Generative Adversarial Network (GAN) for MNIST Image Generation

Introduction

This lab focuses on the implementation of a Generative Adversarial Network (GAN) using the KNIME Analytics Platform with embedded Python scripts. The objective is to train a GAN on the MNIST handwritten digits dataset in order to generate realistic synthetic digit images. The GAN consists of two neural networks trained adversarially: a generator that creates fake images from random noise, and a discriminator that distinguishes between real and fake images.

a) Load and Prepare the MNIST Dataset

The MNIST dataset contains grayscale images of handwritten digits from 0 to 9, each with a resolution of 28×28 pixels.

The dataset is loaded using the torchvision library. Since GAN training is unsupervised, class labels are not used during training.

To be compatible with the generator's output activation function (tanh), pixel values are normalized from the range [0,1] to [−1,1].

Batch training is enabled using a batch size of 128 to improve training stability.

Data Normalization

The following transformation is applied:

- Conversion to tensor
- Normalization using mean = 0.5 and standard deviation = 0.5

This maps pixel values as follows: [0,1] ⟶ [−1,1]

Sample Visualization

A subset of MNIST images is visualized after normalization (mapped back to [0,1] for display) to understand the data characteristics.

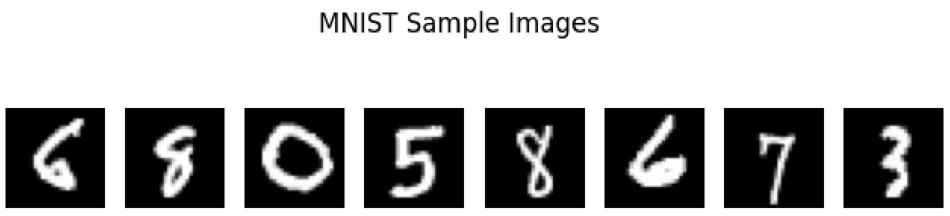

Figure 1: Sample images from the MNIST dataset after normalization.

—

b) Generator Architecture Design

The generator network takes as input a latent vector of dimension 100 sampled from a standard normal distribution.

It uses fully connected (dense) layers to progressively upscale the latent vector into a 28×28 image.

Architecture Details

- Input: 100-dimensional noise vector

- Hidden layers: Fully connected layers with ReLU activation
- Output layer: Fully connected layer with tanh activation
- Output shape: 28×28×1 (flattened to 784 values)

The tanh activation ensures that generated pixel values lie in the range [−1,1], consistent with the normalized MNIST data.

—

c) Discriminator Architecture Design

The discriminator receives either real MNIST images or fake images generated by the generator and outputs a single probability indicating whether the image is real or fake.

Architecture Details

- Input: Flattened 28×28 image (784 values)
- Hidden layers: Fully connected layers with LeakyReLU activation
- Dropout is applied to prevent the discriminator from becoming too strong too early
- Output layer: Sigmoid activation producing a probability in [0,1]

—

d) Adversarial Training Configuration

Binary Cross-Entropy (BCE) loss is used for both generator and discriminator.

Separate Adam optimizers are configured:
- Learning rate: 0.0002
- One optimizer for the generator
- One optimizer for the discriminator

Training follows a two-phase update strategy:
1. Train the discriminator using real images (label = 1) and fake images (label = 0)
2. Train the generator while keeping the discriminator fixed, encouraging it to fool the discriminator

Training is performed for 50 epochs using a batch size of 128.

—

e) Training Process and Stability Monitoring

During training:
- The discriminator learns to distinguish real MNIST images from generated ones
- The generator learns to produce images that the discriminator classifies as real

Both generator and discriminator losses are recorded after each epoch.

Loss Tracking

The evolution of losses over epochs is exported from the Python Script node and visualized in KNIME using a Line Plot or JavaScript Line Chart node.

This allows monitoring:
- Training stability
- Potential divergence (high oscillations)
- Signs of mode collapse (loss stagnation)

