POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

NAML Project Report

Author(s): **Abdallah Alkhetiar**
Luigi Inguaggiato

Academic Year: 2024-2025

Contents

1 | Introduction

In this project, we implement a GAN using the JAX library, a high-performance numerical computing framework that enables efficient automatic differentiation and GPU acceleration. Our GAN is designed to generate images from the MNIST dataset, which consists of handwritten digits (0–9). The generator learns to produce realistic digit images, while the discriminator evaluates their authenticity. The adversarial training process continues until the generator produces samples that the discriminator can no longer reliably classify as fake.

The project leverages the Flax library for building and training the neural networks, alongside Optax for optimization. Key hyperparameters, such as the learning rates for the generator and discriminator, the latent space dimension, and the number of training epochs, are carefully tuned to achieve optimal performance. Additionally, we incorporate techniques such as checkpointing to save and restore model states, ensuring robustness during training.

The primary objectives of this project are:

- To implement and train a GAN capable of generating realistic MNIST digit images.
- To evaluate the quality of the generated images using both qualitative inspection and quantitative metrics, such as the Fréchet Inception Distance (FID).
- To explore the impact of hyperparameter tuning on the GAN's performance and stability.

This report documents the methodology, implementation details, and experimental results of our GAN. We begin by describing the dataset and preprocessing steps, followed by an overview of the model architecture and training procedure. Finally, we present the generated results and discuss potential avenues for future improvements.

2 | Data Loading and Preprocessing

We use the MNIST dataset, downloaded from *Kaggle*, consisting of 70,000 grayscale images of handwritten digits (0–9) with dimensions of 28×28 pixels. The dataset is divided into 60,000 training samples and 10,000 test samples. Below, we detail the steps taken to load and preprocess the data for training our GAN.

2.1. Dataset Loading

The MNIST dataset is loaded directly from binary files stored in the `MNIST` directory. The data is read using the following steps:

1. **Loading Images and Labels**: The training and test images, along with their corresponding labels, are loaded from the binary files `train-images-idx3-ubyte` and `train-labels-idx1-ubyte` (for training), and `t10k-images-idx3-ubyte` and `t10k-labels-idx1-ubyte` (for testing). The images are stored in a flattened format of 784 pixels (28×28) and later reshaped into their original dimensions.

2. **Reshaping**: The flattened images are reshaped into a 4D tensor of dimensions (batch_size, height, width, channels), where:

    - `batch_size` is the number of samples in each batch.
    - `height` and `width` are both 28 pixels.
    - `channels` is 1 (since MNIST images are grayscale).

2.2. Data Preprocessing

To improve training stability and convergence, the following preprocessing steps are applied:

1. **Normalization**: The pixel values, originally in the range $[0, 255]$, are scaled to $[-1, 1]$ using the transformation:

$$\text{normalized\_image} = 2 \times \left(\frac{\text{image}}{255.0} \right) - 1$$

   This scaling is crucial for GAN training as it aligns the input distribution with the activation functions used in the network (e.g., tanh in the generator).

2. **Shuffling**: The training dataset is shuffled to ensure that the model does not learn any unintended order-based biases. This is done using a random permutation of indices generated via JAX's pseudorandom number generator (PRNG).

3. **Training-Validation Split**: The training set is further divided into:

   - A **training subset** (50,000 samples).
   - A **validation subset** (10,000 samples).

   The validation set is used to monitor the model's performance during training but is not directly used for updating weights.

2.3. Final Dataset Structure

After preprocessing, the datasets are structured as follows:

Subset	**Images**	**Shape**
Training	50,000	$(28, 28, 1)$
Validation	10,000	$(28, 28, 1)$
Test	10,000	$(28, 28, 1)$

Table 2.1: MNIST dataset split and shapes.

The class distribution within the training set is reasonably balanced, with each digit class represented by approximately 5,000 to 6,700 samples, as shown in Figure 2.1.

