SYNTHETIC DIGITS GENERATOR VANILLAGAN

Problem Statement:

To generate realistic handwritten digit images synthetically by learning the data distribution of the MNIST dataset using a Vanilla Generative Adversarial Network (GAN).

Project Scope

- Generates synthetic handwritten digit images (28×28 grayscale).
- Focuses on Vanilla GAN architecture for educational clarity.
- Supports training, evaluation, inference, and deployment.

System Requirements:

- Operating System: Windows / macOS / Linux
- Python Version: 3.9 or higher
- RAM: Minimum 8 GB (16 GB recommended)
- GPU: Optional (CPU-only supported)
- Disk Space: ~2 GB for datasets, logs, and models

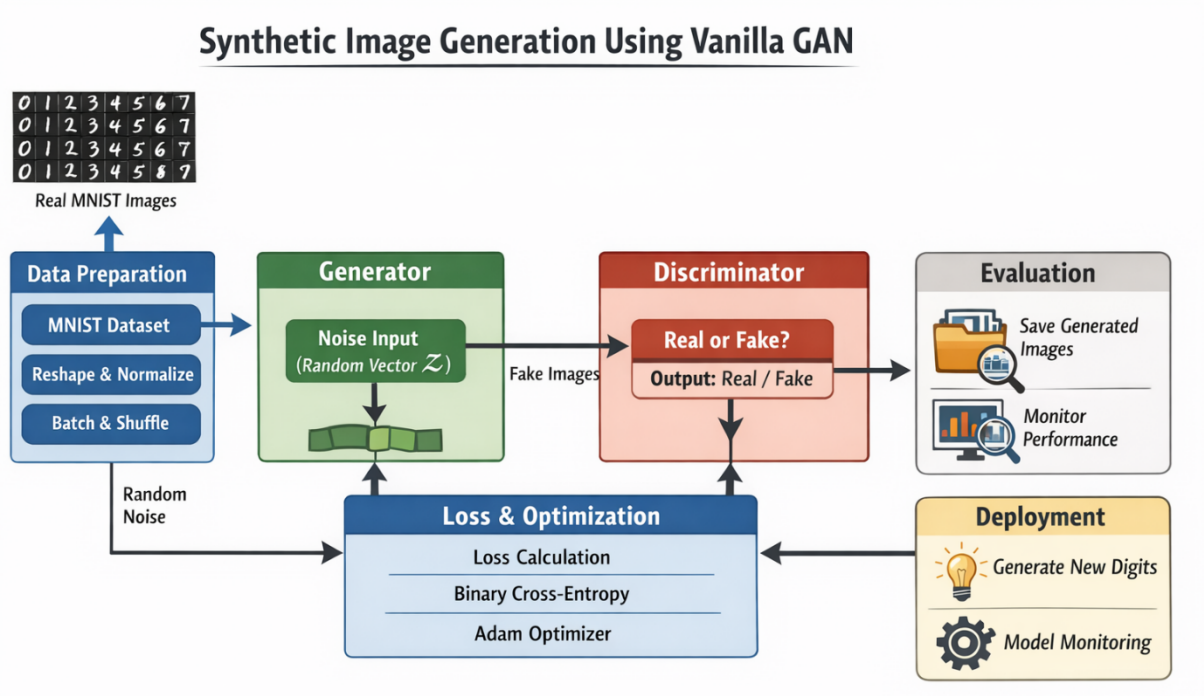
requirements.txt

```
torch
torchvision
numpy
matplotlib
scipy
scikit-learn
pillow
pyyaml
tqdm
tensorboard
streamlit
fastapi
uvicorn
opencv-python
```

config.yaml

```
image_size: 28
channels: 1
noise_dim: 100
batch_size: 64
epochs: 50
lr: 0.0002
beta1: 0.5
train_split: 0.95
normalization: tanh
```

Architecture:



I. Module 1: Data Pipeline

Dataset Description

- Dataset: MNIST Handwritten Digits
- Total Images: 70,000
- Image Size: 28×28 pixels
- Channels: 1 (Grayscale)
- Classes: Digits 0–9
- Source: torchvision.datasets.MNIST

What happens in this module

- Load the MNIST dataset (handwritten digits).
- Reshape images to 28×28×1 format.
- Normalize pixel values from [0, 255] → [-1, 1].
- Shuffle and batch the dataset for efficient training.

