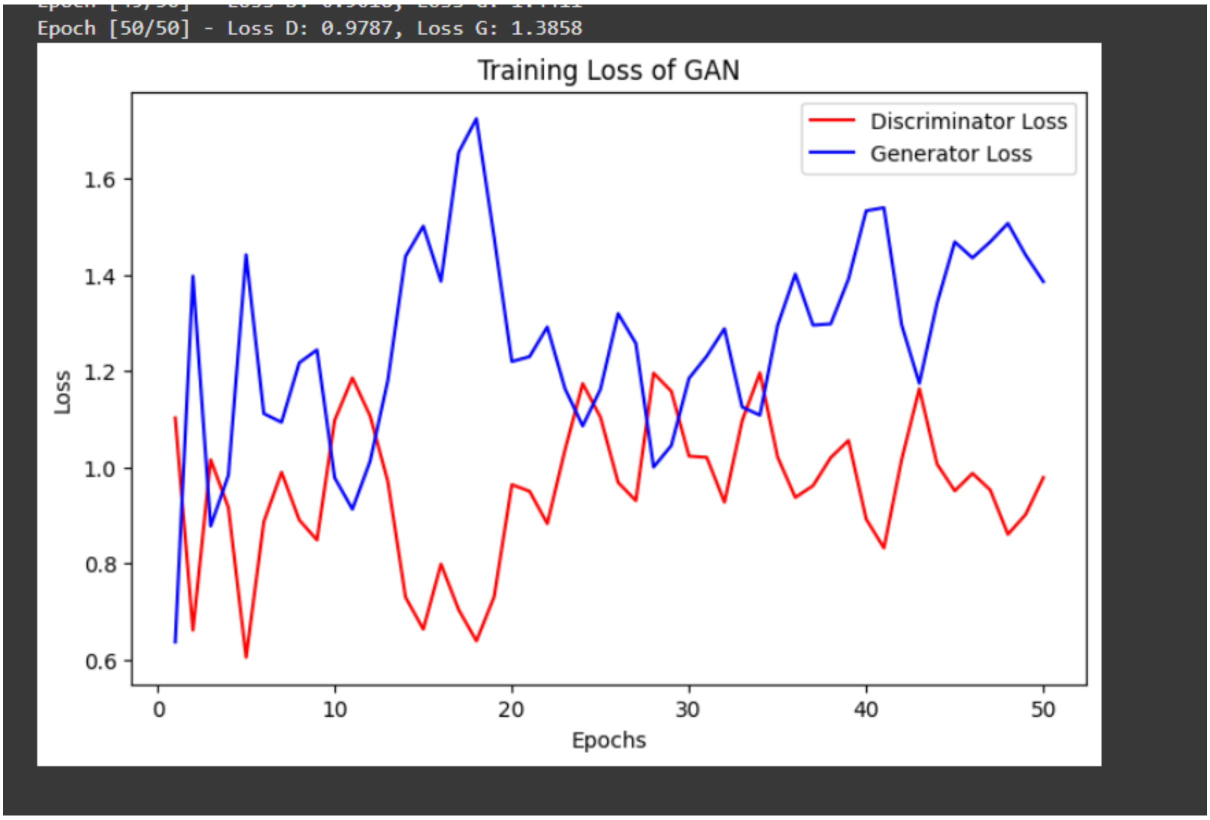
Before training, the images are preprocessed to **normalize** their pixel values and reshape them for input into the neural network.

**Preprocessing steps:**

1. Convert pixel values from **0-255 to 0-1** (or -1 to 1 for better stability).
2. Flatten each image into a **1D vector of 784 features** (28×2828 \times 2828×28).
3. Convert labels into tensors.