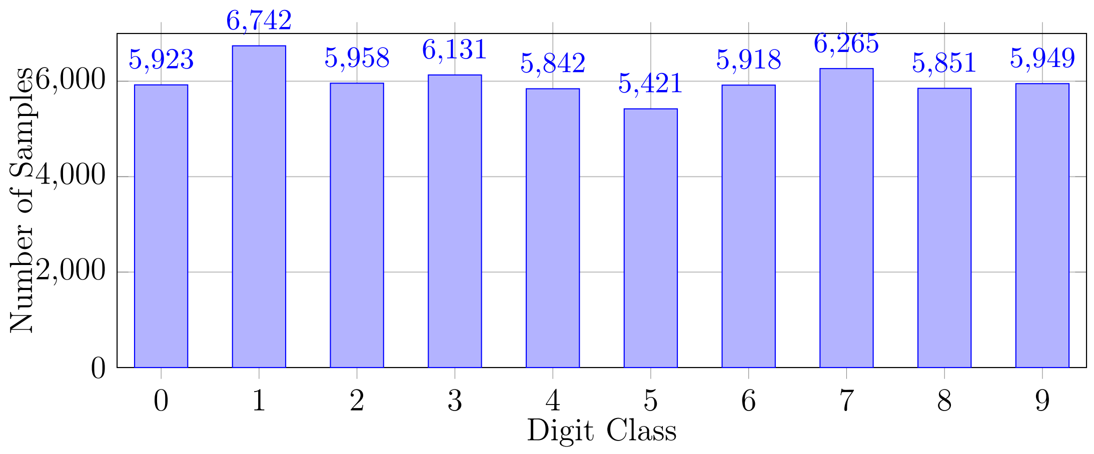

Figure 2.1: Class distribution of digits in the MNIST training set.

3 | Networks' Architectures

This chapter describes the architectures used for the generator and discriminator networks of the conditional Generative Adversarial Network (cGAN) implemented in this project. Both networks were built using `Flax`, a neural network library for JAX, and are conditioned on class labels from the MNIST dataset.

3.1. Generator Architecture

The generator transforms a latent vector $z \in \mathbb{R}^d$ (where $d = 128$ is the latent dimension) into a synthetic grayscale image of shape $28 \times 28 \times 1$. In this implementation, the generator is constructed using fully connected (dense) layers. The network is also conditioned on class labels through one-hot encoding, which is concatenated with the latent vector before entering the first layer.

Table 3.1: Generator Architecture

Layer	**Output Shape**	**Activation**	**Description**
Input (z + one-hot label)	$d + 10$	—	Concatenation of latent noise vector and one-hot encoded class label.
Dense	128	ReLU	Fully connected layer projecting input to 128 dimensions.
Dense	256	ReLU	Expands feature space for further transformation.
Dense	784	`tanh`	Maps to flattened image representation with values in $[-1, 1]$.
Reshape	$(28, 28, 1)$	—	Reshapes the 1D output into a 2D image.

Key Design Choices:

- **Latent Space:** The latent vector $z \sim \mathcal{N}(0, I)$ is sampled from a standard normal distribution.

- **Label Conditioning:** Class labels are encoded using one-hot encoding and concatenated with the noise vector prior to the first layer, enabling class-conditional image generation.
- **Fully Connected Layers:** The generator leverages dense layers to map the latent space to the image domain.
- **Activation Functions:** ReLU activations are used in the hidden layers to introduce non-linearity, while a `tanh` activation is used at the output to map pixel values to the range $[-1, 1]$, consistent with the normalized image data.

3.2. Discriminator Architecture

The discriminator is designed to distinguish between real and generated images, while also incorporating class label information. It receives as input a grayscale image of shape $28 \times 28 \times 1$, along with an associated class label. The image is flattened and concatenated with an embedded representation of the label. The combined input is passed through a series of fully connected layers to produce a single scalar output, representing the probability that the input image is real.

Table 3.2: Discriminator Architecture

Layer	**Output Shape**	**Activation**	**Description**
Input (flattened image)	784	—	Grayscale image of shape 28×28.
Label Embedding	d_e	—	Embedded class label vector.
Concatenate	$784 + d_e$	—	Concatenation of image and label embeddings.
Dense	256	Leaky ReLU ($\alpha = 0.2$)	Fully connected layer.
Dense	128	Leaky ReLU ($\alpha = 0.2$)	Fully connected layer.
Dense	1	Sigmoid	Outputs probability that the input is real.

Key Design Choices:

- **Label Embedding:** Class labels are passed through an embedding layer of size d_e, where $d_e = 10$ (embedding dimension), allowing the model to learn a continuous representation of each class.
- **Input Fusion:** The flattened image is concatenated with the label embedding to condition the discriminator on the class identity.
- **Fully Connected Layers:** A series of dense layers are used to process the concatenated input, reducing it to a scalar probability.
- **Activation Functions:** Leaky ReLU is used in hidden layers to prevent dying neuron problems, while a sigmoid activation in the output layer maps the result to the range $[0, 1]$.