data_loader.py

```
import yaml
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, random_split
import torch


def get_dataloaders():
    with open("config.yaml") as f:
        cfg = yaml.safe_load(f)
    transform = transforms.Compose([
        transforms.Resize(cfg["image_size"]),
```

```
        transforms.ToTensor(),
        transforms.Normalize((0.5,), (0.5,))
    ])
    dataset = torchvision.datasets.MNIST(
        root="data/raw", train=True, download=True, transform=transform
    )
    train_size = int(len(dataset) * cfg["train_split"])
    test_size = len(dataset) - train_size
    train_ds, test_ds = random_split(dataset, [train_size, test_size])
    train_loader = DataLoader(train_ds, batch_size=cfg["batch_size"], shuffle=True)
    test_loader = DataLoader(test_ds, batch_size=cfg["batch_size"], shuffle=False)
    # After split
    torch.save(train_ds, "data/processed/train/train.pt")
    torch.save(test_ds, "data/processed/test/test.pt")
    return train_loader, test_loader


if __name__ == "__main__":
    train_loader, test_loader = get_dataloaders()
    print("MNIST loaded successfully")
    print(f"Train samples: {len(train_loader.dataset)}")
    print(f"Test samples: {len(test_loader.dataset)}")
```

II. Module 2: Model Design

Generator:

- Takes random noise as input.
- Converts noise into synthetic images.
- Uses Dense layers and ReLU activation.
- Final output uses tanh activation.

generator.py

```
import torch.nn as nn


class Generator(nn.Module):
    def __init__(self, noise_dim=100):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(noise_dim, 128 * 7 * 7),
            nn.ReLU(True),
            nn.Unflatten(1, (128, 7, 7)),
            nn.ConvTranspose2d(128, 64, 4, 2, 1),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 1, 4, 2, 1),
            nn.Tanh()
        )


    def forward(self, z):
        return self.net(z)
```

Discriminator:

- Takes real or fake images as input.
- Classifies images as real or fake.
- Uses convolutional layers followed by a sigmoid activation.

discriminator.py

```
import torch.nn as nn
```

```
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 64, 4, 2, 1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 128, 4, 2, 1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Flatten(),
            nn.Linear(128 * 7 * 7, 1),
            nn.Sigmoid()
        )


    def forward(self, x):
        return self.net(x)
```

III. Module 3: GAN Training

What happens in this module

- Generator creates fake images from noise.
- Discriminator compares real and fake images.
- Loss is calculated using Binary Cross-Entropy.
- Optimizers (Adam) update model weights.
- Training runs for multiple epochs.

Training Steps:

a) Load hyperparameters from ‘config.yaml’.
b) Prepare MNIST dataloaders.
c) Build Generator and Discriminator using ‘build_gan’.
d) Run training for each epoch:
e) Update Discriminator with real and fake data.
f) Update Generator to fool the Discriminator.
g) Log losses and save generated samples/checkpoints.

train.py

```
import yaml
import torch
import os
from data_loader import get_dataloaders
from vanilla_gan import build_gan
from utils.visualizer import save_samples
from utils.logger import init_logger, log_epoch
from torch.utils.tensorboard import SummaryWriter


device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
with open("config.yaml") as f:
    cfg = yaml.safe_load(f)
train_loader, _ = get_dataloaders()
G, D, criterion, opt_G, opt_D = build_gan(cfg, device)
writer = SummaryWriter()
init_logger()


for epoch in range(1, cfg["epochs"] + 1):
    for real, _ in train_loader:
        real = real.to(device)
        bsz = real.size(0)
        real_labels = torch.full((bsz,1), 0.9).to(device)
```

```
        fake_labels = torch.zeros(bsz,1).to(device)
        # Discriminator
        opt_D.zero_grad()
        d_real = criterion(D(real), real_labels)
        z = torch.randn(bsz, cfg["noise_dim"]).to(device)
        fake = G(z)
        d_fake = criterion(D(fake.detach()), fake_labels)
        d_loss = d_real + d_fake
        d_loss.backward()
        opt_D.step()
        # Generator
        opt_G.zero_grad()
        g_loss = criterion(D(fake), real_labels)
        g_loss.backward()
        opt_G.step()
    log_epoch(epoch, d_loss.item(), g_loss.item())
    writer.add_scalar("Loss/D", d_loss, epoch)
    writer.add_scalar("Loss/G", g_loss, epoch)
    if epoch % 10 == 0:
        save_samples(G, cfg["noise_dim"], device, epoch)
    torch.save(G.state_dict(), f"checkpoints/G_epoch_{epoch}.pt")
torch.save(G.state_dict(), "outputs/G_final.pt")
torch.save(D.state_dict(), "outputs/D_final.pt")
```

