# 10. BUILD AND TRAIN A GAN FOR GENERATING HAND-WRITTEN DIGITS

|  |  |
| --- | --- |
| **EX.N0 : 10** | **BUILD AND TRAIN A GAN FOR GENERATING HAND- WRITTEN DIGITS** |
| **DATE : 08/04/2025** |

**AIM:**

To build and train a Generative Adversarial Network (GAN) for generating hand-written digits using the MNIST dataset.

# ALGORITHM:

Step 1: Import required libraries (TensorFlow, Keras, NumPy, Matplotlib). Step 2: Load and preprocess the MNIST dataset for training.

Step 3: Build the Generator and Discriminator models using Keras. Step 4: Define the loss functions and optimizers for both models.

Step 5: Train the GAN by alternately training the discriminator and generator. Step 6: Generate and visualize synthetic digit images.

# PROGRAM:

import torch

import torch.nn as nn import torchvision

import torchvision.transforms as transforms import matplotlib.pyplot as plt

import numpy as np

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

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

train\_loader = torch.utils.data.DataLoader( torchvision.datasets.MNIST('.', train=True, download=True, transform=transform), batch\_size=128, shuffle=True )

class Generator(nn.Module):

def init (self):

super(). init () self.model = nn.Sequential(

nn.Linear(100, 256), nn.ReLU(True),

nn.Linear(256, 512), nn.ReLU(True), nn.Linear(512, 784), nn.Tanh() )

def forward(self, x): return self.model(x).view(-1, 1, 28, 28) class Discriminator(nn.Module):

def init (self):

super(). init () self.model = nn.Sequential( nn.Flatten(),

nn.Linear(784, 512), nn.LeakyReLU(0.2), nn.Linear(512, 1), nn.Sigmoid() )

def forward(self, x): return self.model(x) generator = Generator().to(device) discriminator = Discriminator().to(device) criterion = nn.BCELoss()

optimizer\_G = torch.optim.Adam(generator.parameters(), lr=0.0002) optimizer\_D = torch.optim.Adam(discriminator.parameters(), lr=0.0002) epochs = 30

fixed\_noise = torch.randn(64, 100, device=device) for epoch in range(epochs):

for imgs, \_ in train\_loader:

imgs = imgs.to(device) batch\_size = imgs.size(0)

real\_labels = torch.ones(batch\_size, 1, device=device) fake\_labels = torch.zeros(batch\_size, 1, device=device) z = torch.randn(batch\_size, 100, device=device) fake\_imgs = generator(z)

d\_loss = criterion(discriminator(imgs), real\_labels) + criterion(discriminator(fake\_imgs.detach()), fake\_labels)

optimizer\_D.zero\_grad() d\_loss.backward() optimizer\_D.step()

g\_loss = criterion(discriminator(fake\_imgs), real\_labels) optimizer\_G.zero\_grad()

g\_loss.backward() optimizer\_G.step()

print(f"Epoch [{epoch+1}/{epochs}], D Loss: {d\_loss.item():.4f}, G Loss: {g\_loss.item():.4f}") if (epoch + 1) % 5 == 0:

with torch.no\_grad():

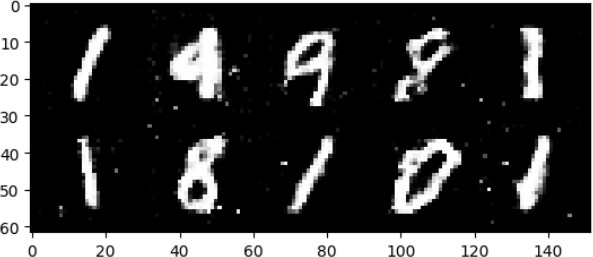
generated = generator(fixed\_noise).cpu()

grid = torchvision.utils.make\_grid(generated, nrow=8, normalize=True) plt.imshow(np.transpose(grid, (1, 2, 0)))

plt.title(f"Epoch {epoch+1}") plt.axis("off")

plt.show()

# OUTPUT:

****

**RESULT:**

Thus the Program has been executed successfully and verified.