Procedural Asset Generation for Game Development Using GANs: From NumPy Foundations to Practical DCGAN Applications

George Kotti IV
Department of Computer Science and Engineering
Mississippi State University
Starkville, United States
gkotti4@gmail.com

Abstract—**This project investigates Generative Adversarial Networks (GANs) as a procedural content generation tool in game development, focusing on both foundational understanding and practical application. In Phase 1, we implemented a vanilla GAN entirely in NumPy—manually coding forward and backward passes, applying stochastic gradient descent, and managing memory constraints—to generate 28×28 grayscale textures from MNIST. Phase 2 scaled this approach to a Deep Convolutional GAN (DCGAN) in PyTorch, employing transpose-convolution layers, Batch Normalization, and Adam optimization to synthesize high-resolution (64×64) RGB images from the CelebA dataset. Key challenges—mode collapse, training instability, dataset preprocessing, and hyperparameter sensitivity—were overcome through hyperparameter tuning, gradient clipping, and regular checkpointing. Our findings show that even small datasets yield viable low-fidelity tiles, while DCGANs produce detailed, stable textures suitable for terrain, foliage, and architectural assets. We demonstrate how these models can integrate into procedural pipelines—with potential for blending biome transitions and supporting runtime generation. Future work includes higher-resolution architectures, conditional GAN variants for style control, and real-time engine integration to automate diverse, artist-guided asset creation in modern game engines.**

Keywords—Generative Adversarial Network (GAN), Deep Convolutional GAN (DCGAN), Procedural Content Generation, Vanilla GAN, Mode Collapse, Batch Normalization, Stochastic Gradient Descent (SGD), Rectified Linear Unit (ReLU), Leaky ReLU, Tanh, PyTorch, NumPy, Autoencoder, Latent Space, Latent Dimensions

I. Introduction and Importance

Procedural texture and asset generation in the game development pipeline has long relied on noise-based algorithms such as Perlin noise and fractal techniques. While these methods excel at creating organic, tileable patterns, they tend to lack fine detail, stylistic consistency, and adaptive control—forcing artists to manually tweak parameters or hand-paint variations.

Generative Adversarial Networks (GANs) present a powerful alternative. By learning directly from example datasets, GANs can produce richly detailed, high-fidelity textures that retain the statistical properties of real-world materials or custom artwork. They enable:

- **Enhanced realism:** Capturing subtle variations in color, lighting, and microstructure far beyond simple noise.
- **Style control:** Steering outputs toward specific themes or palettes via conditional inputs or latent-space manipulation.
- **Automated variation:** Generating hundreds of unique assets from a small set of examples, reducing the repetitive workload on artists.
- **Scalability:** Scaling to higher resolutions or new domains by retraining on different datasets.

This project aimed to:

- **Understand adversarial theory:** Delve into the minimax game between generator and discriminator, and study how loss functions and network architectures drive convergence.
- **Implement a NumPy-based Vanilla GAN:** Manually code forward/backward passes to solidify low-level insights into gradient flow, numerical stability, and mode collapse. See appendix E.
- **Scale to DCGAN with PyTorch:** Leverage convolutional transpose layers and BatchNorm to generate 64×64 RGB outputs from the CelebA dataset, demonstrating real-world applicability. See appendix E.
- **Evaluate for 2D game assets:** Assess generated samples for use as tile sets, terrain textures, and sprite backgrounds in typical game engines.
- **Investigate real-time workflows:** Explore inference-time optimizations and pipeline integration for engines like Unreal, including GPU acceleration and lightweight model variants.
- **Benchmark performance:** Profile training speed, memory usage, and stability under varying

hyperparameters to identify practical guidelines for in-game procedural generation.

II. Technical Overview of GANs

Generative Adversarial Networks (GANs), introduced by Ian Goodfellow et al. in 2014, are a class of generative models that use deep learning to produce new data samples that closely resemble a target dataset. Rather than predicting labels or outcomes, GANs aim to learn the underlying distribution of a dataset and then generate new data points from that learned distribution. This capability makes GANs especially useful for tasks such as image synthesis, data augmentation, texture generation, and style transfer.
At the core of a GAN are two neural networks playing an adversarial game:

- The **Generator (G)** learns to transform random noise into synthetic data that resembles the real training data. It starts with no knowledge of the data and progressively improves by learning from feedback.

- The **Discriminator (D)** acts as a binary classifier that distinguishes between real samples (from the dataset) and fake samples (from the generator).

These two networks are trained simultaneously in a minimax (zero-sum) *game*, where the Generator tries to fool the Discriminator, and the Discriminator tries to correctly identify real versus fake inputs. The training objective is defined by the following equation:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))]$$

Where:

- $x \sim pdata$ is a real data sample from the dataset,

- $z \sim pz$ is a latent noise vector sampled from a known distribution (usually Gaussian),

- G(z) is the generator's synthetic output.

Through this adversarial process, the generator learns to map from a latent space (noise) to a high-dimensional data space, often expressed as:

$$G(z) \rightarrow x$$

This formulation indicates that the generator is learning a transformation from a simple noise distribution to a complex data distribution, such as the distribution of images, textures, or other structured content. [1,2,3]

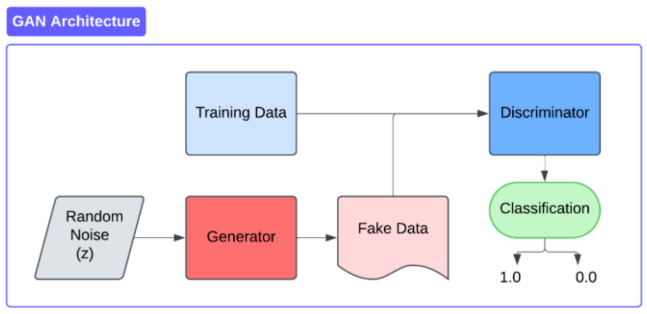

Fig. 1. GAN Flowchart

Advantages of GANs:

- **High-quality generation:** GANs can produce highly realistic and detailed synthetic data that often surpasses older generative techniques in visual fidelity.

- **Versatility:** They are highly adaptable and can be applied across different data modalities, including images, audio, video, and even text. Their core architecture is domain-agnostic, making GANs usable in a wide range of fields from art to medicine.

- **Data augmentation:** GANs can generate large quantities of diverse and label-free data. This makes them useful for augmenting datasets, especially in scenarios with limited real samples (e.g., rare diseases or synthetic game textures), to improve the performance of other machine learning models.

Challenges of GANs:

- **Unstable training:** Training GANs is notoriously difficult. Common issues include mode collapse (where the generator produces repetitive outputs) and vanishing gradients (where the generator receives no learning signal). Proper hyperparameter tuning, architectural design, and regularization strategies are essential to achieve balance.