# IV. EXPERIMENTS AND EVALUATION:

# Epochs = 50, noise dim = 100:




## Observations:

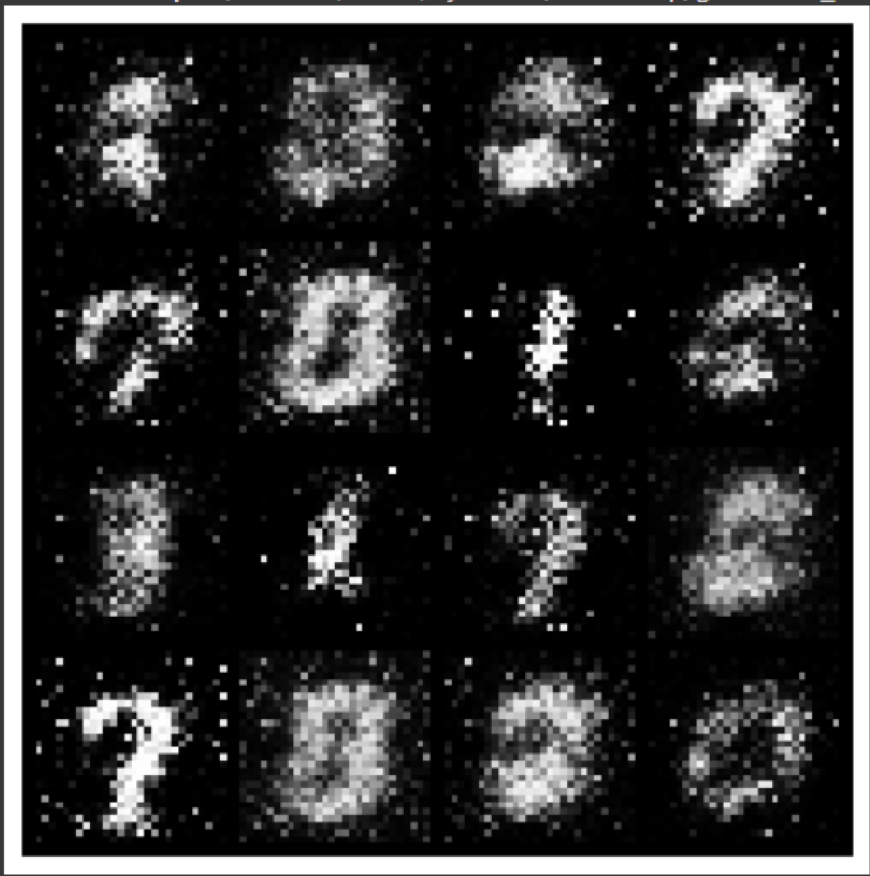
1. **Discriminator Loss (D Loss):** Fluctuates between **0.6 and 1.2**, without a clear downward trend. This suggests that the discriminator is neither overpowering the generator nor converging smoothly.
2. **Generator Loss (G Loss):** Ranges from **1.0 to 1.6**, with noticeable oscillations. A high generator loss may indicate that it struggles to produce realistic samples.
3. **Training Dynamics:**
    - In the early epochs, the generator improves as the discriminator loss decreases.
    - Around the mid-training phase, instability increases, and neither model dominates.
    - By epoch 50, both losses remain erratic, which may suggest training instability or insufficient convergence.

**Potential Issues & Fixes:**

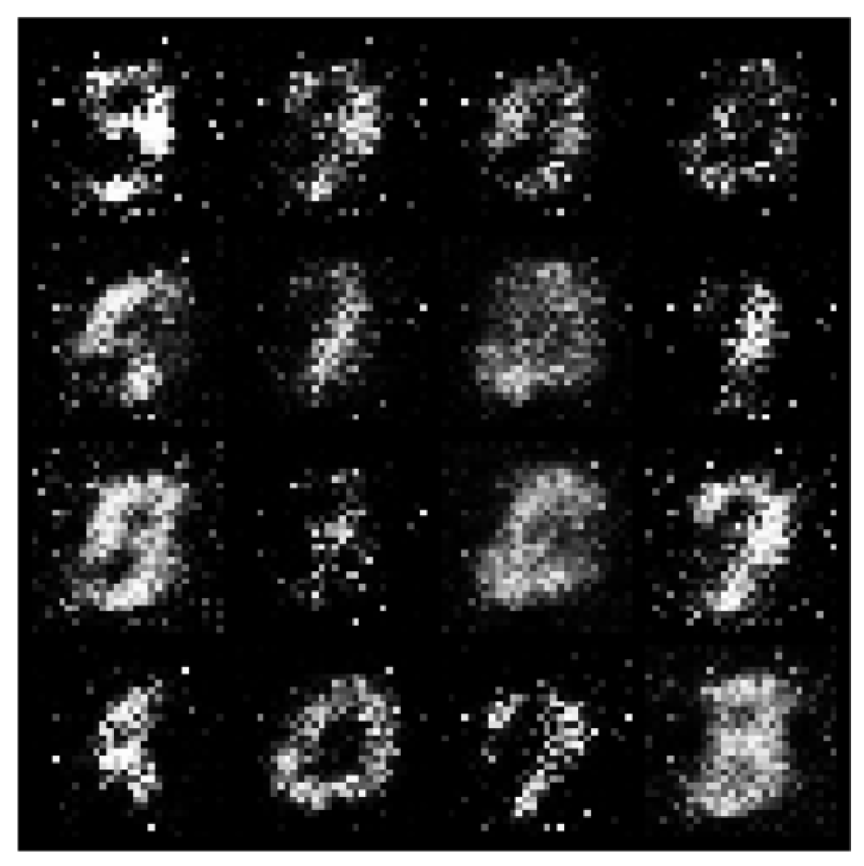
- **Unstable Training:** Loss fluctuations suggest the training process is not stable. Try **feature matching** or **label smoothing** to stabilize updates.
- **Mode Collapse:** If the generator produces limited variations, check generated images and consider techniques like **mini-batch discrimination**.
- **Hyperparameter Tuning:** Experiment with different **learning rates** (e.g., lowering generator LR) and **batch sizes** to improve stability.