3.3. Weight Initialization

All dense layers in both networks are initialized using Xavier uniform initialization, which is well-suited for layers with ReLU or tanh activations. This helps to maintain the scale of gradients throughout the network and accelerates convergence during training.

4 | Loss Functions and Optimization

The training dynamics of a GAN hinge on the adversarial interplay between the generator (G) and discriminator (D), formalized through carefully designed loss functions. This section details the loss functions used, their mathematical formulations, and their role in training stability.

4.1. Adversarial Objective

The GAN framework formulates a two-player minimax game between:

- The **discriminator** D, which aims to **maximize** its ability to distinguish:
  - Real images ($\mathbf{x} \sim p_{\text{data}}$) from the training dataset
  - Fake images ($G(\mathbf{z})$) where $\mathbf{z} \sim p_{\mathbf{z}}$ is random noise
- The **generator** G, which aims to **minimize** D's discrimination accuracy by producing realistic samples that fool D.

4.1.1. Probability Distributions

- $p_{\text{data}}(\mathbf{x})$: Data distribution of real MNIST images
- $p_{\mathbf{z}}(\mathbf{z})$: Prior noise distribution ($\mathcal{N}(0, \mathbf{I})$)

4.1.2. Objective Function

The original GAN objective (binary cross-entropy) is:

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

where:

- $D(\cdot)$ outputs a probability in $[0, 1]$ (realism score)
- $G(\cdot)$ maps noise $\mathbf{z}$ to image space ($\mathbb{R}^{28\times 28}$ for MNIST)

4.1.3. Component Losses

From $\mathcal{L}_{\text{adv}}$, we derive:

1. **Discriminator Loss**:

$$\mathcal{L}_D = -\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[\log D(\mathbf{x})] - \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[\log(1 - D(G(\mathbf{z})))]$$

2. **Generator Loss**:

$$\mathcal{L}_G = \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[\log(1 - D(G(\mathbf{z})))]$$

4.2. Optimization

The training of GANs is highly sensitive to the choice of optimization algorithms. While simple batch gradient descent (GD) was initially experimented with, we adopted the **Adam** optimizer due to its adaptive momentum properties, which empirically demonstrate superior stability in adversarial training. Separate Adam optimizers are used for the generator (G) and discriminator (D) to independently control their learning dynamics.

4.2.1. Adam Update Rules

For each network (with parameters θ), Adam performs the following updates at step t:

Update:

$$\begin{aligned}
m_t &= \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla_\theta \mathcal{L}, \\
v_t &= \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (\nabla_\theta \mathcal{L})^2, \\
\hat{m}_t &= \frac{m_t}{1 - (\beta_1)^t}, \quad \hat{v}_t = \frac{v_t}{1 - (\beta_2)^t}, \\
\theta_t &= \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon},
\end{aligned}$$

where:

- m_t and v_t are estimates of the first and second moments of the gradients,

- $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected versions,
- $\epsilon = 10^{-8}$ is a smoothing term to avoid division by zero.

5 | Training

5.1. Overview

The training process alternates between updating the generator and the discriminator via the Adam optimizer (see Section 4.2). The training follows the adversarial setting of GANs: the discriminator learns to distinguish real from generated data, while the generator learns to produce realistic images to fool the discriminator.

5.2. Training Loop

The training proceeds for a fixed number of epochs. At each epoch, the dataset is shuffled, split in batches of size 64, and a fresh set of latent noise vectors $z \sim \mathcal{N}(0, I)$ is sampled.

Two JAX-jitted functions handle the parameter updates:

- **Discriminator training:** For each batch, real and fake images are fed to the discriminator along with their corresponding labels. The discriminator loss is computed and its parameters are updated using the gradients.
- **Generator training:** The generator takes sampled z vectors and class labels to produce synthetic images. The generator loss is computed by evaluating the discriminator's predictions on these images, and gradients are backpropagated through both models.

Each epoch computes and logs the average generator and discriminator loss values. Every 100 epochs, samples are generated using a fixed set of latent vectors and stored for visualization and evaluation.

5.3. Model Initialization and Checkpointing

If the flag `LOAD_MODEL` is set, pretrained model parameters and optimizer states are loaded from disk using Python's `pickle` module. Otherwise, the model parameters are initialized with Xavier uniform initialization and the optimizer states are freshly created.

Checkpoints are saved in a dedicated directory for each model (since we perform validation for different types of models). This enables training to resume or evaluation to be performed at a later time using the same model state.