- **Control over outputs:** While GANs can generate impressive results, it is often difficult to direct the output toward a specific style or variation. Developers typically have limited control over the features or structure of the generated content without additional techniques like conditional GANs (cGANs) or latent space manipulation.

- **Reproducibility:** Due to the stochastic nature of adversarial training and sensitivity to initialization and hyperparameters, it is difficult to reproduce GAN outputs exactly across different training runs. Even small differences in setup can lead to significantly different results.

- **Computational and memory cost:** Because GANs often require large neural networks and datasets, especially for high-resolution outputs, they demand considerable CPU/GPU memory and compute power. Training time can also be lengthy, especially without access to hardware acceleration.

A. Supervised vs. Unsupervised: Where GANs Fit

In contrast to ***supervised learning***, where each input x is paired with a known label y and the model learns a direct mapping $f(x) \rightarrow y$, GANs operate in an unsupervised fashion. The training data for GANs does not need to be labeled. Instead, the generator learns from the implicit feedback of the discriminator, which itself is learning what makes data look "real."

This lack of reliance on labeled data is a major strength of GANs, especially in domains where labeled datasets are expensive or infeasible to acquire — such as stylized game textures, creative assets, or niche image datasets.

Despite the lack of labels, GANs are able to capture the structure and complexity of the real data distribution. This includes modeling:

- Non-linear relationships
- Multimodal distributions (e.g., generating multiple categories or styles from one noise source)
- High-dimensional representations (e.g., generating 64×64 or 256×256 images from 100D latent vectors)

B. How GANs Differ from Other Models

Many traditional models, such as logistic regression or support vector machines (SVMs), rely on assumptions like linearly separable classes or require explicitly labeled training data, as previously mentioned. Even modern supervised neural networks depend on clear input-output mappings. In contrast:

- GANs do not require linearity or separability in the data.
- GANs do not require labeled data.
- GANs do not aim to classify — they aim to synthesize.

Because of this, GANs are uniquely positioned in the field of machine learning and generative modeling. They do not simply memorize or replicate examples; they learn to approximate the true data distribution and sample from it — a much harder and more powerful goal.

C. Deep Convolutional GANs (DCGANs)

Deep Convolutional GANs (DCGANs), introduced by Radford et al. in 2015, replace the fully connected layers of vanilla GANs with convolutional and transposed-convolutional layers, bringing substantial gains in stability and output quality for image generation. Convolutional layers excel at capturing spatial hierarchies—such as edges, textures, and patterns—by sharing weights across the image and preserving translation invariance. They also dramatically reduce parameter count compared to dense layers, making it feasible to train deeper networks without overfitting.

In a DCGAN, the Generator employs a series of ConvTranspose2D (fractional-strided convolution) layers to progressively upsample a latent noise vector into a coherent image. Each upsampling step is typically followed by Batch Normalization and a ReLU activation (with a final Tanh to bound outputs). Conversely, the Discriminator uses Conv2D layers to downsample input images, interleaving LeakyReLU activations and Batch Normalization to maintain gradient flow even when activations are negative. This fully convolutional design avoids pooling operations—favoring strided convolutions in the Discriminator and strided transposed convolutions in the Generator—avoiding any hidden dense layers. [3]

The DCGAN paper codified a set of architectural guidelines that are now standard practice:

- Replace pooling layers with strided convolutions in the Discriminator and fractional-strided convolutions in the Generator, ensuring learned up/downsampling.
- Apply Batch Normalization after every convolutional layer (except the output), which smooths the optimization landscape.
- Use ReLU activations in the Generator's hidden layers, with Tanh at the output to map pixels into the −1,1 range.
- Use LeakyReLU activations in the Discriminator, preventing "dead" neurons when inputs are negative.
- Remove all fully connected hidden layers, keeping the networks fully convolutional for better spatial learning.

These design choices led to faster convergence, more stable adversarial training, and the ability to generate higher-resolution images—commonly 64×64 or 128×128 RGB—without collapse.

For procedural texture and tile generation in games, DCGANs offer several advantages over vanilla GANs. First, the convolutional Generator synthesizes sharper, more detailed patterns that resemble real materials. Second, shared filters and BatchNorm yield greater style consistency across generated assets. Third, the convolutional architecture is inherently more stable during training, reducing the frequency of collapse events. Finally, DCGANs are readily implemented in GPU-accelerated frameworks like PyTorch or TensorFlow, enabling rapid experimentation, scaling to larger datasets, and integration into real-time pipelines.

In this project, we trained a PyTorch DCGAN on the CelebA face dataset to validate these benefits. Although CelebA consists of human faces, the same convolutional principles apply equally to any structured image domain—whether 2D game tiles, texture atlases, or environment sprites—making DCGANs a powerful tool for automated game asset creation. [1,3, 69]

D. Extensions and Adaptations of the GAN Framework

Since the original GAN and DCGAN architectures, researchers have introduced several extensions and adaptations to enhance stability, control, and applicability:

VAE-GANs
Researchers have combined autoencoding with adversarial training in the VAE–GAN. Here, a Variational Autoencoder enforces a smooth latent space by minimizing

$$L_{\mathrm{VAE}} = \mathbb{E}_{q(z|x)}\left[\|x-\tilde{x}\|^2\right] + \mathrm{KL}\big(q(z \mid x) \parallel p(z)\big),$$

while a GAN discriminator adds

$$-\,\mathbb{E}_{z\sim q(z|x)}\left[\log D(\tilde{x})\right]$$

to sharpen realism and reduce mode collapse. [2, 32, 57]

Conditional GANs

A Conditional GAN (cGAN) incorporates an extra label or attribute vector y, so the generator becomes G(z,y) and the discriminator evaluates D(x,y). This allows a single model to produce distinct asset types (e.g. grass vs. stone textures) by simply altering y, without retraining separate networks.

Progressive Growing

For high-resolution synthesis, progressive growing trains at low resolution first (e.g. 4×4) and then incrementally adds layers to reach 256×256. Early stages learn global structure; later stages refine fine detail, dramatically improving stability.

Multi-Scale Discriminators

Some implementations include multi-scale discriminators, each examining different image scales or patch sizes. By enforcing realism both globally and locally, these critics ensure coherent large-scale layout and crisp textures alike.

Real-world images often vary in aspect ratio. Common preprocessing strategies include:

- Uniform resize to a square canvas (may distort).
- Resize + center-crop to preserve central content.
- Resize + padding to maintain aspect ratio with blank borders.

Spectral Normalization

Finally, spectral normalization regularizes each layer by dividing weights by their largest singular value, enforcing a 1-Lipschitz constraint. This curbs discriminator overconfidence and stabilizes gradients without heavy computation.