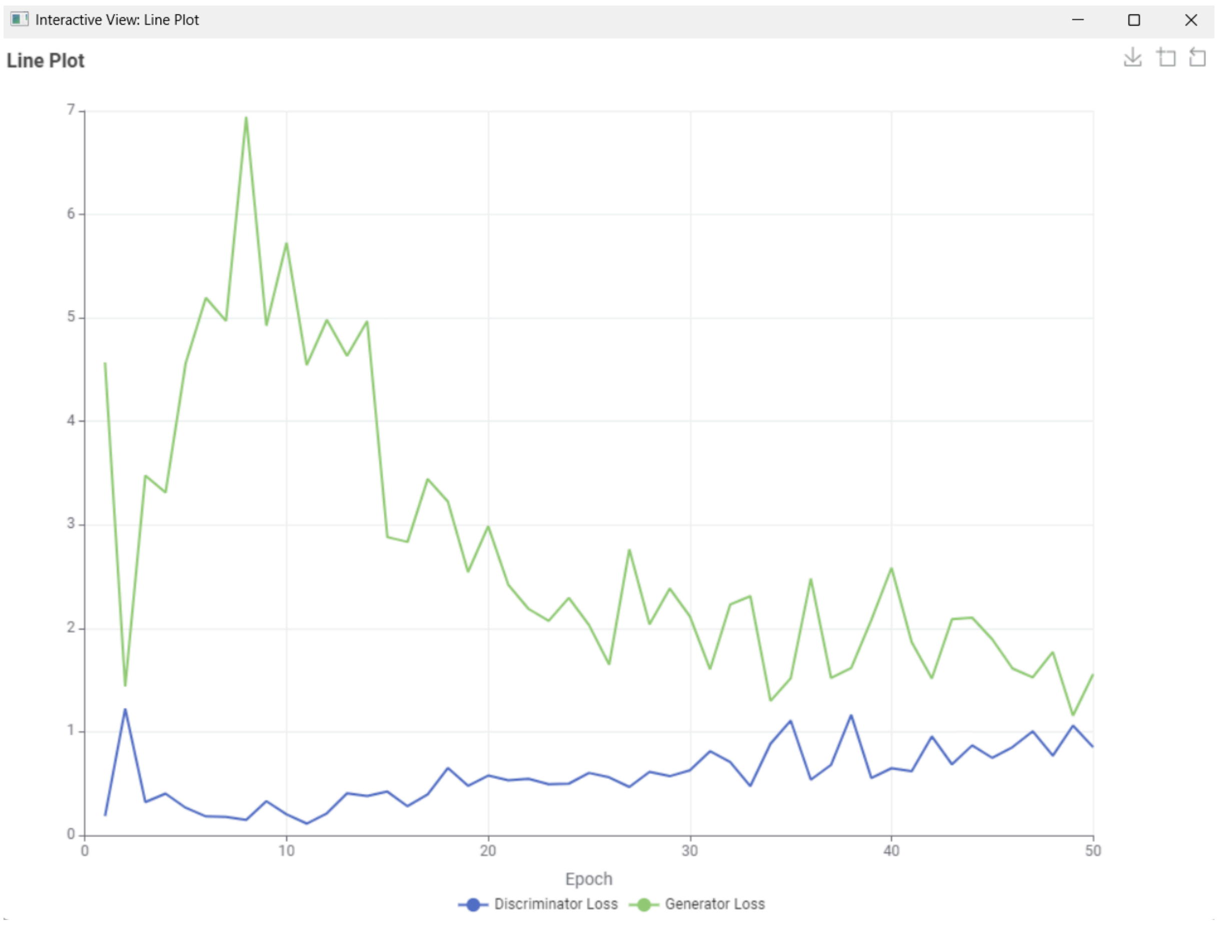

Figure 2: Generator and discriminator loss curves over training epochs.

Periodic Image Saving

To visually track training progress, generated images are saved every 5 epochs using a fixed noise vector. This allows direct comparison of image quality improvements across epochs.

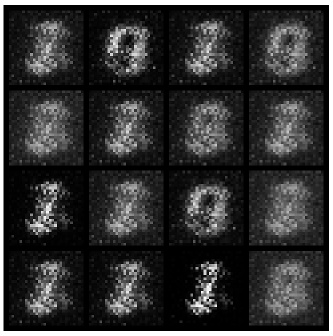

Figure 3: Generated images at early training stage (epoch 5).

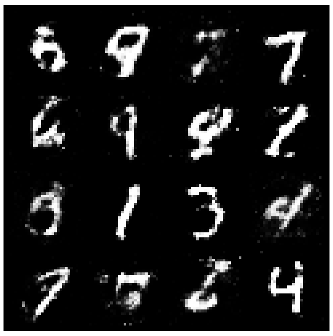
Figure 4: Generated images at later training stage (epoch 50).

—

f) Evaluation of Generated Images

After training, the generator is evaluated by producing a large batch of synthetic images using randomly sampled noise vectors.