**Results:**

Ảnh đã lưu tại: /content/drive/My Drive/lab2-adip/generated_image.png



Ảnh đã lưu tại: /content/drive/My Drive/lab2-adip/generated_image.png



## Analysis of Generated Images

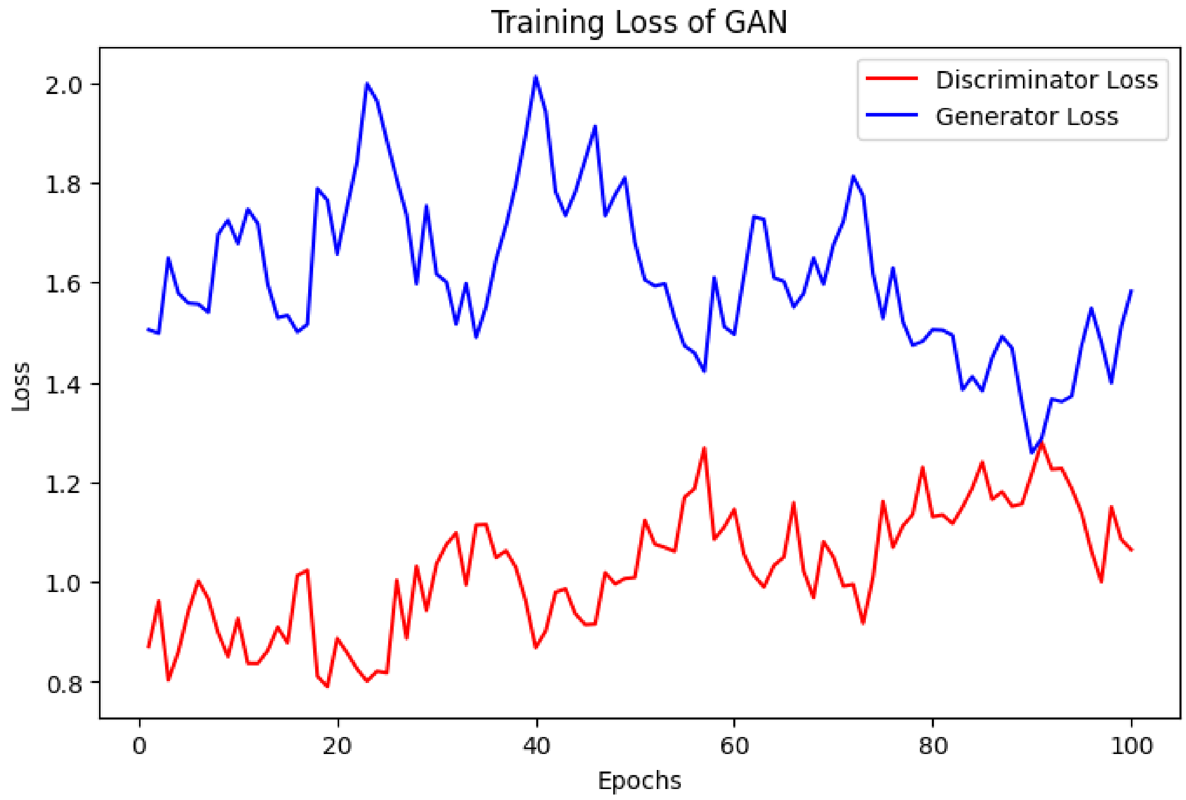
**Observations:**

1. **Blurry and Noisy Outputs:**
   - The digits are recognizable but very **blurry and grainy**, indicating that the generator is struggling to produce sharp images.
   - The background contains excessive **white noise**, suggesting unstable training or poor discriminator feedback.
2. **Mode Collapse Signs:**
   - Some digits appear **similar**, hinting that the generator may have collapsed to producing only a few variations.
   - Some numbers are incomplete or distorted (e.g., the digit "1" in the middle row is overly thin).
3. **Training Instability:**
   - The low quality of details suggests **generator-discriminator imbalance**.
   - The generator may not be learning meaningful features from the dataset.