5.4. Sample Generation

To qualitatively monitor training, synthetic images are generated every 100 epochs using a fixed noise vector z and class labels $\{0, 1, ..., 9\}$. This enables consistent visual tracking of the generator's progress in learning class-conditional distributions.

6 | Model Evaluation

Evaluating Generative Adversarial Networks (GANs) involves both **qualitative assessment**—to judge the perceptual quality of the outputs—and **quantitative evaluation**—to measure the model's performance using established metrics such as the Fréchet Inception Distance (FID). In this chapter, we present the complete evaluation framework for our GAN model trained on the MNIST dataset.

6.1. Evaluation Methodology

The model is evaluated using a combination of visual inspection, numerical metrics, and training diagnostics:

- **Qualitative Evaluation**: Visual inspection to assess realism and diversity.
- **Quantitative Evaluation**: Employs FID to objectively measure output quality.
- **Training Monitoring**: Tracks losses and gradients to detect issues.

6.2. Qualitative Evaluation

6.2.1. Visual Inspection

Generated digits are plotted alongside real MNIST digits. Evaluation criteria include:

- Image sharpness
- Class diversity
- Mode coverage (all 10 digits)

6.3. Training Dynamics Monitoring

Training diagnostics help identify instabilities during GAN training.

Table 6.1: Training Diagnostics

Metric	**Purpose**
Generator loss $\mathcal{L}_G$	Detects vanishing gradients
Discriminator loss $\mathcal{L}_D$	Checks for overfitting
$D(\mathbf{x})$ vs $D(G(\mathbf{z}))$	Evaluates model balance
Gradient norms	Detects gradient explosion/vanishing

6.4. Quantitative Evaluation

6.4.1. Fréchet Inception Distance (FID)

FID measures similarity between generated and real images:

$$\text{FID} = \|\mu_r - \mu_g\|^2 + \text{Tr}(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2}),$$

- μ_r, μ_g: Mean feature vectors from Inception-v3
- Σ_r, Σ_g: Covariance matrices

Implementation:

- 10,000 real and 10,000 generated images
- Images resized from 28×28 to 299×299
- Calculated using `pytorch-fid`

Lower FID indicates closer resemblance to real data.

6.5. Sample Generation and Visualizations

Every 100 epochs synthetic images are generated using fixed latent vectors and class labels. At the end of the training, for evaluating the model a composite image is created by concatenating the previously generated images which shows a qualitative progress of the generator over the epochs.

6.6. Reproducibility and Checkpointing

The following artifacts are stored and ready to be loaded:

- Generator/discriminator model parameters and optimizer states
- Final evaluation metrics (FID)
- Image grid of generated samples

7 | Results and Analysis

7.1. Experimental Configurations

To identify the optimal GAN setup, we trained four different models, varying the generator and discriminator learning rates, the Adam optimizer parameters (β_1, β_2), and the number of training epochs. For simplicity, the generator and discriminator always shared the same β_1 and β_2 values. Table 7.1 summarizes the hyperparameters and resulting FID scores of each model.

Table 7.1: Summary of Trained Models and Hyperparameters

Model ID	**Epochs**	**Gen LR**	**Disc LR**	$\boldsymbol{\beta_1}$	$\boldsymbol{\beta_2}$	**Validation FID**
M1	1500	1e-4	1.5e-4	0.9	0.999	74.87
M2	2000	5e-5	7e-5	0.7	0.999	74.49
M3	1500	1e-4	1.618e-4	0.5	0.999	69.93
M4	1800[1]	varied[2]	varied[2]	0.5	0.999	**34.74**

As seen in Table 7.1, Model M4 achieved the lowest FID score of 34.74 on the validation set, outperforming all other configurations. This model, which includes a scheduled learning rate decay in its final training phase, was selected for final testing.

7.2. Best Model Analysis (M4)

Model M4, configured with a scheduled learning rate and Adam parameters $\beta_1 = 0.5$, $\beta_2 = 0.999$, produced the most visually convincing and statistically accurate results.

[1]Model M4 was trained for 1600 epochs with fixed learning rates, followed by 200 epochs with decaying learning rates.

[2]Learning rate schedule: epochs 0–1600 with Gen LR $= 1.5 \times 10^{-4}$, Disc LR $= 2.427 \times 10^{-4}$; epochs 1600–1700 with Gen LR $= 5 \times 10^{-5}$, Disc LR $= 1.427 \times 10^{-4}$; epochs 1700–1800 with Gen LR $= 2 \times 10^{-5}$, Disc LR $= 6.45 \times 10^{-5}$.

