



**Faculty of Engineering – Cairo University**  
**Electronics And Electrical Communications Department**  
**Fourth year - Mainstream**  
**ELC4028 Neural Networks – Assignment 3**

**Submitted by:**

ID	Sec	Name
9213327	3	محمد أيمن فاروق سيد عبد الغفار
9211027	3	محمد علاء الدين عطفت مصطفى محمد رستم
9211039	4	محمد مجدي مبروك ندا خير
9213468	4	يوسف تامر صلاح الدين السيد
9211418	4	يوسف عصام أبوبكر محمد

**Submitted to:**

Prof. Mohsen Rashwan

# Problem 1: Autoencoder-Based MFCC Representation and Classification of Spoken Digit Utterances

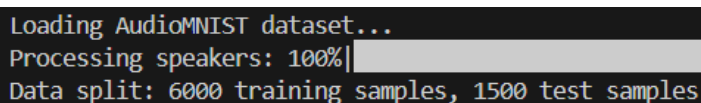
## 1. Introduction

Using Mel Frequency Cepstral Coefficients (MFCCs) as the core feature representation, we explore a baseline method and several autoencoder-based strategies to convert variable-length speech signals into fixed-size vectors suitable for classification. We aim to represent spoken digit utterances (digits 0 through 9) as fixed-length vectors focusing on data compression using autoencoders to generate compact and consistent feature representations from sequences of MFCCs, without applying data augmentation.

## 2. Data Preprocessing and Feature Extraction

The dataset comprises audio recordings of digits 0 through 9, spoken by multiple speakers. Each recording undergoes the following preprocessing steps:

- Audio signals are divided into 15 ms overlapping frames with 10 ms hop-length.
- For each frame, 13-dimensional MFCCs are computed.
- Since utterances vary in length, each sequence is padded with zero-valued frames to match the maximum number of frames in any utterance.
- The resulting input per utterance is a matrix of shape (max\_frames, 13).



```
Loading AudioMNIST dataset...  
Processing speakers: 100%  
Data split: 6000 training samples, 1500 test samples
```

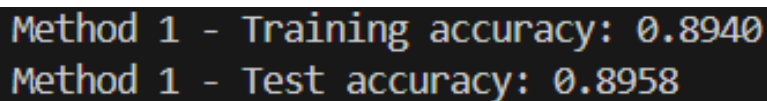
Figure 1: Loading data

## 3. Baseline Method: Average Frame MFCC Representation

The baseline approach represents each utterance by computing the average of its MFCC vectors across all frames, resulting in a single 13-dimensional vector per sample. This simple method ignores temporal dynamics but provides a quick way to convert sequences into fixed-length vectors. A classifier is trained on these representations.

**Baseline Training Accuracy: 89.4 %**

**Baseline Test Accuracy: 89.58 %**



```
Method 1 - Training accuracy: 0.8940  
Method 1 - Test accuracy: 0.8958
```

Figure 2: Baseline Accuracy

## 4. Autoencoder-Based Representations

### 4.1 Method A: Concatenated Frames with Zero Padding

Each utterance's MFCC frames are flattened into a single long vector (concatenated row-wise) and padded with zeros to match a fixed total length. An autoencoder is trained on these vectors to reconstruct the original padded sequences.

**Method A Training Accuracy: 98.44%**

**Method A Test Accuracy: 97.57%**

---

```
Running Method 2: Autoencoder Method A (Flattened Utterance)
Training Autoencoder A...
Epoch 10/100, Loss: 0.079021
Epoch 20/100, Loss: 0.071075
Epoch 30/100, Loss: 0.067368
Epoch 40/100, Loss: 0.065101
Epoch 50/100, Loss: 0.063888
Epoch 60/100, Loss: 0.062664
Epoch 70/100, Loss: 0.061880
Epoch 80/100, Loss: 0.061120
Epoch 90/100, Loss: 0.060591
Epoch 100/100, Loss: 0.060020
Encoding training data: 100%|
Encoding test data: 100%|
Method 2 - Training accuracy: 0.9844
Method 2 - Test accuracy: 0.9757
```

Figure 3: Method A Accuracy

#### 4.2 Method B: Frame-wise Sequential Compression

This method starts by encoding the first two MFCC frames using an autoencoder. The resulting compressed vector is concatenated with the third frame and re-encoded. This process continues sequentially across all frames. The final output of the autoencoder serves as the utterance representation.

**Method B Training Accuracy: 64.12%**

**Method B Test Accuracy: 44.5%**

---

```
Running Method 3: Autoencoder Method B (Sequential Compression)
Training Autoencoder B...
Epoch 10/100, Loss: 0.020184
Epoch 20/100, Loss: 0.013671
Epoch 30/100, Loss: 0.011980
Epoch 40/100, Loss: 0.011704
Epoch 50/100, Loss: 0.011388
Epoch 60/100, Loss: 0.011255
Epoch 70/100, Loss: 0.011203
Epoch 80/100, Loss: 0.011108
Epoch 90/100, Loss: 0.011001
Epoch 100/100, Loss: 0.010949
Encoding training data: 100%|
Encoding test data: 100%|
Method 3 - Training accuracy: 0.6412
Method 3 - Test accuracy: 0.4450
```

Figure 4: Method B Accuracy

### 4.3 Method C: Optimal Frame Pairing

Train an autoencoder that compresses pairs of MFCC frames into a single MFCC-sized vector. For each utterance, implement a greedy compression strategy. Calculate reconstruction error for all adjacent frame pairs. Select and compress the pair with the lowest reconstruction error. Replace this pair with its compressed representation. Recalculate errors with the new compressed frame and continue. Repeat until only one vector remains, capturing the entire utterance. Use this final compressed vector for classification.

