**1. Introduction**

Generative Adversarial Networks (GANs), introduced by Ian Goodfellow et al. in 2014, are a class of machine learning frameworks where two neural networks,  generator and the discriminator, compete in a zero-sum game. The generator creates fake data resembling the real data, while the discriminator attempts to distinguish between real and generated data. Over time, the generator improves its ability to create realistic data samples. In this study, we explore the generative capabilities of a simple GAN using the MNIST dataset, a well-known dataset of handwritten digits. Our goal is to observe the effects of tuning parameters on the quality of generated digits.

**2. Dataset Description**

We used the **MNIST dataset**, which contains 70,000 grayscale images of handwritten digits (60,000 for training and 10,000 for testing). Each image is 28x28 pixels, and labels range from 0 to 9. For this study, only the image data was used, as GANs are unsupervised in nature.

**3. GAN Architecture and Tuning Parameters**

A Generative Adversarial Network (GAN) consists of two core components: the Generator and the Discriminator. The Generator creates images from random noise, while the Discriminator evaluates whether images are real or generated. Both networks are trained in opposition, with the Generator aiming to fool the Discriminator, and the Discriminator striving to correctly classify real and fake images.

**3.1 Generator**

The Generator transforms a noise vector into a 28×28 image resembling handwritten digits. It begins with a fully connected layer to expand the noise into a higher-dimensional space. This is followed by additional linear layers that increase feature complexity. **ReLU** activations are used throughout, except in the final layer, which uses **Tanh** to scale outputs to the [-1, 1] range, matching MNIST’s normalized input.

| **Layer** | **Purpose** |
| --- | --- |
| **Linear(100 → 256)** | Transforms low-dimensional noise into a richer representation. |
| **ReLU activation** | Introduces non-linearity and helps gradients flow better. |
| **Linear(256 → 512 → 1024)** | Progressively increases feature complexity and dimensionality. |
| **Linear(1024 → 784)** | Outputs a vector of 784 = 28×28 pixels, i.e., an entire image. |
| **Tanh activation** | Scales pixel values to [-1, 1], matching the normalized input range. |

**3.2 Discriminator**

The Discriminator serves as a binary classifier that distinguishes between real and generated images. It flattens the 28×28 image into a vector and passes it through fully connected layers. LeakyReLU is used to allow gradients even for negative values, and a Sigmoid activation at the end outputs a probability between 0 and 1.

| **Layer** | **Purpose** |
| --- | --- |
| **Flatten input (28×28 → 784)** | Converts image into 1D vector for fully connected layers. |
| **Linear(784 → 1024)** | First hidden layer extracts high-level features. |
| **LeakyReLU(0.2)** | Allows small gradient for negative inputs, avoids dead neurons. |
| **Linear(1024 → 512 → 256)** | Reduces dimensionality, focusing features down to decision-critical ones. |
| **Linear(256 → 1)** | Outputs probability that input is real (1) or fake (0). |

**3.3 Training and Variants**

In this project, we implemented a Vanilla GAN, the simplest form of a Generative Adversarial Network, using fully connected layers in both the Generator and Discriminator—ideal for datasets like MNIST. Through adversarial training, the Generator gradually produces more realistic images while the Discriminator improves at detecting fakes. Several advanced variants build on this architecture: DCGAN adds deeper convolutional layers for better spatial understanding and image quality; Conditional GANs incorporate class labels for more controlled generation; and Wasserstein GANsmodify the loss function to enhance training stability and prevent mode collapse.

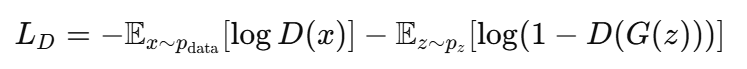
**3.4 Tuning Parameters Used**

| **Parameter** | **Value** |
| --- | --- |
| Latent Dimension (z) | 100 |
| Batch Size | 128 |
| Learning Rate | 0.0002 |
| Optimizer | Adam (ß1 = 0.5, ß2 = 0.999) |
| Loss Function | Binary Cross Entropy (BCDLoss) |
| Epochs | 50 |
| Activation Gen | ReLU, Tanh |
| Activation Disc | LeakyReLU, Sigmoid |

These parameters were selected based on common practices in GAN literature and through light experimentation to ensure convergence and stable training.

**3.5 GAN Loss function**

The loss function in a GAN defines the adversarial dynamic between the Generator and the Discriminator by framing their interaction as a two-player minimax optimization problem. During training, the Discriminator aims to correctly classify real samples and reject generated ones. Its loss function is defined as:  
The Generator, in turn, is optimized to produce outputs that the Discriminator classifies as real. Its loss function is given by:  
A black text on a white background

AI-generated content may be incorrect.

This formulation encourages the Generator to increase the likelihood that the Discriminator assigns high confidence to generated samples. In practice, this version of the Generator loss is preferred over minimizing *log(1-D(G(z)),* as it provides stronger gradients early in training when the Discriminator is more accurate.

A key design choice in implementing these loss functions is the use of *m* samples from both the real data distribution and the noise distribution in each training iteration. Using equal numbers of real and generated samples ensures a balanced comparison and prevents the Discriminator from becoming biased due to unequal data exposure. While this symmetry simplifies training and stabilizes gradients, many real-world applications vary the number of noise and real samples to accommodate computational constraints or task-specific goals. However, such imbalance must be handled carefully, as it can affect the adversarial dynamics and overall training stability. Selecting appropriate sampling strategies is thus essential for effective GAN performance.