Together, these adaptations—hybrid VAE–GANs, conditional inputs, progressive growth, multi-scale critics, flexible preprocessing, and spectral normalization—provide a versatile toolkit for stable, high-quality texture and tile generation in game asset pipelines. [32,39]

III. Methodology

A. Phase 1: Vanilla GAN (NumPy-Based)

In the first phase of the project, we built a "vanilla" GAN entirely from scratch using NumPy to gain familiarity with the mechanics of adversarial training. The generator takes as input a 100-dimensional noise vector z~N(0,1) and maps it through three fully connected layers—128→256→784—with ReLU activations in the hidden layers and a Tanh activation at the output. The final 784-dimensional vector is reshaped into a 28×28 grayscale image. The discriminator mirrors this structure in reverse: it flattens a 28×28 input image into a 784-vector, then passes it through two LeakyReLU-activated dense layers (784→256→128) before arriving at a single sigmoid-activated output neuron representing "real" or "fake."

Training proceeds in mini-batches. For each batch, we sample real MNIST images (normalized to [−1,1]) and an equal number of noise vectors. We compute the discriminator loss

$$L_D = -\left[\mathbb{E}_{x\sim p_{\mathrm{data}}} \log D(x) \;+\; \mathbb{E}_{z\sim p_z} \log\big(1 - D(G(z))\big)\right]$$

And the generator loss

$$L_G = -\mathbb{E}_{z\sim p_z} \log D\big(G(z)\big).$$

Gradients are calculated by hand using chain-rule expansions and clipped to prevent NaNs. We update both networks via simple SGD with a fixed learning rate of *0.0002*. To avoid running out of memory, we enforced process limits using ulimit and kept batch sizes small (typically 64). This manual implementation exposed subtleties of gradient flow, numerical stability, and mode collapse, which provided the foundation for more sophisticated models.

B. Phase 2: DCGAN (PyTorch-Based)

Building on those insights, Phase 2 employed a standard DCGAN architecture in PyTorch following the readily available PyTorch DCGAN documentation, to generate higher-quality, full-color images. The generator begins with a 100-dimensional noise vector and upsamples it through a sequence of four ConvTranspose2D layers (kernel size 4, stride 2), each followed by BatchNorm and ReLU, ending with a Tanh activation to produce a 64×64 RGB image. The discriminator uses the inverse structure—Conv2D layers (stride 2, kernel 4) with LeakyReLU and BatchNorm—to downsample a 64×64 image into a single probability via a final sigmoid. We initialized weights from a Normal(0, 0.02) distribution and trained both networks with the Adam optimizer ($\alpha = 0.0002, \beta_1 = 0.5$).

Data preprocessing involved resizing and center-cropping CelebA face images to 64×64, converting to tensors, and normalizing to [−1,1]. During each epoch (up to 50), the discriminator is first updated on a mix of real and generated images, then the generator is updated to maximize the discriminator's output on its fakes. We logged losses, saved model checkpoints, and visualized samples in Google Colab using Matplotlib. Training on a Tesla T4 GPU reduced epoch time from several minutes (CPU) to under one minute (GPU), enabling rapid iteration and hyperparameter tuning with the main bottleneck being network speed and a steady connection. [2]

Key implementation details:

- **Manual vs. framework**: Phase 1's NumPy backprop contrasts sharply with Phase 2's automatic differentiation in PyTorch.
- **Stability measures**: In both phases, we applied gradient clipping, careful weight initialization, and monitoring of losses to prevent collapse.

- **Resource management**: Phase 1 relied on OS limits (ulimit), whereas Phase 2 harnessed GPU memory and checkpointing.

This two-stage methodology—starting with low-level manual implementation then scaling up to a convolutional, GPU-accelerated model—ensured a deep understanding of GAN fundamentals and practical expertise in applying them to real-world texture generation tasks.

1) Core Neural Operations and Parameters

In both vanilla and convolutional GAN architectures, a handful of building-block operations repeatedly appear. Understanding their roles is crucial for grasping how these networks learn and process data.

Batch Normalization (BatchNorm)

BatchNorm normalizes each layer's activations over a mini-batch. Given pre-activation outputs μ_i for $i = 1, \ldots, m$, it computes the batch mean μ and variance σ^2, then rescales and shifts via learnable parameters γ and β:

$$\hat{u}_i = \frac{u_i - \mu}{\sqrt{\sigma^2 + \epsilon}}, \quad y_i = \gamma\,\hat{u}_i + \beta.$$

By keeping layer inputs on a consistent scale, BatchNorm accelerates convergence, enables higher learning rates, and reduces sensitivity to initialization.

Rectified Linear Unit (ReLU)

ReLU is a simple, non-saturating activation:

$$\mathrm{ReLU}(x) = \max(0, x).$$

It zeros out negative inputs and passes positive inputs unchanged. This yields sparse activations (improving generalization) and avoids the vanishing-gradient issues common in sigmoid or tanh units.

Leaky ReLU

A variant of ReLU, Leaky ReLU allows a small, non-zero gradient when $x < 0$:

$$\mathrm{LeakyReLU}(x) = \begin{cases} x, & x \geq 0, \\ \alpha\,x, & x < 0, \end{cases}$$

with a typical slope $\alpha = 0.2$. This prevents "dead" neurons that never activate and helps maintain information flow in the discriminator's negative-activation scheme.

Hyperbolic Tangent (Tanh)

Tanh maps real inputs into (-1, 1):

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

Because it is zero-centered, Tanh is often used in the generator's final layer to produce output pixels normalized to [-1, 1], matching preprocessing of real images and aiding stable adversarial training.

Convolutional Strides and Kernels

- **Kernel (Filter) Size:** A $k \times k$ window of learnable weights that slides over the input. Each position computes a weighted sum of k^2 inputs, capturing local spatial patterns (edges, textures).

- **Stride (s)** controls the step size of the kernel's movement. A stride of 1 examines every overlapping patch; a stride of 2 downsamples by moving two pixels at a time, halving spatial dimensions. In transpose-convolutions, stride 2 upsamples by roughly doubling dimensions.

These operations—normalization, non-linear activations, and carefully chosen convolutional parameters—form the backbone of both vanilla and deep convolutional GANs, enabling them to learn complex, high-dimensional data distributions.

IV. Challenges and Solutions

In our pure-NumPy implementation, mode collapse manifested early: after a few epochs the generator converged on producing only one digit shape (for example, repeatedly outputting "1"), rather than the full variety of MNIST classes. Because the discriminator—implemented as two small fully connected layers—quickly learned to reject anything but those few patterns, the generator found it easier to keep "fooling" the discriminator with repeated outputs than to explore new digit forms.