**Potential Fixes:**

- **Use Batch Normalization or Spectral Normalization** to stabilize training.
- **Adjust Learning Rates:** Reduce the generator's LR to improve fine details
- **Use Feature Matching Loss:** Helps the generator learn better structures
- **Train Longer:** The model might need more epochs for convergence.

# Epochs = 100, noise dim = 100



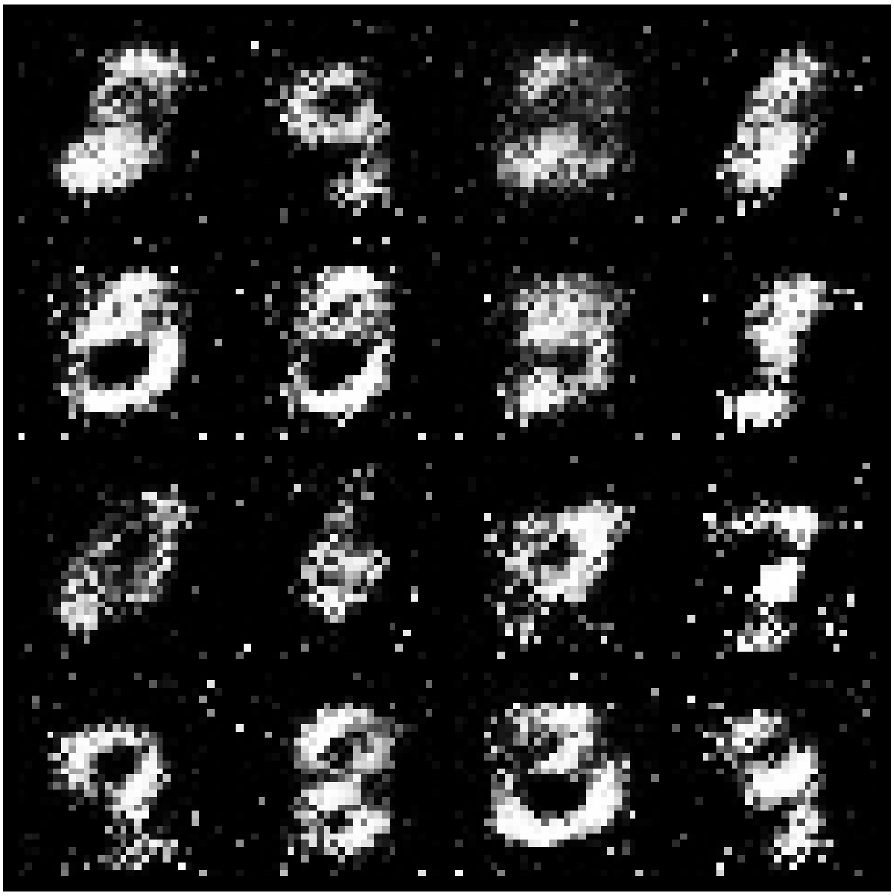

## Observations

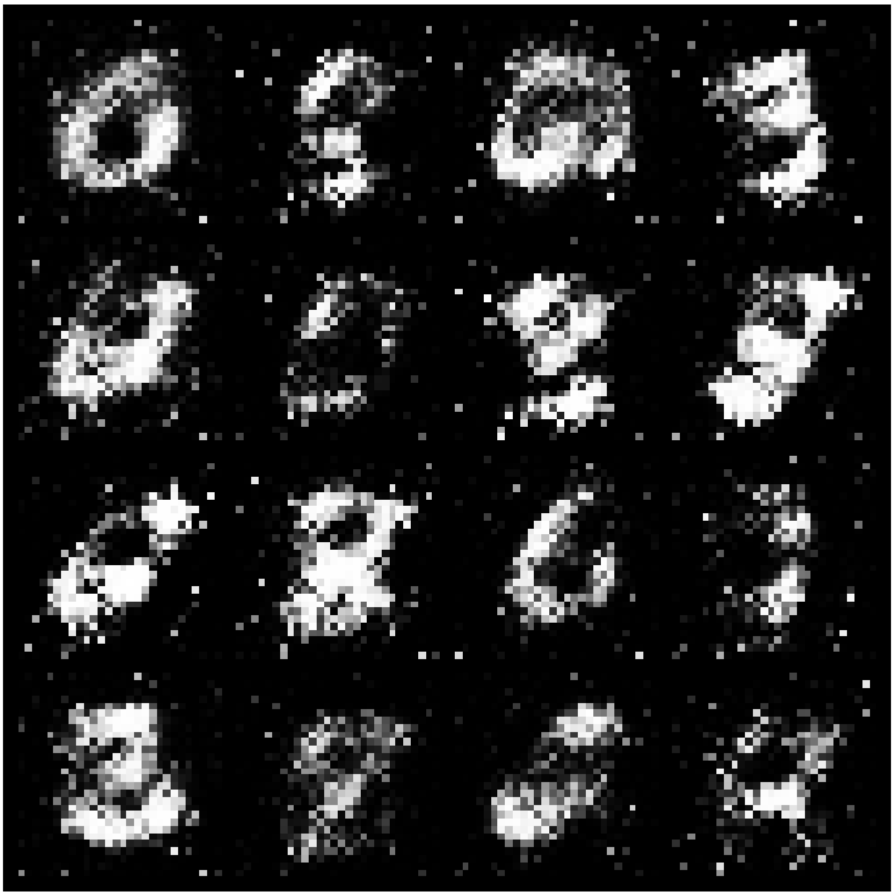
**1. Training Loss Analysis:**

- **Discriminator Loss (D Loss):** Fluctuates between 0.8 and 1.2, showing no clear downward trend. This suggests that the discriminator is neither overpowering the generator nor stabilizing effectively.
- **Generator Loss (G Loss):** Ranges from 1.4 to 2.0, with significant oscillations. The high loss indicates that the generator struggles to produce realistic samples.
- **Training Dynamics:**
    - In the early epochs, both losses fluctuate, indicating initial learning.
    - Around the mid-training phase, instability increases, and neither model dominates.
    - By epoch 100, both losses remain highly erratic, suggesting unstable training or insufficient convergence.

**Potential Issues & Fixes:**

- **Unstable Training:** Loss fluctuations indicate instability. Try **feature matching** or **label smoothing** to stabilize updates.
- **Mode Collapse:** If the generator produces limited variations, check for repetition in generated images and consider **mini-batch discrimination**.
- **Hyperparameter Tuning:**
  - Reduce **learning rate** for the generator to prevent oscillations.
  - Experiment with **batch size** to balance updates.





## Analysis of Generated Images

**1. Blurry and Noisy Outputs:**

- The digits are somewhat recognizable but appear **blurry and grainy**, indicating that the generator struggles to refine details.
- The background contains excessive **white noise**, suggesting unstable training or poor discriminator feedback.

