# Explanation of your GAN model, design decisions, training-test dataset descriptions and what other factors were considered to improve your model. *(6 marks)*

Implemented a GAN to generate realistic handwritten digit images based on the MNIST dataset. The model is composed of:

* Generator (G): A 4-layer fully connected neural network that maps a 100-dimensional latent vector to a 28×28 image. We used LeakyReLU activations between layers and applied Batch Normalization after each hidden layer to stabilize training. The output layer uses Tanh to match the expected image normalization range.
* Discriminator (D): A 3-layer neural network that takes a flattened 28×28 image as input and outputs a probability using a Sigmoid activation. It uses LeakyReLU in hidden layers to avoid dying ReLU issues.
* Loss function: Binary Cross-Entropy Loss (BCELoss) to reflect the binary nature of real/fake classification.
* Optimizers: Adam with learning rate = 0.0002 and default betas for both G and D.

Dataset:

* We used train data (from kaggle) which consists of 42,000 grayscale images of digits, each of size 28×28, flattened into 784-length vectors. The "label" column indicates the actual digit class (0–9), and the rest are pixel values (0–255).
* Link - https://www.kaggle.com/competitions/digit-recognizer/data?select=test.csv

Normalization and Preprocessing:

* Real MNIST images (from CSV) were reshaped from flat 784-length arrays into 28×28 grayscale images.
* All real and fake images were normalized to [-1, 1] using transforms.Normalize((0.5,), (0.5,)) to match the output range of the generator (tanh activation).
* Before saving, GAN-generated images were converted from [-1, 1] back to [0, 255] using the formula (img \* 127.5 + 127.5).astype(np.uint8) for proper grayscale visualization and storage.
* During evaluation, the saved fake images (S1) were loaded from disk and re-normalized to [-1, 1] using the same normalization method to ensure compatibility with the classifier input.
* Real MNIST test samples (S0) were also normalized to [-1, 1] using transforms.ToTensor() followed by the same transforms.Normalize((0.5,), (0.5,)), ensuring input consistency across both datasets (S0 and S1).

Training Improvements:

The Generator was trained for 50 epochs, and we visually inspected outputs across epochs. We noticed that digit quality became significantly clearer after nearly 30 epochs, so we chose to stop at 50 to balance training time and quality.

During early training, generated digits were often blurry or malformed. To address this:

* increased batch size to 128 to stabilize gradient updates.
* applied LeakyReLU instead of ReLU in the Generator and Discriminator to avoid dead neurons.
* fine-tuned the learning rate (0.0002) for both Generator and Discriminator after initial instability with higher values.

After generating 500 images, we manually selected the best 100, ensuring each digit was clearly human-recognizable and represented a balanced class distribution.

Selected fake samples were normalized using the same transformation pipeline as the real MNIST images, ensuring compatibility with the classifier’s expected input range.

# Training accuracy of the GAN model at the end of each epoch. (2 marks)

training progress is monitored using the Generator Loss and Discriminator Loss, which reflect how well each model is performing its role in the adversarial setting.

These values show that: The Generator is improving over time by reducing its loss and making more realistic samples.

The Discriminator loss varies due to the adversarial dynamic — when the Generator improves, the Discriminator temporarily struggles, as expected.

# Explanation of your Classifier model, design decisions, training-test dataset descriptions and what other factors were considered to improve your model. (6 marks)

A CNN classifier is designed to distinguish between handwritten digits in the MNIST dataset.

Model Architecture:

* Input: 1-channel (grayscale) 28×28 images
* Conv Layer 1: 32 filters, kernel size 3×3, followed by ReLU and max-pooling (14×14)
* Conv Layer 2: 64 filters, kernel size 3×3, followed by ReLU and max-pooling (7×7)
* Fully Connected Layers:  
  + Flattened to 64×7×7 = 3136 features
  + First dense layer with 128 units (ReLU)
  + Output layer with 10 units (for 10 digit classes), using softmax (via CrossEntropyLoss)

Training and Dataset:

* Trained on the official MNIST training set (60,000 images)
* Tested on the standard MNIST test set (10,000 images)
* Inputs were normalized to [-1, 1] using transforms.Normalize((0.5,), (0.5,)) for compatibility with the GAN-generated images

Design Decisions:

* We used ReLU activations and MaxPooling to retain important features while reducing dimensionality.
* Adam optimizer was chosen for fast convergence with a learning rate of 0.001.
* Training was run for 5 epochs, which provided near-optimal performance without overfitting.
* CrossEntropyLoss was used as the loss function, standard for multi-class classification tasks.

