GAN-Based Procedural Asset Generation: A Study from MNIST to CelebA

George Kotti (ghk24)
CSE 4693
Intro to Machine Learning

Introduction

Goal: Explore how GANs can generate assets for game development

Understand how GANs work internally

Apply GANs to real-world texture generation

Simple Vanilla GAN first → practical DCGAN demo

Gather findings, explain results, and show correlation

Project Overview

Two complementary phases:

Phase 1-1: Numpy GAN

Phase 1-2: Optimized Numpy GAN (Colab)

Phase 2: DCGAN (CelebA, Colab)

Learn theory → Apply to modern datasets and prove use in asset generation

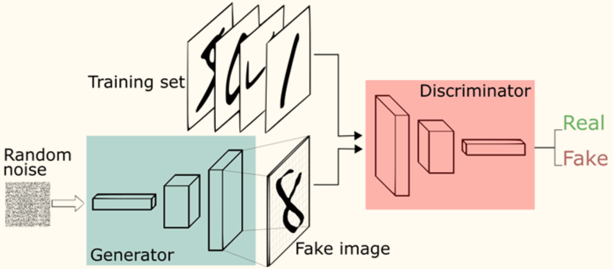
GAN Basics

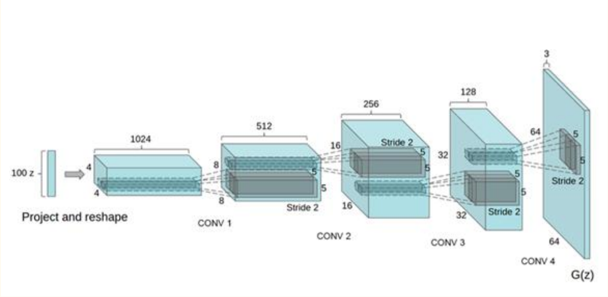
Generative Adversarial Networks (Gans)

- GAN = Generator (G) + Discriminator (D): Minimax
- Generator: Produces fake images from random noise
- Discriminator: Judges real vs fake
- Adversarial training improves both– ZeroSum Game
- BCE loss function (WGANs use Wasserstein loss)
- Data-driven, can adapt to different visual styles if the dataset is diverse.

Deep Convolutional GAN (DCGAN)

- Adapted to help stabilize GAN training on image data
- Uses convolutional layers instead of Fully Connected
- Learns with Filters





$$\min_G \max_D \; \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[\log(1 - D(G(\mathbf{z})))].$$

GAN Challenges

Mode Collapse:

- Generator learns to produce only a few types of outputs
- Reduces diversity and leads to repetitive samples

Training Instability:

- Discriminator and Generator must stay in balance
- If Discriminator becomes too strong, Generator gradients vanish (no learning)
- If Generator wins too easily, Discriminator can't learn meaningful features

Sensitive to Hyperparameters:

- Learning rates must be tuned carefully
- Batch size affects convergence behavior
- Activation function choice (ReLU, LeakyReLU, Tanh) impacts training speed and stability

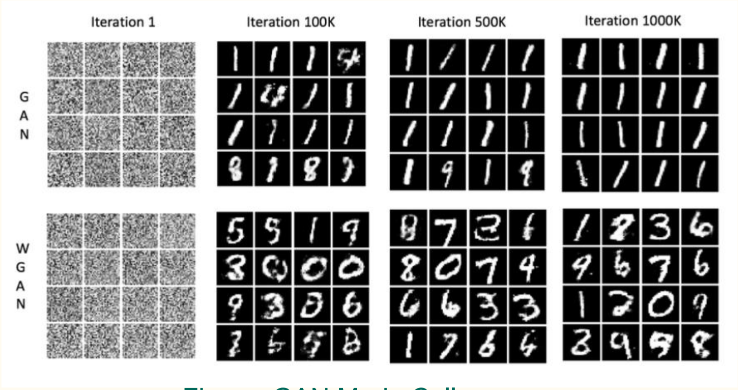


Figure: GAN Mode Collapse

Memory Management and Optimization (NumPy Implementation):

- Without frameworks, arrays were manually managed
- Needed ulimit restrictions to prevent system freezes
- GPU programming greatly optimizes performance

Vanilla GAN

Built completely from scratch using NumPy

Handwritten forward, backward, and update (train_step) functions

No Keras, PyTorch, or auto-grad

Focus:

- Understand how GANs really learn
- Build upon the foundations for project specific focus
- Test and optimize the GAN architecture

Vanilla GAN Architecture

Forward → Backward + Update (train_step)

Generator:

Noise (100D) → Dense(256) → ReLU → Dense(128) → ReLU → Dense(784) → Tanh

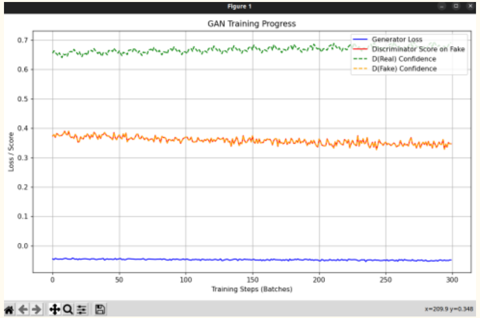
Discriminator:

Image (784D) → Dense(256) → Leaky ReLU → Dense(128) → Leaky ReLU → Dense(1) → Sigmoid

Dense: h1=x × W1 + b1 - linear transformation

Leaky ReLU: f(x)={x x if > 0, αx if x <= 0 - activation function

Noise: randn(latent_dim) input



Small fully connected networks

Good for simple data like MNIST digits

Vanilla GAN Overview

Tested Training settings:

- 1,000-70,000 MNIST images used
- Batch size: 1-256 (avg. 64)
- Learning rates: 0.0001-0.002 (avg. 0.0002)
- 10–20 epochs per training cycle

Output quality:

- Early epochs: random noise
- Later epochs: emergence of basic digit shapes (1s, 3s, 5s, 7s)

Performance limitations:

- Slow convergence (no convolutional structure)
- Unstable training after 30+ training sessions
- Memory limited training setting parameters