**2. Signs of Mode Collapse:**

- Some digits appear **similar**, meaning the generator might be producing only a few variations.
- Several numbers are **incomplete or distorted** (e.g., digit "1" in the middle row looks overly thin).

**3. Training Instability:**

- The low quality of generated details suggests a **generator-discriminator imbalance**.
- The generator may not be **learning meaningful features** from MNIST.

**Potential Fixes:**

- **Use Batch Normalization** or **Spectral Normalization** to stabilize training.
- **Adjust Learning Rates:** Reduce the **generator's learning rate** to improve fine details.
- **Use Feature Matching Loss:** Helps the generator learn better structures.

- **Train Longer:** The model might need **more epochs** to reach better convergence.

## Comparison Between Custom Code and Lab Code & Its Impact on Results

**1. Comparison of Discriminator Architecture**

| Component | Custom Code | Lab Code |
|---|---|---|
| **Parameter Initialization** | Uses nn.Parameter with torch.randn | Uses nn.Linear |
| **Parameter Management** | Manually manages weight matrices and biases | nn.Linear automatically manages them |
| **Hidden Activation** | torch.maximum(0.2 * h, h) (manual LeakyReLU) | nn.LeakyReLU(0.2) (built-in) |
| **Number of Layers** | 2 layers (w1, w2) | 2 layers (fc1, fc2) |
| **Output Activation** | torch.sigmoid | torch.sigmoid |

**Impact on Performance**

- **Custom Code**: Since nn.Parameter is manually declared, weight updates and backpropagation may not be as optimized as nn.Linear, which can affect training stability.
- **Lab Code**: nn.Linear automatically manages parameters, reducing errors due to incorrect tensor size calculations or manual updates.
- **Efficiency**: Using nn.Linear is computationally more efficient due to PyTorch's internal optimizations, leading to smoother training.