7.2.1. Training Dynamics

Figure 7.1 shows the generator and discriminator loss curves for Model M4.

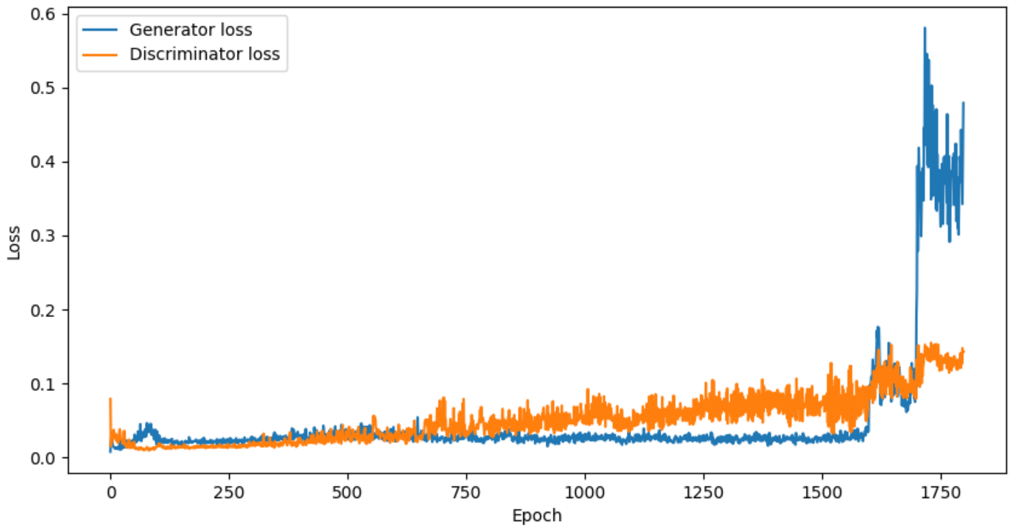

Figure 7.1: Generator and Discriminator Losses for Model M4

7.2.2. Qualitative Results

Figures 7.2 and 7.3 present generated digit samples and a progression grid respectively. The model successfully captured all 10 digit classes with high visual fidelity.