We diagnosed mode collapse by monitoring the diversity of generated samples: when successive batches looked nearly identical, it signaled the collapse. To mitigate this, we experimented with:

- **Reducing the learning rate** (from 0.0002 to 0.0001) to slow the discriminator's learning, giving the generator more room to explore.

- **Training the generator more frequently** (two generator updates per discriminator update) so that G could catch up rather than falling behind.

- **Adding small Gaussian noise** (σ≈0.05) to both real and fake inputs before feeding them to D, which softened discrimination and encouraged G to diversify its outputs.

These adjustments restored diversity—generator outputs began cycling through multiple digit shapes again—and provided firsthand insight into how delicate the adversarial balance is when implemented entirely by hand.

V. Findings

Our experiments yielded several key insights into how architectural choices and data scale affect GAN performance:

First, fully connected ("vanilla") GANs are capable of producing recognizable, digit-like patterns from a purely random noise input. However, their outputs showed signs of plateau in quality: after many training sessions the generator learns to create only rudimentary shapes (e.g. rough outlines of "1"s and "7"s) and further training yields diminishing returns. The absence of spatial inductive biases in dense layers limits their ability to capture local image structures, leading to low-fidelity, blocky artifacts.

In contrast, deep convolutional GANs (DCGANs) deliver dramatically better results. By employing learned up- and down-sampling via strided convolutional layers, combined with Batch Normalization and appropriate activation functions, DCGANs generate sharp, coherent 64×64 color images with rich textural detail. Training on a GPU further accelerates convergence—per-epoch run times drop from several minutes (CPU) to under one minute—allowing for more extensive hyperparameter exploration. Stability also improves: DCGANs exhibit fewer collapse events, thanks to normalized feature statistics and leaky ReLU gradients.

We also found that small datasets—on the order of a few thousand samples—are sufficient for generating basic game assets. Training a DCGAN on a 1,000-image subset of MNIST produced varied digit-inspired textures, suitable for simple tile-based games. Scaling up to tens of thousands of higher-resolution faces (CelebA) amplified output quality and diversity, demonstrating that dataset size directly correlates with fidelity but is not strictly required for proof-of-concept asset generation.

A. Comparison to Baselines

Approach	**Output**	**Training Time**	**Stability**
Vanilla GAN (NumPy)	Basic digit shapes	Slow (no GPU acceleration)	Unstable without careful hyperparameter tuning
DCGAN (PyTorch)	High-quality 64×64 color textures	Fast (GPU-accelerated)	More stable; fewer collapse events

Running the vanilla GAN purely in NumPy underscored the cost of manual implementation and lack of dedicated deep-learning tooling—training was slow, and minor misconfigurations could derail convergence. By comparison, the PyTorch-based DCGAN pipeline proved both performant and resilient, making it a more practical choice for real-world procedural texture and tile generation in game development.

B. Visualizing GAN Training and Outputs

To illustrate both the learning dynamics and the quality of generated samples for our Vanilla GAN and DCGAN implementations, several complementary figures can be included:

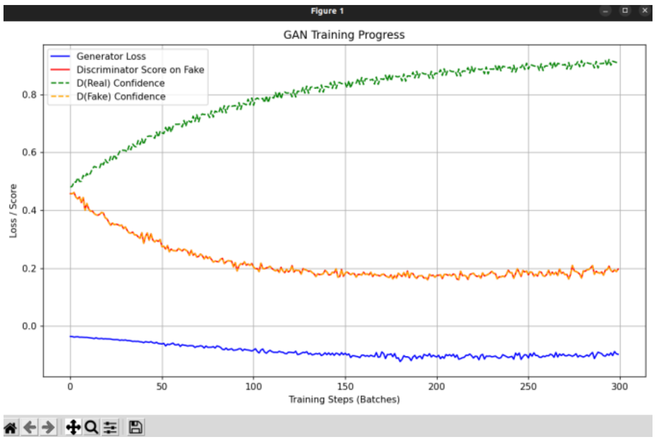

Fig. 2. Vanilla GAN: Initial training losses

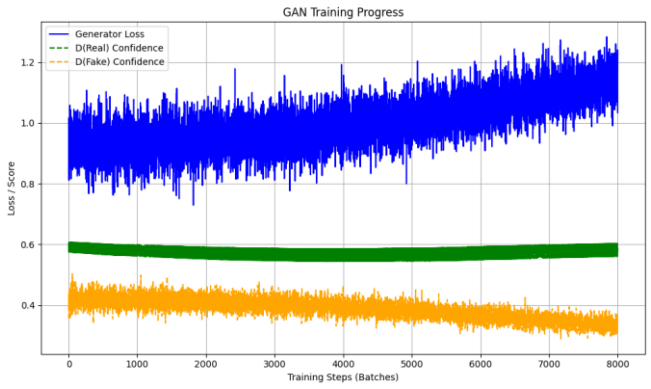

Fig. 3. Vanilla GAN: Failing generator

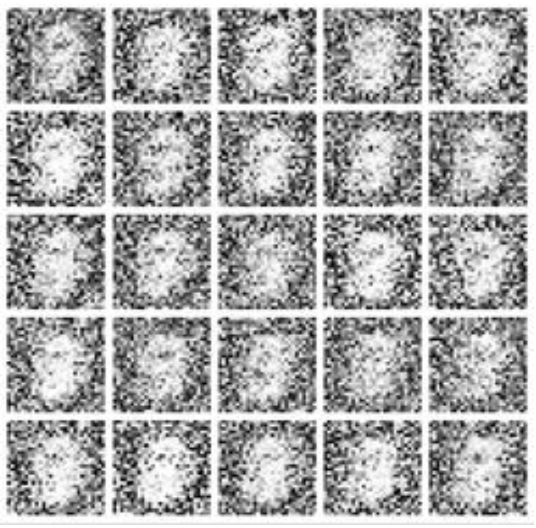

Fig. 4. Vanilla GAN: Mode collapse

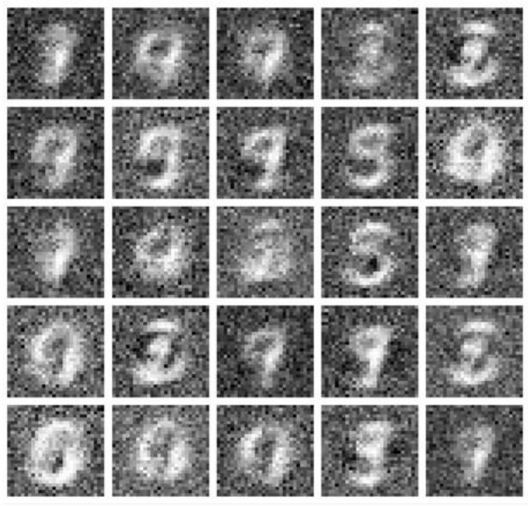

Fig. 5. Vanilla GAN: Mid-training generator output

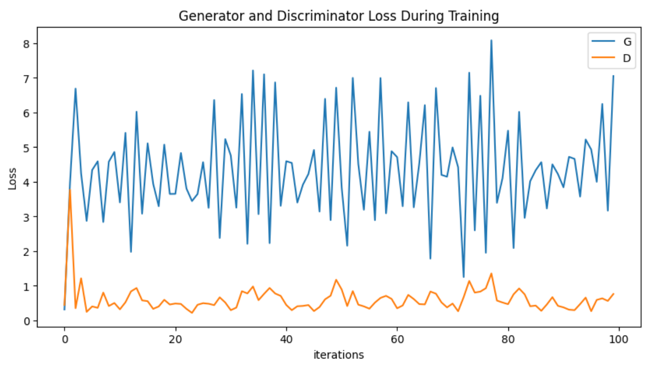

Fig. 6. DCGAN: Single episode loss diagram

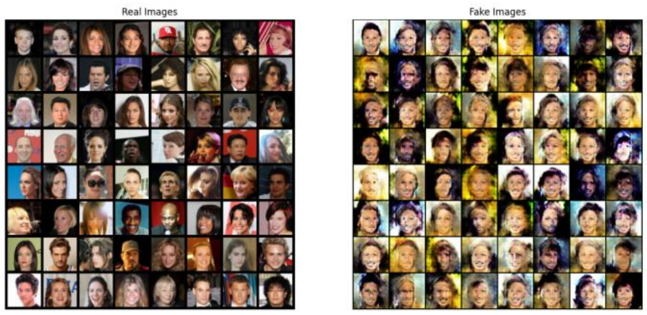

Fig. 7. DCGAN: Single episode output comparison

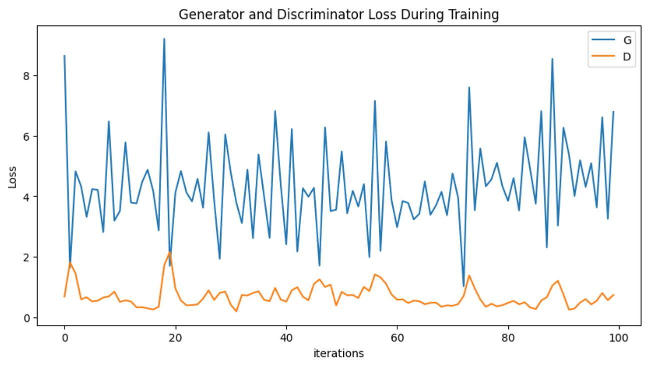

Fig. 8. DCGAN: Fourth episode loss diagram

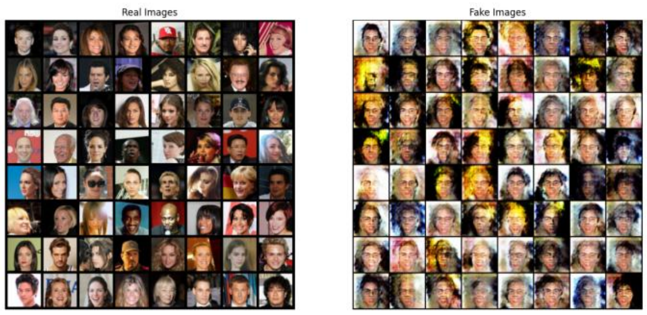

Fig. 9. DCGAN: Fourth episode output comparison

C. *Optimization Techniques*

Beyond raw architecture, a suite of optimization strategies proved essential for stable, efficient GAN training. First, hardware acceleration on GPUs (specifically the Tesla T4 in Google Colab) yielded a 2-8x (median of 5x) speedup over CPU-only runs (with shared usage and network instability causing the gap). This made it feasible to operate on increased training sizes, increased batch sizes, and model depth within practical timeframes.

At the algorithmic level, we found that careful hyperparameter tuning—particularly of the Adam optimizer's learning rate (α) and momentum term (β_1)—was critical. Reducing α from 0.0002 to 0.0001 often prevented mode collapse (specifically within the NumPy implementation), while a β_1 of 0.5 balanced stability and convergence speed.

Architectural regularizers also played a major role. Inserting Batch Normalization after every convolution (and before each activation) smoothed the optimization landscape, reducing internal covariate shift and allowing higher learning rates. Leaky ReLU activations in the discriminator prevented "dead" units and maintained gradient flow even when inputs were negative. In the generator, Tanh outputs ensured generated pixels lay in the same [−1,1] range as the real data, aiding adversarial feedback.

Finally, runtime safeguards such as gradient clipping (via torch.nn.utils.clip_grad_norm_) prevented sporadic gradient spikes from derailing training. Combined with regular checkpointing and visual sample inspection, these optimization techniques transformed otherwise fragile GANs into robust, high-performing texture generators.

VI. Application to Game Development

GAN-based asset generation fundamentally changes how textures and environmental details are created in games. By training a GAN on a curated set of material samples—stone, grass, or earth—developers can automatically produce a virtually unlimited variety of terrain textures. Each generated tile preserves the properties of the source data, ensuring consistent style while allowing variations in color, orientation, and micro-details.

In addition to ground cover, GANs can populate skies and foliage: a cloud model yields unique yet cohesive skyboxes, and a small set of bark or leaf photos scales to hundreds of sprites or texture maps for trees and plants. This removes the need for manual hand-painting of each variant and accelerates both 2D and 3D pipelines.

Architectural elements such as walls and floors also benefit. GANs generate these assets on demand in matching styles, and—with conditional inputs or style-transfer—can switch themes (e.g., mossy vs. sun-baked stone) by tweaking latent variables rather than retraining.

When integrated into a procedural map system, GAN-generated textures eliminate repetitive tiling: dozens of subtly different grass or stone patches blend seamlessly, and biome transitions (such as grass to sand) become smooth, data-driven gradients. Since generation can occur at build-time or even at runtime, each player's world can be unique without inflating storage requirements. Similar to how Minecraft creates landscapes from 2D generated maps, GANs have the same potential with much greater variation. [4]

Combining GANs with traditional noise methods (Perlin noise) or gameplay parameters (biome type, difficulty) lets developers create:

- **Unlimited Texture Variation:** Automatic generation of countless tile and sprite variants.
- **Style Consistency:** Shared visual identity across all generated assets.
- **Dynamic Biome Transitions:** Smooth, data-driven blends between environmental types.
- **Runtime Generation:** On-the-fly asset creation for personalized or infinite worlds.
- **Hybrid Pipelines:** Seamless integration with procedural noise and gameplay logic for enhanced control.