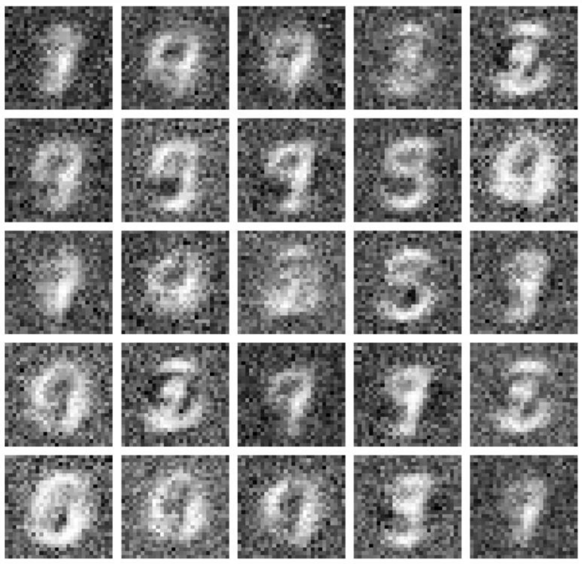


Figure: Train Size 1000

Vanilla GAN Generator Image Examples

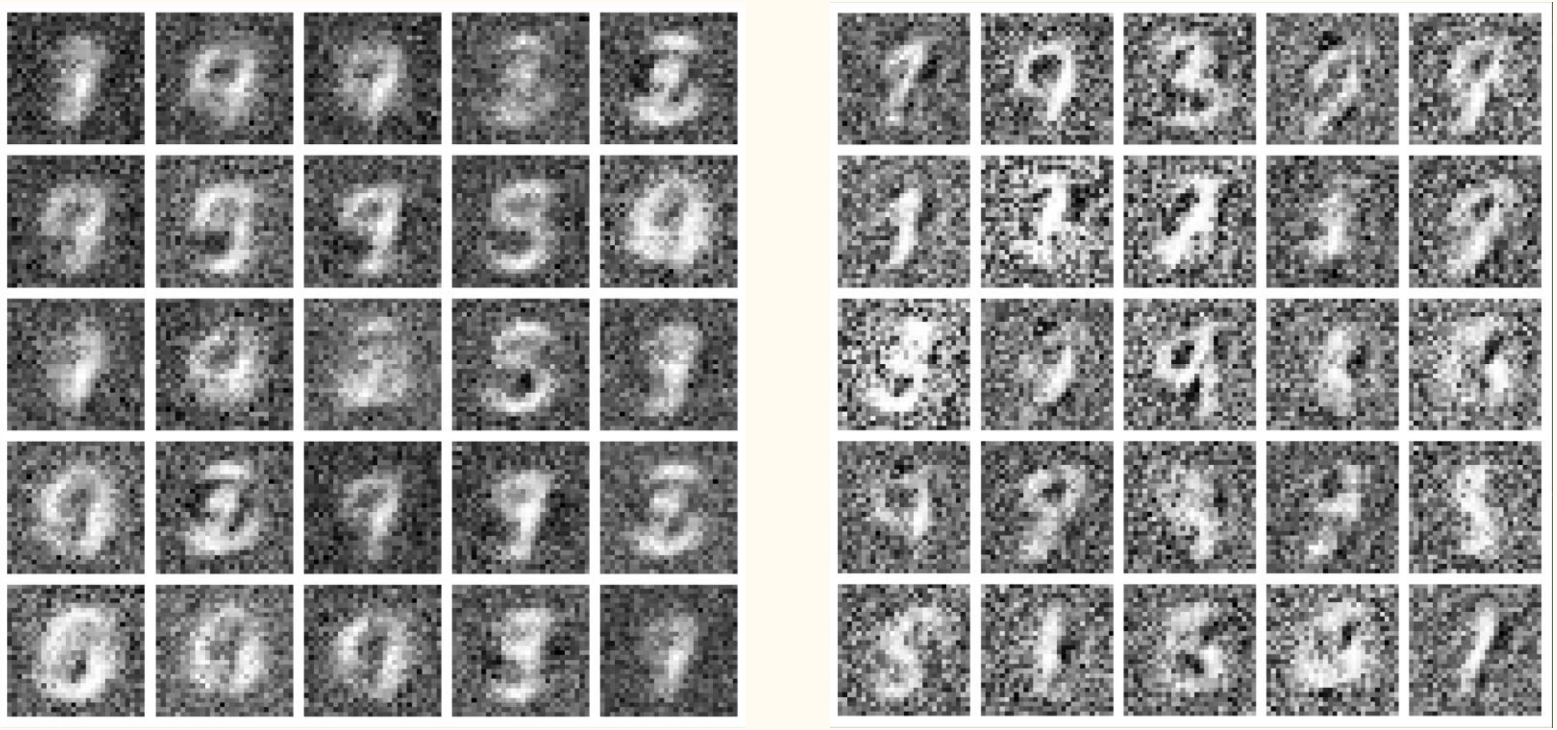


Figure: Midway training images

Vanilla GAN Loss Diagram Examples

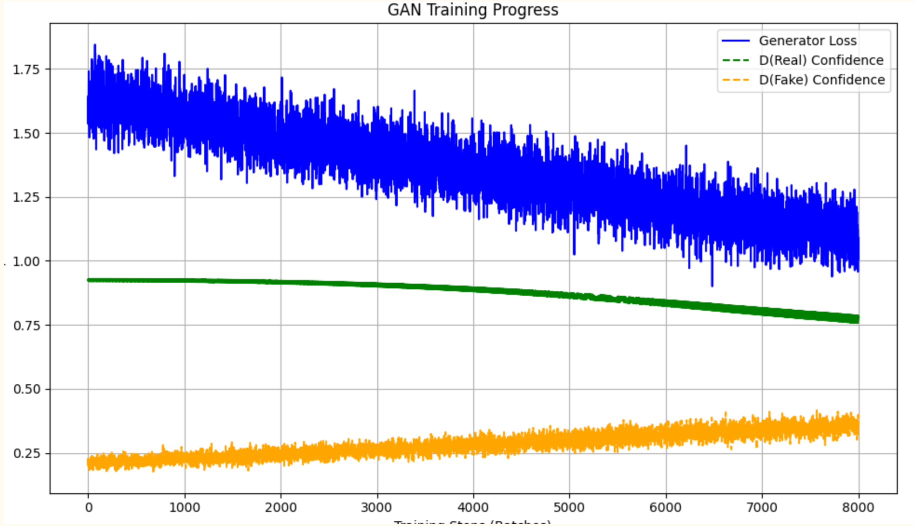


Figure: Improving GAN

Figure: Generator Failing Overtime

Vanilla GAN Loss Diagram Examples

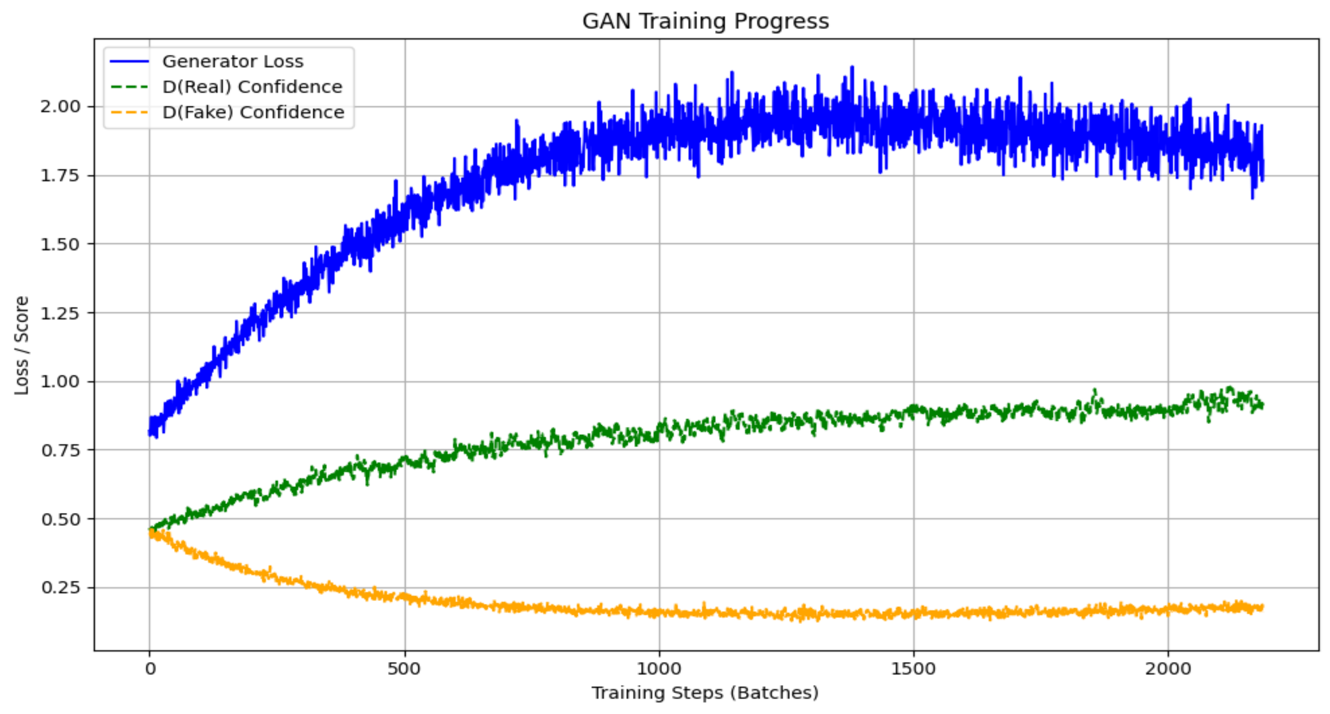


Figure: Initial Training Start

DCGAN Demo Architecture (CelebA)

Generator:

Noise (100D) → ConvTranspose(512) → BatchNorm → ReLU → ConvTranspose(256) → ... → Output (64x64x3)

Discriminator:

Input (64x64x3) → Conv(64) → LeakyReLU → Conv(128) → ... → Dense(1) → Sigmoid

Fully convolutional

Generates high-quality 64x64 RGB images

Challenges

General:

Finding quality datasets

Performance and memory

NumPy:

Harder to stabilize training

Manual gradient debugging was very time-consuming

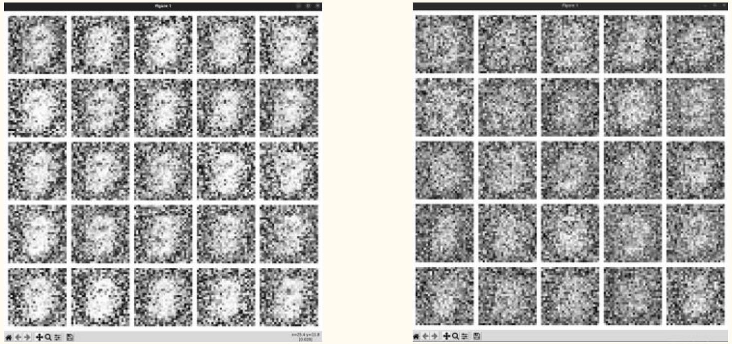


Figure: Mode Collapse

DCGAN:

Much harder to implement ‘manually’

Almost exclusively relies on external ML libraries (PyTorch, Keras, TensorFlow)

Optimization Techniques for GAN Training

Hardware Acceleration (GPUs):

- GANs involve heavy matrix operations (dot products, convolutions)
- GPUs dramatically reduce training time by parallelizing operations
- Examples:
  - Google Colab free GPU (Tesla T4, P100)
  - CUDA-enabled PyTorch/TensorFlow models
  - Training MNIST GAN: ~5–10x faster on GPU vs CPU

Advanced Optimizers:

- **Adam Optimizer** (adaptive learning rates) improves convergence
- **RMSProp**: Used sometimes in WGANs for stable critic updates
- Manual SGD (like in NumPy project) often too slow for deep GANs

Optimization Techniques for GAN Training

Model Architecture Tricks:

- **Batch Normalization:**
    - Reduces internal covariate shift
    - Leads to faster and more stable training
    - Used heavily in DCGANs
- **Leaky ReLU:**
    - Allows minor negative activations
    - Prevents "dying ReLU" problem during training

Alternative Models and Combinations:

- **Autoencoders + GANs (VAE-GANs):**
    - Combine latent space encoding with generative power
    - Improves sample diversity and feature structure
- **Wasserstein GAN (WGAN):**
    - Uses Wasserstein distance instead of BCE loss
    - Provides smoother, more stable training
    - Mitigates mode collapse

Findings

GAN training is inherently unstable without careful architecture design and loss balancing

- Even small errors in gradient computation can cause full collapse

Simpler models (Fully Connected layers) are capable of rough generation but plateau quickly

Adding convolutional layers (DCGAN) massively improves:

- Training speed
- Image quality
- Stability over long epochs

Memory management is critical when working without automatic deep learning frameworks

Procedural generation using GANs is achievable even with small datasets

- MNIST GAN could serve as a texture generator base for simple game assets (Limited variety with 1000 train_size, but results were there.
- DCGAN (CelebA) showed the potential for much more detailed content

Current Related Applications in Game Development

- **Existing 2D Tile Resources**
  - **Sprite Editing Tools** (Aseprite, Tiled) **no built-in ML** generation – powerful, but **manual**

- **Traditional Procedural Methods**
  - **Noise-Based Approaches** (Perlin, Simplex) are popular for **terrain** or background **patterns**, but often lack detail

- **ML / AI Art Tools**
  - **General AI Generators** (Stable Diffusion, DALL*E) can produce 2D images, but not specialized for tile generation or consistent tile sets

Applications to Game Development

- **2D Tiles and Texture Creation**:
  - **Simple GANs** (like our NumPy GAN) can generate rough, randomized tile patterns
  - **DCGANs** (PyTorch CelebA model) show clear potential for creating **structured, detailed textures**
  - Generated tiles could be adapted for:
    - **Terrain tiles** (grass, stone, sand)
    - **Background patterns** (walls, floors)
    - **Environment sprites** (clouds, trees, decorations)
- **Advantages Over Traditional Methods**:
  - **Variation**: Automatically creates many stylistic versions of the same asset
  - **Consistency**: DCGANs can maintain a visual "style" across generated outputs
  - **Efficiency**: Reduces manual effort needed to handcraft large tile sets or texture libraries
- **Potential Game Development Uses**:
  - Dynamic level generation (procedural maps with new textures each run)
  - Texture blending (smooth transitions between biomes)
  - Personalized player environments (each player’s world looks unique)
- **Key Finding**:

GAN-based procedural content generation can offer **fast, scalable, and stylistically coherent** asset creation for games, especially in indie and procedural-heavy projects.

Thanks!

References

Welcome to PyTorch Tutorials — PyTorch Tutorials 2.5.0+cu124 documentation. (2023). Pytorch.org. https://pytorch.org/tutorials

Rafatirad, S., Homayoun, H., Chen, M. Z. Q., & Dinakarrao, S. M. P. (2023). *Machine learning for computer scientists and data analysts: From an applied perspective*. Springer.