A grid of 64 generated images is visualized to assess:

- Visual quality
- Diversity of generated digits
- Coverage of different digit classes (0–9)

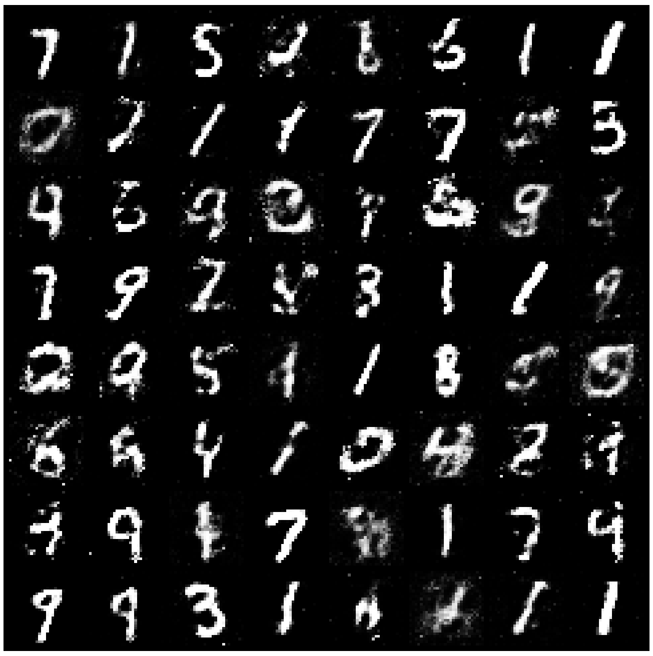
Figure 5: Grid of synthetic MNIST digits generated by the trained GAN.

Quality Assessment

The generated digits are clearly recognizable and show significant improvement compared to early training stages.

The diversity of shapes and styles indicates that the generator does not suffer from severe mode collapse.

Comparing early-epoch samples with final outputs demonstrates a clear enhancement in image sharpness and structure, confirming successful adversarial training.

KNIME Workflow Implementation

The implementation of the GAN is carried out using the KNIME Analytics Platform, which integrates Python scripts to enable deep learning model development.

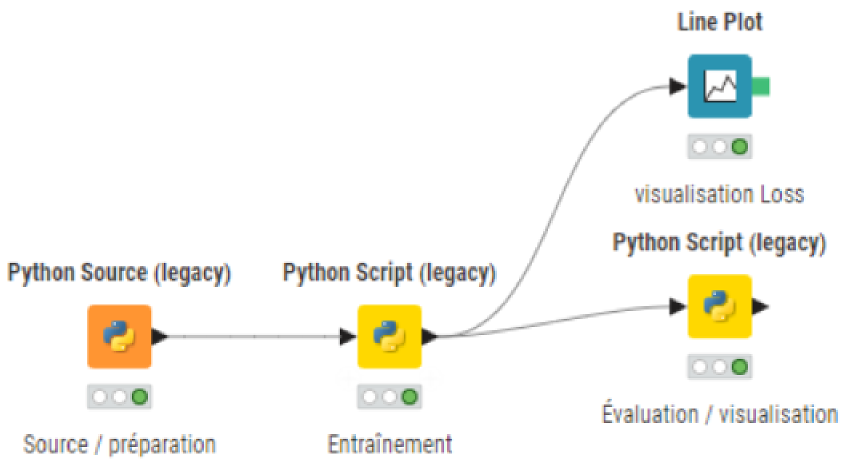

Figure 6: KNIME workflow for GAN-based MNIST image generation.

Workflow Description

The workflow starts with a Python Source node responsible for loading the MNIST dataset, defining the generator and discriminator architectures, and configuring the adversarial training process using PyTorch.

```python
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import pandas as pd

BATCH_SIZE = 128
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

mnist = datasets.MNIST(
    root="./data",
    train=True,
    download=True,
    transform=transform
)

dataloader = DataLoader(mnist, batch_size=BATCH_SIZE, shuffle=True)

# Visualisation
images, _ = next(iter(dataloader))
images_vis = (images[:8] + 1) / 2

plt.figure(figsize=(8,2))
for i in range(8):
    plt.subplot(1,8,i+1)
    plt.imshow(images_vis[i].squeeze(), cmap="gray")
    plt.axis("off")
plt.show()

# KNIME OUTPUT TABLE
output_table = pd.DataFrame({
    "Info": ["MNIST loaded"],
    "Batch size": [BATCH_SIZE],
    "Samples": [len(mnist)]
})
```