VII. Conclusion

This work confirms that Generative Adversarial Networks are a practical and scalable solution for automated texture and asset generation in game development. By first constructing a fully manual NumPy-based GAN, we gained crucial insights into gradient dynamics, architecture sensitivity, and resource constraints. Transitioning to a PyTorch-powered DCGAN unlocked high-fidelity 64×64 RGB outputs and GPU-accelerated training, demonstrating how convolutional structures, BatchNorm, and carefully tuned hyperparameters yield stable, diverse, and stylistically consistent assets.

Looking ahead, applying these techniques to higher resolutions (128×128 and beyond), fine-tuning models on genre-specific art styles, and integrating real-time inference engines in Unity or Unreal could further streamline production pipelines. Ultimately, GAN-driven workflows promise to reduce artist workload, enhance visual variety, and enable dynamic, personalized environments without sacrificing performance or quality.

VIII. Use of LLMs

Large Language Models helped to expedite this project through generating structured outlines for the paper along with draft checking through grammar and punctuation checks. They offered help on the programming side through code walkthroughs for more advanced python libraries where readily available information and tutorials are harder to find.

References

[1] Rafatirad, S. et al. (2023) Machine learning for computer scientists and data analysts: From an applied perspective. Cham: Springer.

[2] I. J. Goodfellow et al., "Generative Adversarial Networks," arXiv.org, Jun. 10, 2014. https://arxiv.org/abs/1406.2661.

[3] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," *arXiv.org*, 2015. https://arxiv.org/abs/1511.06434.

[4] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, "Procedural content generation for games," ACM Transactions on Multimedia Computing, Communications, and Applications, vol. 9, no. 1, pp. 1–22, Feb. 2013, doi: https://doi.org/10.1145/2422956.2422957.

[5] M. Freiknecht, "Procedural Content Generation for Games." Available: https://madoc.bib.uni-mannheim.de/59000/1/Procedural%20Content%20Generation%20for%20Games.pdf.

[6] A. Summerville et al., "Procedural Content Generation via Machine Learning (PCGML)," IEEE Transactions on Games, vol. 10, no. 3, pp. 257–270, Sep. 2018, doi: https://doi.org/10.1109/tg.2018.2846639.

[7] J. Liu, S. Snodgrass, A. Khalifa, S. Risi, G. N. Yannakakis, and J. Togelius, "Deep learning for procedural content generation," Neural Computing and Applications, vol. 33, no. 1, pp. 19–37, Oct. 2020, doi: https://doi.org/10.1007/s00521-020-05383-8.

[8] "Procedural Content Generation in Games: A Survey with Insights on Emerging LLM Integration," Arxiv.org, 2022. https://arxiv.org/html/2410.15644v1.

[9] "Procedural Content Generation via Generative Artificial Intelligence," Arxiv.org, 2019. https://arxiv.org/html/2407.09013v1?utm_source=chatgpt.com.

[10] "Reinforcement Learning-Enhanced Procedural Generation for Dynamic Narrative-Driven AR Experiences Accepted at GRAPP 2025 - 20th International Conference on Computer Graphics Theory and Applications," Arxiv.org, 2025. https://arxiv.org/html/2501.08552v1.

[11] S. Saffari, M. Dorrigiv, and F. Yaghmaee, "Harnessing Machine Learning for Procedural Content Generation in Gaming: A Comprehensive Review," Technology Journal of Artificial Intelligence and Data Mining, vol. 12, no. 4, pp. 583–597, 2024, doi: https://doi.org/10.22044/jadm.2025.15016.2603.

[12] M. C. Green, L. Mugrai, A. Khalifa, and J. Togelius, "Mario Level Generation From Mechanics Using Scene Stitching," arXiv.org, 2020. https://arxiv.org/abs/2002.02992.

[13] A. Sarkar, Z. Yang, and S. Cooper, "Conditional Level Generation and Game Blending," arXiv.org, 2020. https://arxiv.org/abs/2010.07735.

[14] I. Srivastava, "A comparative analysis of generative models for terrain generation in open-world video games," Journal of High School Science, vol. 8, no. 1, pp. 120–143, Feb. 2024, Accessed: Feb. 21, 2025. [Online]. Available: https://jhss.scholasticahq.com/article/92856-a-comparative-analysis-of-generative-models-for-terrain-generation-in-open-world-video-games.

[15] "A Case Study of Generative Adversarial Networks for Procedural Synthesis of Original Textures in Video Games | IEEE Conference Publication | IEEE Xplore," ieeexplore.ieee.org. https://ieeexplore.ieee.org/document/8712070.

[16] L. Z. Kelvin and Anand Bhojan, "Procedural Generation of Roads with Conditional Generative Adversarial Networks," pp. 1–2, Aug. 2020, doi: https://doi.org/10.1145/3388770.3407422.

[17] M .-Y. Liu, X. Huang, J. Yu, T.-C. Wang, and A. Mallya, "Generative Adversarial Networks for Image and Video Synthesis: Algorithms and Applications," Proceedings of the IEEE, pp. 1–24, 2021, doi: https://doi.org/10.1109/jproc.2021.3049196.

[18] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, "Generative Adversarial Networks: An Overview," IEEE Signal Processing Magazine, vol. 35, no. 1, pp. 53–65, Jan. 2018, doi: https://doi.org/10.1109/msp.2017.2765202.

[19] A. Aggarwal, M. Mittal, and G. Battineni, "Generative adversarial network: An overview of theory and applications," International Journal of Information Management Data Insights, vol. 1, no. 1, p. 100004, Jan. 2021, doi: https://doi.org/10.1016/j.jjimei.2020.100004.

[20] M. Ben-Yosef and D. Weinshall, "Gaussian Mixture Generative Adversarial Networks for Diverse Datasets, and the Unsupervised Clustering of Images," arXiv.org, 2018. https://arxiv.org/abs/1808.10356.

[21] A. Wulff-Jensen, N. N. Rant, T. N. Møller, and J. A. Billeskov, "Deep Convolutional Generative Adversarial Network for Procedural 3D Landscape Generation Based on DEM," Springer eBooks, pp. 85–94, Jan. 2018, doi: https://doi.org/10.1007/978-3-319-76908-0_9.

[22] A. Hald, J. S. Hansen, J. Kristensen, and Paolo Burelli, "Procedural Content Generation of Puzzle Games using Conditional Generative Adversarial Networks," arXiv (Cornell University), pp. 1–9, Sep. 2020, doi: https://doi.org/10.1145/3402942.3409601.

[23] K. Ping and Luo Dingli, "Conditional Convolutional Generative Adversarial Networks Based Interactive Procedural Game Map Generation," Advances in intelligent systems and computing, pp. 400–419, Jan. 2020, doi: https://doi.org/10.1007/978-3-030-39445-5_30.

