**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% |

1. **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)

1. **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**:

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A screen shot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A black rectangular with green text

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A screen shot of a computer program

Description automatically generated** **A screenshot of a computer program

Description automatically generated**

**III. Summarization of the usage**

**Framework:  
A diagram of a discriminator

Description automatically generated**

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.

**A white background with black text

Description automatically generated**

**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**:



**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:

A math equations on a white background

Description automatically generated

**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 
   * 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 
   * 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:**

**A graph of a graph

Description automatically generated with medium confidence**

**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:**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**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**

A graph of a graph of two people

Description automatically generated with medium confidence

**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.

A collage of images of feet

Description automatically generated

A collage of images of a heart

Description automatically generated

**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:  
A graph of a graph

Description automatically generated**

**A collage of images of a person's face

Description automatically generated**

**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.