**2. Comparison of Generator Architecture**

| Component | Custom Code | Lab Code |
|---|---|---|
| **Parameter Initialization** | Uses nn.Parameter with torch.randn | Uses nn.Linear |
| **Parameter Management** | Manually manages weight matrices and biases | nn.Linear automatically manages them |
| **Hidden Activation** | torch.maximum(0.2 * h, h) (manual LeakyReLU) | nn.LeakyReLU(0.2) (built-in) |
| **Number of Layers** | 2 layers (w1, w2) | 2 layers (fc1, fc2) |
| **Output Activation** | torch.tanh | torch.tanh |

**Impact on Performance**

- **Custom Code**:
  - Manually managing parameters with nn.Parameter can make weight updates less optimized, potentially causing unstable training.

    - The use of torch.matmul(x, self.w1) + self.b1 instead of nn.Linear might slow down training due to lack of PyTorch's internal optimizations.
- **Lab Code**:
    - nn.Linear ensures better computational efficiency, leading to smoother forward and backward propagation.
    - Easier to maintain and less prone to errors.
- **Efficiency**: Like the Discriminator, using nn.Linear improves computational efficiency and ensures optimal training performance.

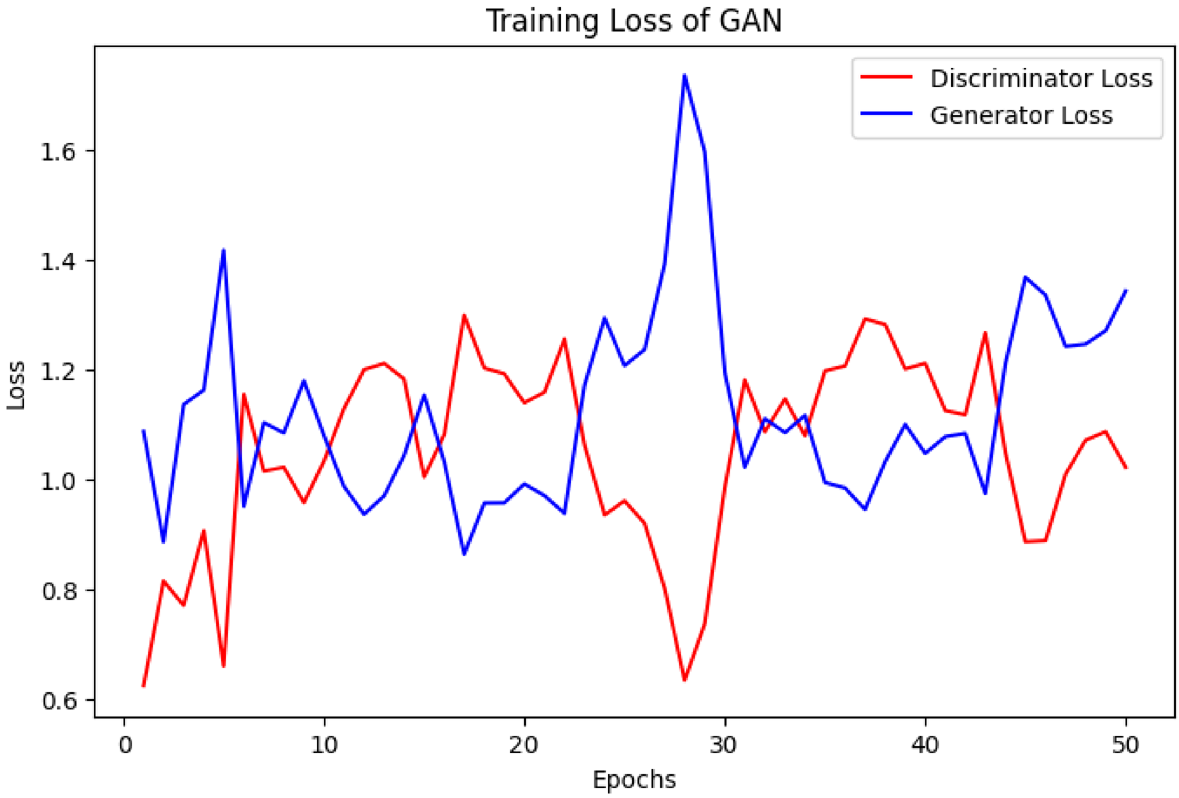
**3. Overall Impact on Training and Performance**