Additional Improvements:

* Model was evaluated not only on the full MNIST test set, but also on:  
  + S0: 100 real samples randomly selected from MNIST test set
  + S1: 100 fake samples generated by our GAN and manually labeled
* The classifier was saved as C.pkl for reproducibility and later evaluation
* Input normalization was kept consistent across real and fake datasets to avoid domain shift

# Training accuracy of the Classifier model at the end of each epoch. (2 marks)

Epoch 1/5 - Loss: 142.9428 - Accuracy: 95.36%

Epoch 2/5 - Loss: 40.9160 - Accuracy: 98.67%

Epoch 3/5 - Loss: 28.1035 - Accuracy: 99.04%

Epoch 4/5 - Loss: 21.0026 - Accuracy: 99.27%

Epoch 5/5 - Loss: 15.6487 - Accuracy: 99.44%

# Testing accuracy of the Classifier model. (For the whole MNIST test data set. One number at 2 decimal points) (2 marks)

98.98%

# Report the classification error for test set *S0* . One number at 2 decimal points. (3 marks)

0.00%

# Report the classification error for test set *S1. One number at 2 decimal points.* (3 marks)

18.00%

# Discuss your observations. (6 marks)

The Classifier trained on real MNIST data achieved a training accuracy of 99.44% and a testing accuracy of 98.98%, demonstrating excellent generalization.

On the S0 test set (100 real MNIST digits), the classifier correctly classified all samples, resulting in a 0.00% error rate. The classifier is highly reliable when evaluating real handwritten digits from the MNIST dataset. It perfectly generalizes to unseen real data, confirming that it learned meaningful and accurate features during training.

On the S1 test set (100 GAN-generated digits), the classifier misclassified 18 samples, giving an error rate of 18.00%. This shows that some generated digits were not clear or realistic enough to be classified correctly.

The difference in performance between S0 and S1 highlights the difficulty of generating fake digits that are both visually realistic and accurately match the patterns learned by the classifier.

Additionally, the errors in S1 often came from digits that lacked strong structure, appeared blurred, or showed features that overlapped between multiple digit classes (ex: 2 and 7). This reveals that while the GAN learns to approximate the MNIST distribution, it still struggles with fine-grained details needed for perfect classification.

# Instead of generating random digits, if I asked you to generate specific digits (e.g., generate an image of 0 when the integer 0 is given as input) how would you do it? You have to use the Generator that you have already created. Describe your architecture along with a diagram and explain how you train and test it.

To generate specific digits using our existing Generator, we need to build a separate neural network that takes a digit label (like 0, 1, 2…) and outputs a matching latent vector z. This latent vector is then passed to the Generator to produce the desired digit image.

This new neural network is called the Latent Code Predictor. It takes a one-hot encoded digit label (a vector of size 10 where only the target digit is 1, and the rest are 0s) and outputs a 100-dimensional latent vector. This output is fed into the Generator to produce a digit image.

We do not retrain the Generator. Instead, we freeze it and only train the Latent Code Predictor. To train this predictor, we:

* Generate many latent vectors and use them with the Generator to produce images.
* Pass these images through the trained Classifier to get predicted labels.
* Keep only the (label, latent vector) pairs where the Classifier gives the correct label.
* Train the Latent Code Predictor to learn the mapping from label → latent vector using those correct pairs.

After training, to generate a specific digit (say, 3), we just give the one-hot encoded label for 3 to the Latent Code Predictor. It gives us a latent vector, which we pass into the Generator to get an image of the digit 3.

This method gives us control over what digit we want the Generator to produce, without changing the Generator itself.

# 