vanilla_gan.py

```
import torch.nn as nn
import torch.optim as optim
from generator import Generator
from discriminator import Discriminator


def build_gan(cfg, device):
    G = Generator(cfg["noise_dim"]).to(device)
    D = Discriminator().to(device)
    criterion = nn.BCELoss()
    opt_G = optim.Adam(G.parameters(), lr=cfg["lr"], betas=(cfg["beta1"], 0.999))
    opt_D = optim.Adam(D.parameters(), lr=cfg["lr"], betas=(cfg["beta1"], 0.999))
    return G, D, criterion, opt_G, opt_D
```

visualizer.py

Utility for saving visualizations of generated images during training.

- **save_samples(G, noise_dim, device, epoch, out_dir="samples")**

Samples images from the generator, arranges them in a grid, and saves as PNG.

```
import os
import torch
import torchvision
import matplotlib.pyplot as plt


def save_samples(G, noise_dim, device, epoch, out_dir="samples"):
    os.makedirs(out_dir, exist_ok=True)
    G.eval()
    z = torch.randn(16, noise_dim).to(device)
    with torch.no_grad():
        images = G(z).cpu()
    grid = torchvision.utils.make_grid(
        images, nrow=4, normalize=True, value_range=(-1, 1)
    )
    plt.figure(figsize=(5, 5))
    plt.imshow(grid.permute(1, 2, 0))
    plt.axis("off")
    plt.title(f"Epoch {epoch}")
    plt.savefig(f"{out_dir}/epoch_{epoch:03d}.png")
    plt.close()
```

```
    G.train()
```

logger.py

Logs GAN training losses to CSV for monitoring and reproducibility.

- **init_logger():** Creates CSV with header if not present.
- **log_epoch(epoch, d_loss, g_loss):** Appends epoch loss data to CSV.

```
import csv
import os


LOG_FILE = "logs/training_logs.csv"


def init_logger():
    os.makedirs("logs", exist_ok=True)
    if not os.path.exists(LOG_FILE):
        with open(LOG_FILE, "w", newline="") as f:
            writer = csv.writer(f)
            writer.writerow(["Epoch", "D_Loss", "G_Loss"])


def log_epoch(epoch, d_loss, g_loss):
    with open(LOG_FILE, "a", newline="") as f:
        writer = csv.writer(f)
        writer.writerow([epoch, d_loss, g_loss])
```

Training Configuration Rationale

- noise_dim = 100: Standard latent size for Vanilla GANs.
- batch_size = 64: Balances convergence stability and memory usage.
- lr = 0.0002: Recommended learning rate for GAN training.
- beta1 = 0.5: Helps stabilize Adam optimizer in GANs.
- epochs = 50: Sufficient for MNIST convergence without overfitting.

IV. Module 4: Evaluation

Evaluation Methods

- Visual inspection of generated images
- Generator and Discriminator loss tracking
- FID score computation
- Diversity score estimation
- t-SNE visualization of real vs fake samples

metrics.py

Provides metrics functions for GAN evaluation.