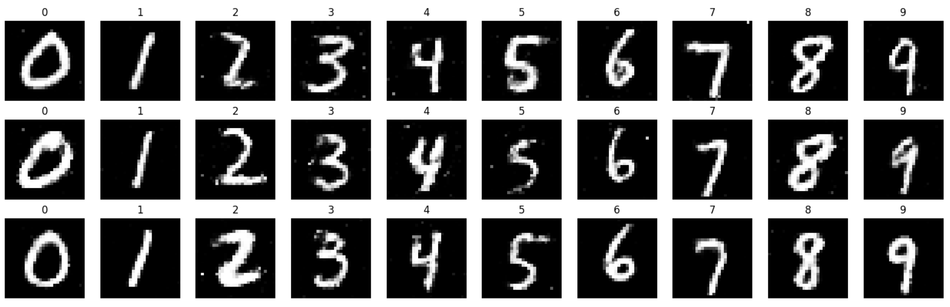

Figure 7.2: Class-Conditional Samples Generated by Model M4

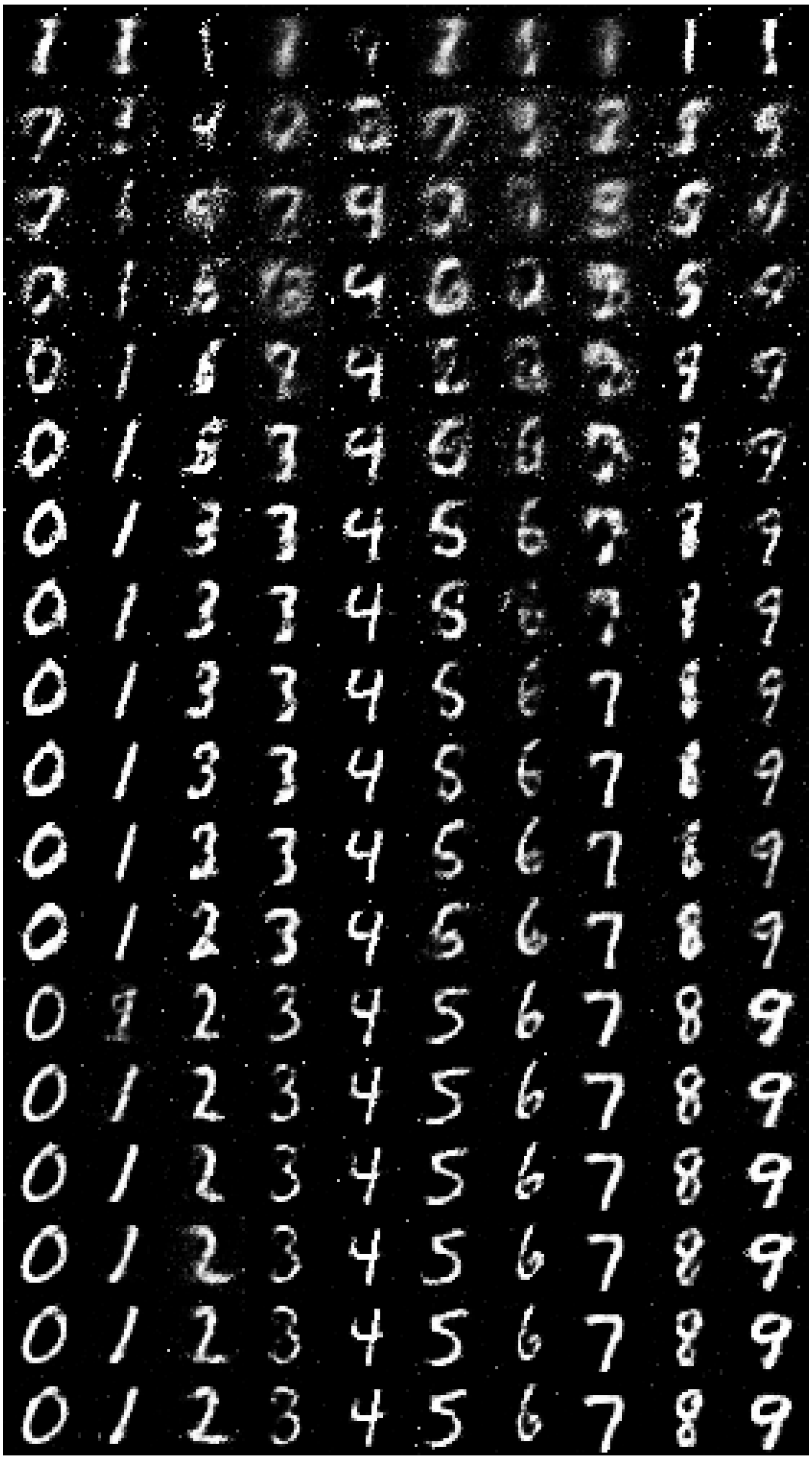
Figure 7.3: Generated Samples Across Training Epochs (Model M4)

7.3. Evaluation on Test Set

Model M4 was evaluated on the MNIST test set using the Fréchet Inception Distance (FID). The resulting test FID score confirmed the model's generalization ability:

```
Test FID: 34.65
```

The identical score on validation and test sets indicates stable and consistent generative performance without significant overfitting.

7.4. Conclusions

7.4.1. Training Difficulties and Challenges

Training generative adversarial networks (GANs) presented several significant challenges:

- **Training Instability**:
  - The generator and discriminator exhibited high sensitivity to hyperparameters, frequently leading to oscillating losses or divergent behavior
  - Small changes in learning rates, optimizer settings, or architecture choices could destabilize the entire training process
- **Loss Interpretation Limitations**:
  - Unlike traditional supervised learning, GAN loss values did not directly correlate with output quality
  - The generator could achieve deceptively low loss values while still producing poor samples if the discriminator failed to provide meaningful gradients
- **Mode Collapse**:
  - Without proper conditioning, the generator frequently collapsed to producing only a limited subset of digits (e.g., predominantly generating "1" or "7")
  - We mitigated this through techniques like label conditioning
- **Hyperparameter Sensitivity**:
  - Achieving balance between generator and discriminator learning rates required extensive experimentation
  - Critical parameters included batch size, learning rates and β_1, β_2

7.4.2. Key Lessons Learned

Our experimentation yielded several important insights:

- **GAN-Specific Findings**:
  - Patience and careful monitoring were essential - we found visual inspection of generated samples more reliable than loss metrics alone
- **Broader Machine Learning Insights**:
  - Small implementation differences (random seeds, library versions) could dramatically affect outcomes
  - Qualitative evaluation complemented quantitative metrics like FID scores

In conclusion, the model we have found works pretty well even if it has some room for improvement regarding the FID score, however the quality of the images is quite good. This project allowed us to better understand the workflow behind building a machine learning model, all the difficulties it imposes and how to appropriately overcome them.

List of Figures

List of Tables