[24] inus Gisslén, A. Eakins, C. Gordillo, J. Bergdahl, and Konrad Tollmar, “Adversarial Reinforcement Learning for Procedural Content Generation,” arXiv (Cornell University), Aug. 2021, doi: https://doi.org/10.1109/cog52621.2021.9619053.

[25] M. Saito, E. Matsumoto, and S. Saito, “Temporal Generative Adversarial Nets with Singular Value Clipping,” International Conference on Computer Vision, Oct. 2017, doi: https://doi.org/10.1109/iccv.2017.308.

[26] Z. Lin, A. Khetan, G. Fanti, and S. Oh, “PacGAN: The power of two samples in generative adversarial networks,” arXiv.org, 2017. https://arxiv.org/abs/1712.04086.

[27] T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida, “Spectral Normalization for Generative Adversarial Networks,” arXiv:1802.05957 [cs, stat], Feb. 2018, Available: https://arxiv.org/abs/1802.05957.

[28] B. Gan, Y. Saatchi, and A. Wilson, Accessed: Feb. 21, 2025. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/312351bff07989769097660a56395065-Paper.pdf.

[29] Z. Guo, H. Liu, Y.-S. Ong, X. Qu, Y. Zhang, and J. Zheng, “Generative Multiform Bayesian Optimization,” IEEE Transactions on Cybernetics, vol. 53, no. 7, pp. 4347–4360, Jul. 2023, doi: https://doi.org/10.1109/tcyb.2022.3165044.

[30] T. S. Rodrigues and P. R. Pinheiro, “Hyperparameter Optimization in Generative Adversarial Networks (GANs) Using Gaussian AHP,” IEEE Access, pp. 1–1, Jan. 2024, doi: https://doi.org/10.1109/access.2024.3518979.

[31] Lars Mescheder, S. Nowozin, and A. Geiger, “Adversarial Variational Bayes: Unifying Variational Autoencoders and Generative Adversarial Networks,” PMLR, pp. 2391–2400, Jul. 2017, Available: https://proceedings.mlr.press/v70/mescheder17a.html?ref=https://githubhelp.com.

[32] S. Vivekananthan, “Comparative Analysis of Generative Models: Enhancing Image Synthesis with VAEs, GANs, and Stable Diffusion,” arXiv.org, 2024. https://arxiv.org/abs/2408.08751.

[33] S. Bond-Taylor, A. Leach, Y. Long, and C. G. Willcocks, “Deep Generative Modelling: a Comparative Review of VAEs, GANs, Normalizing Flows, Energy-Based and Autoregressive Models,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 44, no. 11, pp. 1–1, 2021, doi: https://doi.org/10.1109/TPAMI.2021.3116668.

[34] D. Bank, N. Koenigstein, and R. Giryes, “Autoencoders,” arXiv:2003.05991 [cs, stat], Apr. 2021, Available: https://arxiv.org/abs/2003.05991.

[35] Umberto Michelucci, “An Introduction to Autoencoders,” arXiv (Cornell University), Jan. 2022, doi: https://doi.org/10.48550/arxiv.2201.03898.

[36] P. Baldi, “Autoencoders, Unsupervised Learning, and Deep Architectures,” vol. 27, pp. 37–50, 2012, Available: https://proceedings.mlr.press/v27/baldi12a/baldi12a.pdf.

[37] Liang, R. G. Krishnan, M. D. Hoffman, and T. Jebara, “Variational Autoencoders for Collaborative Filtering,” Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW ’18, 2018, doi: https://doi.org/10.1145/3178876.3186150.

[38] A. Cemgil et al., “The Autoencoding Variational Autoencoder.” Available: https://papers.nips.cc/paper/2020/file/ac10ff1941c540cd87c107330996f4f6-Paper.pdf.

[39] “Papers with Code - VAE Explained,” Paperswithcode.com, 2020. https://paperswithcode.com/method/vae.

[40] D. P. Kingma and M. Welling, “An Introduction to Variational Autoencoders,” Foundations and Trends® in Machine Learning, vol. 12, no. 4, pp. 307–392, 2019, doi: https://doi.org/10.1561/2200000056.

[41] D. P. Kingma and M. Welling, “Auto-Encoding Variational Bayes,” arXiv.org, Dec. 20, 2013. https://arxiv.org/abs/1312.6114.

[42] .-T. Truong, A. Salah, and H. W. Lauw, “Bilateral Variational Autoencoder for Collaborative Filtering,” Web Search and Data Mining, Mar. 2021, doi: https://doi.org/10.1145/3437963.3441759.

[43] A. Sarkar and S. Cooper, “Sequential Segment-based Level Generation and Blending using Variational Autoencoders,” arXiv.org, 2020. https://arxiv.org/abs/2007.08746?utm_source=chatgpt.com.

[44] S. Thakkar, C. Cao, L. Wang, Tae Jong Choi, and J. Togelius, “Autoencoder and Evolutionary Algorithm for Level Generation in Lode Runner,” 2019 IEEE Conference on Games (CoG), Aug. 2019, doi: https://doi.org/10.1109/cig.2019.8848076.

[45] A. Sarkar, Z. Yang, and S. Cooper, “Controllable Level Blending between Games using Variational Autoencoders,” arXiv (Cornell University), Feb. 2020, doi: https://doi.org/10.48550/arxiv.2002.11869.

[46] S. Saito, L. Wei, L. Hu, K. Nagano, and H. Li, “Photorealistic Facial Texture Inference Using Deep Neural Networks,” arXiv.org, 2016. https://arxiv.org/abs/1612.00523.

[47] Y. Song, L. Bao, S. He, Q. Yang, and M.-H. Yang, “Stylizing face images via multiple exemplars,” Computer Vision and Image Understanding, vol. 162, pp. 135–145, Sep. 2017, doi: https://doi.org/10.1016/j.cviu.2017.08.009.

[48] Y. Zhou et al., “HairNet: Single-View Hair Reconstruction using Convolutional Neural Networks,” arXiv.org, 2018. https://arxiv.org/abs/1806.07467.

[49] K. Wang, M. Savva, A. X. Chang, and D. Ritchie, “Deep convolutional priors for indoor scene synthesis,” ACM Transactions on Graphics, vol. 37, no. 4, pp. 1–14, Aug. 2018, doi: https://doi.org/10.1145/3197517.3201362.

[50] Anđelo Martinović and Luc Van Gool, “Bayesian Grammar Learning for Inverse Procedural Modeling,” CiteSeer X (The Pennsylvania State University), Jun. 2013, doi: https://doi.org/10.1109/cvpr.2013.33.