- **get_inception_model(device):** Loads pre-trained InceptionV3 for feature extraction.
- **extract_features(images, model, device):** Extracts features for FID/diversity.
- **calculate_fid(real_features, fake_features, eps=1e-6):** Computes Frechet Inception Distance.
- **diversity_score(fake_features):** Computes diversity of generated features.

```
import torch
```

```
import torch.nn.functional as F
import numpy as np
from torchvision.models import inception_v3, Inception_V3_Weights
from scipy.linalg import sqrtm


def get_inception_model(device):
    weights = Inception_V3_Weights.IMAGENET1K_V1
    model = inception_v3(weights=weights)
    # Disable auxiliary classifier safely
    model.aux_logits = False
    model.AuxLogits = None
    # Replace final FC with Identity → feature extractor
    model.fc = torch.nn.Identity()
    model.to(device)
    model.eval()
    return model


@torch.no_grad()
def extract_features(images, model, device):
    """ images: (N, 1, H, W) or (N, 3, H, W) """
    if images.shape[1] == 1:
        images = images.repeat(1, 3, 1, 1)
    images = F.interpolate(
        images, size=(299, 299), mode="bilinear", align_corners=False
    )
    images = images.to(device)
    features = model(images)
    return features.cpu().numpy()


def calculate_fid(real_features, fake_features, eps=1e-6):
    mu1 = real_features.mean(axis=0)
    mu2 = fake_features.mean(axis=0)
    sigma1 = np.cov(real_features, rowvar=False)
    sigma2 = np.cov(fake_features, rowvar=False)
    # 🔧 Numerical stability
    sigma1 += np.eye(sigma1.shape[0]) * eps
    sigma2 += np.eye(sigma2.shape[0]) * eps
    diff = mu1 - mu2
    covmean = sqrtm(sigma1 @ sigma2)
    if np.iscomplexobj(covmean):
        covmean = covmean.real
    fid = diff.dot(diff) + np.trace(sigma1 + sigma2 - 2 * covmean)
    return float(fid)


def diversity_score(fake_features):
    return float(np.mean(np.std(fake_features, axis=0)))
```

inference.py

Loads model, generates images, saves them as a grid.

```
import torch
import torchvision
import os
from generator import Generator


def generate_images(
    model_path="outputs/G_final.pt",
    num_images=16,
    noise_dim=100,
    output_dir="samples"
):
    os.makedirs(output_dir, exist_ok=True)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    G = Generator(noise_dim).to(device)
    G.load_state_dict(torch.load(model_path, map_location=device))
```

```
    G.eval()
    z = torch.randn(num_images, noise_dim).to(device)
    with torch.no_grad():
        images = G(z).cpu()
    grid = torchvision.utils.make_grid(
        images, nrow=4, normalize=True, value_range=(-1, 1)
    )
    out_path = os.path.join(output_dir, "inference.png")
    torchvision.utils.save_image(grid, out_path)
    print(f"[INFO] Inference image saved at {out_path}")


if __name__ == "__main__":
    generate_images()
```

evaluation.py

Evaluates the quality of generated images using metrics like FID and diversity, and visualizes with t-SNE.

```
import os
import torch
import yaml
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from generator import Generator
from utils.metrics import extract_features, calculate_fid, get_inception_model


def main():
    os.makedirs("figures", exist_ok=True)
    device = torch.device("cpu") # FORCE CPU (Mac-safe)
    with open("config.yaml") as f:
        cfg = yaml.safe_load(f)
    G = Generator(cfg["noise_dim"]).to(device)
    G.load_state_dict(torch.load("outputs/G_final.pt", map_location=device))
    G.eval()
    transform = transforms.Compose([
        transforms.Resize(cfg["image_size"]),
        transforms.ToTensor(),
        transforms.Normalize((0.5,), (0.5,))
    ])
    real_dataset = datasets.MNIST(
        root="data/raw", train=True, download=True, transform=transform
    )
    real_loader = DataLoader(
        real_dataset, batch_size=32, shuffle=True
    )
    MAX_SAMPLES = 200 # critical fix
    real_imgs = []
    fake_imgs = []
    with torch.no_grad():
        for imgs, _ in real_loader:
            imgs = imgs.to(device)
            z = torch.randn(imgs.size(0), cfg["noise_dim"]).to(device)
            fake = G(z)
            real_imgs.append(imgs.cpu())
            fake_imgs.append(fake.cpu())
            if sum(x.size(0) for x in real_imgs) >= MAX_SAMPLES:
                break
    real_imgs = torch.cat(real_imgs, dim=0)[:MAX_SAMPLES]
    fake_imgs = torch.cat(fake_imgs, dim=0)[:MAX_SAMPLES]
    inception = get_inception_model(device)
    rf = extract_features(real_imgs, inception, device)
    ff = extract_features(fake_imgs, inception, device)
    fid = calculate_fid(rf, ff)
    diversity = float(np.mean(np.std(ff, axis=0)))
```

```
    print(f"FID Score: {fid:.2f}")
    print(f"Diversity Score: {diversity:.4f}")
    tsne = TSNE(
        n_components=2, perplexity=20, random_state=42, init="random"
    )
    emb = tsne.fit_transform(np.vstack([rf, ff]))
    plt.figure(figsize=(6, 6))
    plt.scatter(emb[:len(rf), 0], emb[:len(rf), 1], s=8, label="Real")
    plt.scatter(emb[len(rf):, 0], emb[len(rf):, 1], s=8, label="Fake")
    plt.legend()
    plt.title("t-SNE: Real vs Fake")
    plt.savefig("figures/tsne_real_vs_fake.png")
    plt.close()
    print("t-SNE plot saved to figures/tsne_real_vs_fake.png")


if __name__ == "__main__":
    main()
```