| Factor | Custom Code | Lab Code |
|---|---|---|
| **Training Speed** | Slower due to manual parameter management | Faster due to nn.Linear optimizations |
| **Training Stability** | May be less stable due to manual updates | More stable due to PyTorch's built-in mechanisms |
| **Readability & Maintainability** | More difficult due to manual parameter handling | Easier to maintain with nn.Linear |
| **Flexibility & Generalization** | More error-prone when modifying architecture | More adaptable to changes |

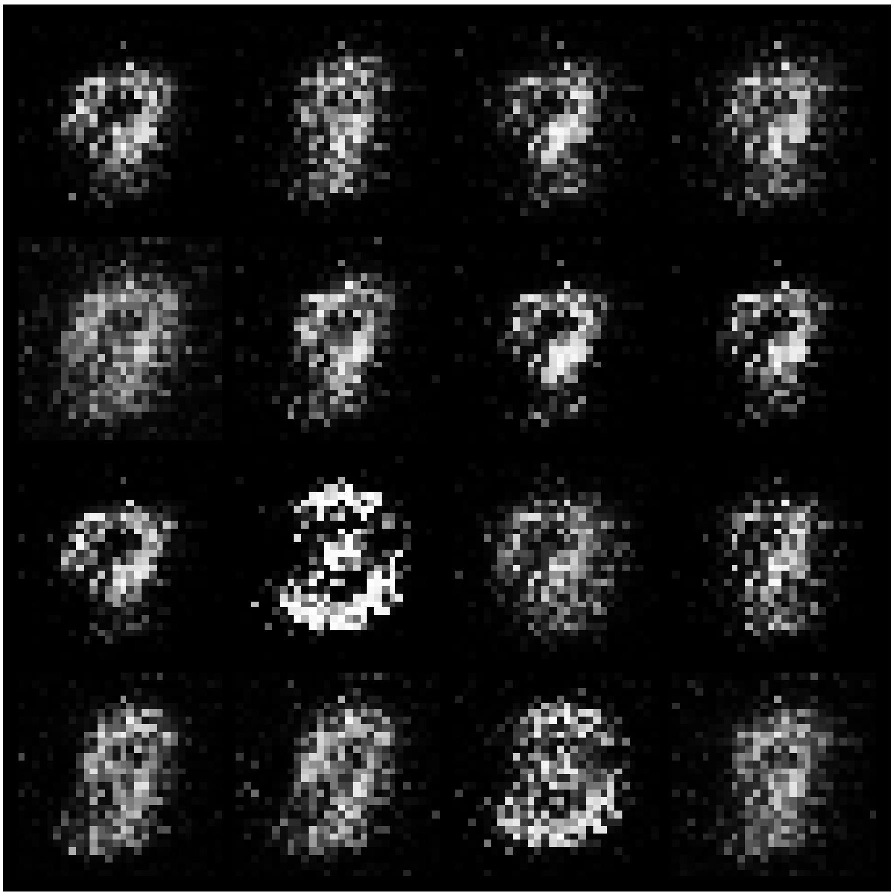
**Conclusion**

- **Lab Code is more efficient in terms of speed, stability, and maintainability.**
- **Custom Code is useful for understanding GAN operations at a lower level but is harder to optimize and prone to errors.**
- **Using nn.Linear leads to faster and more stable training.**

**Results:**



Training Loss of GAN
Discriminator Loss
Generator Loss
Loss
Epochs
1.6
1.4
1.2
1.0
0.8
0.6
0
10
20
30
40
50

## Analysis of GAN Training Loss

### 1. Observations

- Both generator and discriminator losses fluctuate significantly, showing instability.
- No clear convergence; sharp spikes suggest training imbalance.

### 2. Potential Issues

- **Mode Collapse:** The generator may be producing limited outputs, causing loss oscillations.
- **Unstable Training:** High variance in updates, possibly due to learning rate or batch size settings.