The training process is implemented in subsequent Python Script nodes. These nodes handle the adversarial training loop, including discriminator and generator updates, loss computation using binary cross-entropy, and periodic saving of generated samples every few epochs to visually monitor training progress.

```
# IMPORTS
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import pandas as pd
import os
from torchvision.utils import save_image

# DEVICE
device = "cuda" if torch.cuda.is_available() else "cpu"

# PARAMETERS
BATCH_SIZE = 128
LATENT_DIM = 100
IMG_DIM = 28 * 28
EPOCHS = 50
LR = 0.0002

# MNIST TRANSFORM  [0,1] → [-1,1]  (for tanh)

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# LOAD MNIST DATASET (unsupervised)
mnist = datasets.MNIST(
    root="./data",
    train=True,
    download=False,
    transform=transform
)

dataloader = DataLoader(
    mnist,
    batch_size=BATCH_SIZE,
    shuffle=True
)
os.makedirs("generated_samples", exist_ok=True)
```

```python
# Bruit fixe pour comparer les progrès
fixed_noise = torch.randn(16, LATENT_DIM).to(device)

# GENERATOR
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(LATENT_DIM, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, IMG_DIM),
            nn.Tanh()
        )

    def forward(self, z):
        return self.model(z)

# DISCRIMINATOR
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(IMG_DIM, 512),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.model(x)


# INITIALIZE MODELS
generator = Generator().to(device)
discriminator = Discriminator().to(device)

criterion = nn.BCELoss()
opt_gen = optim.Adam(generator.parameters(), lr=LR)
opt_disc = optim.Adam(discriminator.parameters(), lr=LR)

# TRAINING LOOP
losses = []

for epoch in range(EPOCHS):
    for real_imgs, _ in dataloader:

        real_imgs = real_imgs.view(-1, IMG_DIM).to(device)
        batch_size = real_imgs.size(0)

        real_labels = torch.ones(batch_size, 1).to(device)
        fake_labels = torch.zeros(batch_size, 1).to(device)
        # -------- Train Discriminator --------
        noise = torch.randn(batch_size, LATENT_DIM).to(device)
        fake_imgs = generator(noise)

        loss_real = criterion(discriminator(real_imgs), real_labels)
        loss_fake = criterion(discriminator(fake_imgs.detach()), fake_labels)
        disc_loss = loss_real + loss_fake

        opt_disc.zero_grad()
        disc_loss.backward()
        opt_disc.step()

        # -------- Train Generator --------
        noise = torch.randn(batch_size, LATENT_DIM).to(device)
        fake_imgs = generator(noise)
        gen_loss = criterion(discriminator(fake_imgs), real_labels)

        opt_gen.zero_grad()
        gen_loss.backward()
        opt_gen.step()

    losses.append([epoch + 1, disc_loss.item(), gen_loss.item()])
        # SAVE GENERATED IMAGES EVERY 5 EPOCHS
    if (epoch + 1) % 5 == 0:
        with torch.no_grad():
            fake_imgs = generator(fixed_noise)

            # reshape to image format
            fake_imgs = fake_imgs.view(-1, 1, 28, 28)

            # [-1,1] → [0,1] for visualization
            fake_imgs = (fake_imgs + 1) / 2

            save_image(
                fake_imgs,
                f"generated_samples/epoch_{epoch+1}.png",
                nrow=4
            )

# SAVE GENERATOR (for next node)
torch.save(generator.state_dict(), "generator.pth")
# KNIME OUTPUT TABLE (MANDATORY)
output_table_1 = pd.DataFrame(
    losses,
    columns=["Epoch", "Discriminator Loss", "Generator Loss"]
)
```

A Line Plot node is then used to visualize the evolution of the generator and discriminator losses across training epochs. This visualization allows monitoring training stability, detecting divergence, and identifying potential mode collapse.

Finally, the last Python Script node is dedicated to the evaluation phase. In this node, the trained generator model is loaded and used to generate a large batch of synthetic MNIST images from randomly sampled noise vectors. This step provides a final qualitative assessment of the generator's performance and the overall success of the GAN training.

```python
import torch
import torch.nn as nn
from torchvision.utils import save_image
import os
import pandas as pd

device = "cuda" if torch.cuda.is_available() else "cpu"
LATENT_DIM = 100
N_IMAGES = 64  # grand batch

# REDEFINITION DU GENERATOR
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(LATENT_DIM, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 28 * 28),
            nn.Tanh()
        )

    def forward(self, z):
        return self.model(z)

# LOAD TRAINED GENERATOR
generator = Generator().to(device)
generator.load_state_dict(torch.load("generator.pth", map_location=device))
generator.eval()

# GENERATE IMAGES
os.makedirs("final_samples", exist_ok=True)
noise = torch.randn(N_IMAGES, LATENT_DIM).to(device)
with torch.no_grad():
    fake_imgs = generator(noise)
    fake_imgs = fake_imgs.view(-1, 1, 28, 28)
    fake_imgs = (fake_imgs + 1) / 2  # [-1,1] → [0,1]

save_image(fake_imgs, "final_samples/generated_grid.png", nrow=8)
output_table_1 = pd.DataFrame({
    "Status": ["Images generated successfully"]
})
```

Conclusion

In this lab, a Generative Adversarial Network was successfully implemented and trained on the MNIST dataset using KNIME and Python.

The generator learned to produce realistic handwritten digits, while the discriminator effectively guided the training process.

Loss monitoring and periodic image saving proved essential for evaluating training stability and visual quality.

This experiment demonstrates the effectiveness of GANs for unsupervised image generation tasks.