V. Module 5: Deployment Layer

What happens in this module

- Saves trained generator model.
- Generates synthetic images on demand using random noise.
- Stores generated images in folders for future use.

app.py

A Streamlit-based web app for interactively generating and downloading synthetic images.

- Sidebar controls for image count and random seed.
- Main panel: generate button, image grid, and download ZIP.
- Model loading is cached for efficiency.

```
import streamlit as st
import torch
import torchvision
import numpy as np
import os
import zipfile
from generator import Generator
import cv2


def upscale_nearest(img, scale=8):
    return cv2.resize(img, None, fx=scale, fy=scale, interpolation=cv2.INTER_NEAREST)


st.set_page_config(
    page_title="Synthetic Image Generator (Vanilla GAN)",
    layout="wide"
)


st.sidebar.title("⚙️ Controls")
num_images = st.sidebar.slider("Number of images", 1, 64, 16)
seed = st.sidebar.number_input("Random Seed (optional)", value=0, step=1)
generate_btn = st.sidebar.button("Generate Images")


st.title("Synthetic Image Generator (Vanilla GAN)")
st.markdown(
    """
    This application generates **privacy-preserving synthetic images** using a **Vanilla Generative
Adversarial Network (GAN)**.
```

```
    **Use cases**
    - Data augmentation
    - Model robustness testing
    - Privacy-safe dataset sharing
    - Educational demonstrations
    """
)


@st.cache_resource
def load_model():
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = Generator(100).to(device)
    model.load_state_dict(torch.load("outputs/G_final.pt", map_location=device))
    model.eval()
    return model, device


if generate_btn:
    st.subheader("Generated Images")
    with st.spinner("Generating synthetic images..."):
        if seed != 0:
            torch.manual_seed(seed)
        G, device = load_model()
        z = torch.randn(num_images, 100).to(device)
        with torch.no_grad():
            images = G(z).cpu()
        os.makedirs("samples/generated", exist_ok=True)
        for i, img in enumerate(images):
            torchvision.utils.save_image(
                img, f"samples/generated/img_{i}.png", normalize=True, value_range=(-1, 1)
            )
        cols = st.columns(4)
        for i, img in enumerate(images):
            img_np = img.squeeze().numpy()
            img_np = upscale_nearest(img_np)
            cols[i % 4].image(img_np, clamp=True, caption=f"Img {i}", width=100)
        grid = torchvision.utils.make_grid(
            images, nrow=4, normalize=True, value_range=(-1, 1)
        )
        grid_np = grid.permute(1, 2, 0).numpy()
        st.image(grid_np, caption="Synthetic Images (28×28)", width="content")
        st.success("✅ Images generated successfully!")
        zip_path = "samples/generated_images.zip"
        with zipfile.ZipFile(zip_path, "w") as zipf:
            for i in range(num_images):
                zipf.write(f"samples/generated/img_{i}.png")
        with open(zip_path, "rb") as f:
            st.download_button(
                label="⬇️ Download Images as ZIP",
                data=f,
                file_name="synthetic_images.zip",
                mime="application/zip"
            )
```

API Endpoints

This FastAPI service exposes GAN image generation endpoints for programmatic access.