**4. Results and Figures**

**4.1 Sample Output from Generator (Best Model at Epoch 50)**

The first image below shows the Generator’s output after epoch 1, using a randomly initialized noise vector. At this early stage, the model has not yet learned to generate coherent digits. The second image shows the output for epoch 50, where the Generator has significantly improved and is producing recognizable handwritten digits.

A grid of squares with black and white squares

AI-generated content may be incorrect.A number written in white on a black background

AI-generated content may be incorrect.

**4.2 Training Curves**

The graph below shows generator and discriminator loss over 50 epochs. Initially, generator loss rises sharply as it struggles to produce realistic images, while discriminator loss remains low due to its ease in detecting fakes. Around epoch 15, both losses stabilize—generator loss decreases with improved output, and discriminator loss rises slightly as the task becomes harder. This reflects the adversarial nature of GANs, where both networks adapt over time. The flattening curves indicate convergence and a stable equilibrium.

**A graph with a line

AI-generated content may be incorrect.**

**5. Impact of Tuning Parameters**

* **Learning Rate**: A rate of 0.0002 gave the most stable convergence. A higher rate (0.0004) led to generator instability (mode collapse), while 0.0001 slowed training. The model Converged smoothly with this learning rate.
* **Batch Size**: Batch size of 128 balanced training speed and convergence. 64 was too noisy; 256 showed slower discriminator learning.
* **Latent Space**: 100 was perfect latent space for this specific data set. Would increase depending on the complexity of the image ie. colour, resolution etc.
* **Generator Depth (Layers)**: Adding an extra layer increased the quality of images but slightly increased training time. 4 layers for the generator and the discriminator was perfect.

**6. Interpretation of Results**

After 50 epochs, the generator was able to produce images that visually resembled digits in the MNIST dataset. While not perfect, most generated digits were easily identifiable by human observers. In plain language: the model "learned" what a digit looks like by observing many examples, then tried to draw its own digits. Over time, its fake digits became good enough to fool the discriminator model—and even a human viewer. The final model shows that GANs, even in a simple form, can effectively learn to generate realistic samples from basic datasets like MNIST

**7. GANs in Cryptography**

In *"Learning to Protect Communications with Adversarial Neural Cryptography,"* GAN-inspired methods are used to train neural networks to learn secure communication protocols without predefined encryption. The setup involves three networks: Alice (sender), Bob (receiver), and Eve (adversary). Alice encrypts messages with a shared key, Bob learns to decrypt them, and Eve attempts to decode them without the key. This creates an adversarial training dynamic similar to GANs. Alice acts like a Generator, producing ciphertext; Bob assists by decoding it, and Eve plays a Discriminator-like role, trying to break the encryption. The system evolves so that Alice generates messages Bob can decode, but Eve cannot. This shows how GANs can be used in cryptography to autonomously develop secure, end-to-end communication resilient to eavesdropping.

**8. References**

1. DeepFindr. (n.d.). *Generative Adversarial Networks [YouTube video]*. YouTube. <https://youtu.be/OW_2zFqQ1TA>
2. DeepFindr. (n.d.). *Application of GANs in Materials Science [YouTube video]*. YouTube. <https://youtu.be/jCXGKWDd4f8>
3. Wikipedia contributors. (n.d.). *Generative adversarial network*. Wikipedia. <https://en.wikipedia.org/wiki/Generative_adversarial_network>
4. Neptune.ai. (2023). *GAN Loss Functions: A Guide for Practitioners*. <https://neptune.ai/blog/gan-loss-functions>
5. Brownlee, J. (2021). *Generative Adversarial Network Loss Functions*. Machine Learning Mastery. <https://machinelearningmastery.com/generative-adversarial-network-loss-functions/>
6. IBM Developer. (2020). *Generative Adversarial Networks explained*. <https://developer.ibm.com/articles/generative-adversarial-networks-explained/>
7. Techtarget. (n.d.). *CNN vs GAN: How are they different?*<https://www.techtarget.com/searchenterpriseai/feature/CNN-vs-GAN-How-are-they-different>
8. Abadi, M., Andersen, D. G., & Shlens, J. (2016). *Learning to Protect Communications with Adversarial Neural Cryptography*. arXiv preprint arXiv:1610.06918. <https://arxiv.org/pdf/1610.06918v1>
9. Jeong, H., et al. (2019). *Figure 3*. Nature Scientific Reports. <https://www.nature.com/articles/s41598-019-52737-x>
10. AIMultiple. (2023). *Top GAN Use Cases in 2023*. <https://research.aimultiple.com/gan-use-cases/>
11. Abadi, M. (2018). *A Basic Intro to GANs: Generative Adversarial Networks*. Towards Data Science. <https://towardsdatascience.com/a-basic-intro-to-gans-generative-adversarial-networks-c62acbcefff3/>
12. Abu-Srhan, A., Abushariah, M. A., & Al-Kadi, O. S. (2022). *The effect of loss function on conditional generative adversarial networks*. *Journal of King Saud University - Computer and Information Sciences*, 34(9), 6977–6988. <https://doi.org/10.1016/j.jksuci.2022.02.018>
13. Brownlee, J. (2021). *A Tour of Generative Adversarial Network Models*. Machine Learning Mastery. <https://machinelearningmastery.com/tour-of-generative-adversarial-network-models/>
14. Restack. (n.d.). *Adversarial Networks Answer: Components of GAN Architecture*. <https://www.restack.io/p/adversarial-networks-answer-components-of-gan-architecture-cat-ai>
15. Neptune.ai. (n.d.). *Popular GAN Architectures*. <https://neptune.ai/blog/6-gan-architectures>