[51] İ. Demir and D. G. Aliaga, “Guided proceduralization: Optimizing geometry processing and grammar extraction for architectural models,” Computers & Graphics, vol. 74, pp. 257–267, Aug. 2018, doi: https://doi.org/10.1016/j.cag.2018.05.013.

[52] J. O. Talton, L. Yang, R. Kumar, M. Lim, N. D. Goodman, and R. Mech, “Learning design patterns with bayesian grammar induction,” CiteSeer X (The Pennsylvania State University), Oct. 2012, doi: https://doi.org/10.1145/2380116.2380127.

[53] T. Kelly, P. Guerrero, A. Steed, P. Wonka, and N. J. Mitra, “FrankenGAN,” ACM Transactions on Graphics, vol. 37, no. 6, pp. 1–14, Dec. 2018, doi: https://doi.org/10.1145/3272127.3275065.

[54] C. Beckham and C. Pal, “A step towards procedural terrain generation with GANs,” arXiv.org, 2017. https://arxiv.org/abs/1707.03383.

[55] C. Tang, K. Zhang, C. Xing, Y. Ding, and Z. Xu, “Perlin Noise Improve Adversarial Robustness,” arXiv.org, 2021. https://arxiv.org/abs/2112.13408.

[56] H.-J. Bae et al., “A Perlin Noise-Based Augmentation Strategy for Deep Learning with Small Data Samples of HRCT Images,” Scientific Reports, vol. 8, no. 1, p. 17687, Dec. 2018, doi: https://doi.org/10.1038/s41598-018-36047-2.

[57] H. Irobe et al., “Robust VAEs via Generating Process of Noise Augmented Data,” arXiv.org, 2024. https://arxiv.org/abs/2407.18632.

[58] T. Kaneko and T. Harada, “Noise Robust Generative Adversarial Networks,” arXiv.org, 2019. https://arxiv.org/abs/1911.11776.

[59] Y. Peng, “A Comparative Analysis Between GAN and Diffusion Models in Image Generation,” vol. 5, pp. 189–195, Aug. 2024, doi: https://doi.org/10.62051/0f1va465.

[60] S. J. Barigye, J. M. García de la Vega, and Y. Perez-Castillo, “Generative Adversarial Networks (GANs) Based Synthetic Sampling for Predictive Modeling,” Molecular Informatics, Jul. 2020, doi: https://doi.org/10.1002/minf.202000086.

[61] Pedro, R. Lorenz, R. P. Monti, E. Jones, and R. Leech, “Bayesian optimization for automatic design of face stimuli,” arXiv.org, 2020. https://arxiv.org/abs/2007.09989.

[62] C. Badrinath, U. Bhalla, A. Oesterling, S. Srinivas, and H. Lakkaraju, “All Roads Lead to Rome? Exploring Representational Similarities Between Latent Spaces of Generative Image Models,” arXiv.org, 2024. https://arxiv.org/abs/2407.13449?utm_source=chatgpt.com (accessed Feb. 21, 2025).

[63] J. Johnson, A. Alahi, and L. Fei-Fei, “Perceptual Losses for Real-Time Style Transfer and Super-Resolution,” Computer Vision – ECCV 2016, pp. 694–711, 2016, doi: https://doi.org/10.1007/978-3-319-46475-6_43.

[64] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, “Search-Based Procedural Content Generation: A Taxonomy and Survey,” IEEE Transactions on Computational Intelligence and AI in

Games, vol. 3, no. 3, pp. 172–186, Sep. 2011, doi: https://doi.org/10.1109/tciaig.2011.2148116.

[65] . Sarkar and S. Cooper, "Towards Game Design via Creative Machine Learning (GDCML)," arXiv.org, 2020. https://arxiv.org/abs/2008.13548.

[66] S. Snodgrass and S. Ontanon, "Procedural level generation using multi-layer level representations with MdMCs," Aug. 2017, doi: https://doi.org/10.1109/cig.2017.8080447.

[67] X. Wei, B. Gong, Z. Liu, W. Lu, and L. Wang, "Improving the Improved Training of Wasserstein GANs: A Consistency Term and Its Dual Effect," arXiv.org, 2018. https://arxiv.org/abs/1803.01541.

[68] T. Hu et al., "Complexity Matters: Rethinking the Latent Space for Generative Modeling," arXiv.org, 2023. https://arxiv.org/abs/2307.08283.

[69] "Welcome to PyTorch Tutorials — PyTorch Tutorials 2.7.0+cu126 documentation," *Pytorch.org*, 2024. https://docs.pytorch.org/tutorials (accessed May 08, 2025).

APPENDIX

A. *Hyperparameter Configurations*

Phase	Learning Rate	Optimizer	Batch Size	Latent Dim	Epochs
Vanilla GAN (NumPy)	0.0001-0.002 (Avg. 0.0002)	SGD	32-128 (Avg. 64)	32-128 (Avg. 100)	10-100 (Avg. 30)
DCGAN (PyTorch)	0.0001-0.001 (Avg. 0.0002)	Adam ($\beta_1 = 0.5$)	64-128 (Avg. 128)	100	100

B. *Model Architectures*

Model	Architecture
Vanilla GAN (Generator)	Noise (100D) → Dense(256) → ReLU → Dense(128) → ReLU → Dense(784) → Tanh
Vanilla GAN (Discriminator)	Image (784D) → Dense(256) → Leaky ReLU → Dense(128) → Leaky ReLU → Dense(1) → Sigmoid
DCGAN (Generator)	Noise (100D) → ConvTranspose(512) → BatchNorm → ReLU → ConvTranspose(256) → ... → Output (64x64x3)
DCGAN (Discriminator)	Input (64x64x3) → Conv(64) → LeakyReLU → Conv(128) → ... → Dense(1) → Sigmoid

C. *Dataset Details*

MNIST

- 70,000 total grayscale images, 28x28 pixels
- Normalization: pixel values scaled from [0,255] to [-1,1]

CelebA

- ~200K face images, aligned and cropped
- Preprocessed by resizing to 64x64, center-crop, normalization [-1,1]

D. *Compute Environment*

Vanilla GAN: Ubuntu Linux, Windows, Google Colab, Python 3.8, NumPy 1.21, CPU only (13th Gen Intel(R) Core(TM) i9-13900K, 3000 Mhz, 24 Cores, 32 Logical Processors), 32Gb memory.
DCGAN: Google Colab with Tesla T4 GPU, PyTorch 1.12, CUDA 11.3

E. *Code Availability*

NumPy Vanilla GAN (MNIST): https://colab.research.google.com/drive/1VDCtp0m9jOVZud85ZfjdXKnz82WtKNJV?usp=sharing.

PyTorch DCGAN (CELEBA): https://colab.research.google.com/drive/1r2nqc3h9HJRVmZDjBE6B8QbeOPMuQfyX?usp=sharing.