- Root Status Endpoint (GET `/`)
- Checks if the API is running.
- Generate Images (POST `/generate`)
- Generates a batch of synthetic images using the trained GAN.

api.py

```
from fastapi import FastAPI
import torch
from src.generator import Generator


app = FastAPI(title="Synthetic Image Generator API")
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
G = Generator(100).to(device)
G.load_state_dict(torch.load("outputs/G_final.pt", map_location=device))
G.eval()


@app.get("/")
def root():
    return {"status": "API is running"}


@app.post("/generate")
def generate_images(num_images: int = 16):
    z = torch.randn(num_images, 100).to(device)
    with torch.no_grad():
        images = G(z).cpu().tolist()
    return {"images": images}
```

Dockerfile

- Provides a containerized runtime for the FastAPI service.
- Installs dependencies.
- Exposes port 8000.
- Launches the API server using Uvicorn.

```
FROM python:3.10-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["uvicorn", "src.api:app", "--host", "0.0.0.0", "--port", "8000"]
```

VI. Module 6: Monitoring & Update Pipeline

What happens in this module

- Logs inference latency
- Supports model version tracking
- Enables post-deployment monitoring
- Update the model versions.

monitoring.py

This module offers logging for inference latency to CSV files. It ensures reproducibility and transparency in the model's inference performance.

- **log_inference(duration_ms):** Logs the current timestamp and inference duration (ms) into ‘logs.inference_logs.csv’. Automatically creates the logs directory if it doesn't exist.

```
import time
import csv
import os
```

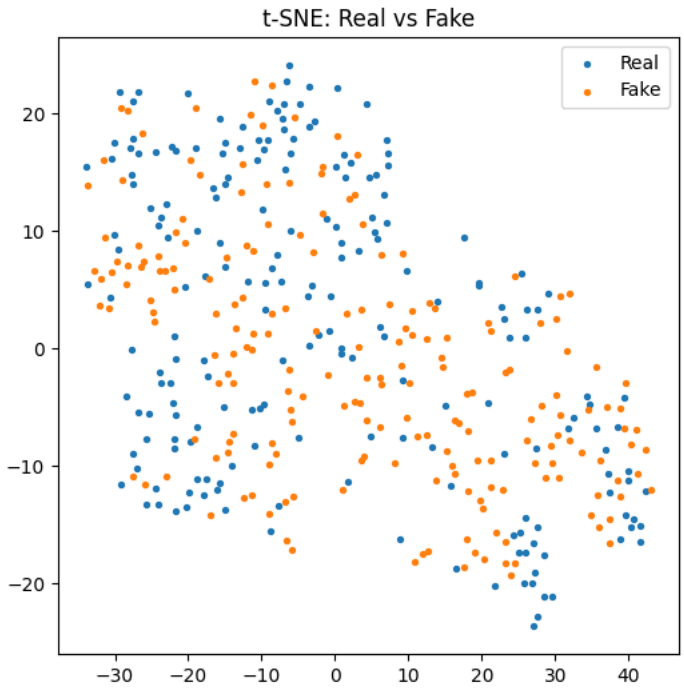
```
LOG_FILE = "logs/inference_logs.csv"


def log_inference(duration_ms):
    os.makedirs("logs", exist_ok=True)
    file_exists = os.path.isfile(LOG_FILE)
    with open(LOG_FILE, "a", newline="") as f:
        writer = csv.writer(f)
        if not file_exists:
            writer.writerow(["timestamp", "latency_ms"])
        writer.writerow([time.time(), duration_ms])
```

Outputs

Generated synthetic digit images stored in ‘samples/’

Generated Image



T-SNE Real vs Fake Plot

Reproducibility

- Fixed random seeds can be used for deterministic generation.
- Configuration is centralized in config.yaml.
- Logs, checkpoints, and generated samples are saved persistently.
- Evaluation metrics can be recomputed using saved models.

Error Handling & Stability

- CPU-only mode enforced during evaluation for macOS stability.
- Numerical stability ensured in FID computation using epsilon.
- Inception auxiliary classifier safely disabled.
- Dataset download checks prevent redundant downloads.

Limitations

- Image quality is limited compared to DCGAN or diffusion models.
- No conditional generation (digits are not class-controlled).
- Evaluation metrics are indicative, not benchmark-level.

Conclusion

The project successfully implements a Vanilla GAN for generating synthetic handwritten digit images. The system includes data preprocessing, adversarial training, evaluation using standard metrics, deployment interfaces, and monitoring mechanisms, providing a complete end-to-end generative learning pipeline.