*Method C Training Accuracy: 72%*

*Method C Test Accuracy: 63%*

```
Running Method 4: Autoencoder Method C (Greedy Compression)
Encoding training data: 100%|
Encoding test data: 100%|
Method 4 - Training accuracy: 0.7200
Method 4 - Test accuracy: 0.6300
```

*Figure 5: Method C Accuracy*

## 5. Evaluation

Evaluation is performed using classification accuracy on a held-out test set.

Method	Description	Accuracy (%)
Baseline	Average of MFCC frames	89.58
Method A	AE on concatenated/padded MFCC frame vectors	97.57
Method B	Stepwise AE compression of MFCC frame sequence	44.5
Method C	Optimized AE compression via lowest error pairs	63

## 6. Discussion

The experimental results highlight significant differences in performance across the evaluated methods, with notable implications for representation learning in speech processing.

The **baseline method**, which averages MFCC frames per utterance, achieved a strong classification accuracy of 89.58%. This suggests that even a simple statistical summary of MFCC features can capture sufficient phonetic characteristics for distinguishing between spoken digits.

**Method A**, which utilizes an autoencoder trained on zero-padded, concatenated MFCC sequences, achieved the highest performance with an accuracy of **97.57%**. This substantial improvement over the baseline illustrates the strength of using learned representations via autoencoders. By compressing high-dimensional MFCC sequences into fixed-length latent vectors, the autoencoder captures richer and more discriminative features than simple averaging, despite the presence of zero-padding.

Surprisingly, **Method B**, which involved stepwise sequential compression of MFCC frames, resulted in a **low accuracy of 44.5%**. This poor performance may be attributed to the accumulation of reconstruction errors at each compression step, potentially leading to the loss of important global features of the utterance. Additionally, combining encoded outputs one frame at a time could have caused the autoencoder to focus too heavily on local frame transitions, rather than capturing the broader structure of the entire sequence.

**Method C**, which aimed to optimize the compression path by greedily selecting frame pairings with the lowest reconstruction error, achieved a moderate improvement over Method B, reaching an accuracy of **63%**. While this approach is designed to retain more informative transitions between frames, its greedy nature might have caused it to converge to suboptimal global representations. Moreover, its higher computational complexity did not translate into a significant performance benefit.

In summary, even though Methods B and C were designed to better understand the time-related patterns in speech, they did not perform as well as the baseline or Method A. This shows that it's important to keep a good balance between learning detailed features and keeping the model simple and stable during training.

## Problem 2: Data Augmentation

### Program Flow:

This Python program implements a pipeline to train and evaluate a modified LeNet-5 convolutional neural network on a reduced MNIST dataset. It starts by checking for a saved dataset file ([mnist\\_reduced.npz](#)). If not found, it downloads the original MNIST dataset, normalizes the pixel values, and selects 1000 training and 200 testing samples per class (digit) to create a balanced and reduced dataset. This reduced dataset is saved for future use.

Data augmentation is then applied to increase the diversity of the training data. Each original training image is augmented three times using random transformations like rotations, translations, and resized crops. These augmented samples, along with the originals, are converted into tensors and grouped using PyTorch's [TensorDataset](#) and [DataLoader](#) classes. A modified LeNet-5 model is defined, featuring two convolutional layers with average pooling and a fully connected output layer suitable for classifying 28x28 grayscale images into 10 classes.

The program then evaluates 12 different training scenarios by varying the number of real and augmented samples per class. The tested combinations are:

- Real samples: 300, 700, and 1000 per class
- Augmented samples: 0, 1000, 2000, and 3000 per class

For each of these 12 ( $3 \times 4$ ) combinations, the training and test datasets are assembled accordingly, and the model is trained from scratch for 10 epochs. After training, both training and test accuracies are reported. Finally, the test accuracies across all scenarios are summarized in a neatly formatted results table. This setup allows for analyzing how data augmentation and varying real sample sizes impact model performance.

### Simulation Results:

```
Training on dataset with 300 real samples and 1000 augm
ented samples...
Epoch 1, Loss: 1.1224, Acc: 63.26%
Epoch 2, Loss: 0.5210, Acc: 83.32%
Epoch 3, Loss: 0.3845, Acc: 87.68%
Epoch 4, Loss: 0.3062, Acc: 90.06%
Epoch 5, Loss: 0.2619, Acc: 91.21%
Epoch 6, Loss: 0.2326, Acc: 92.30%
Epoch 7, Loss: 0.2238, Acc: 92.58%
Epoch 8, Loss: 0.1676, Acc: 94.23%
Epoch 9, Loss: 0.1492, Acc: 95.02%
Epoch 10, Loss: 0.1419, Acc: 95.31%
Final Train Acc: 95.99%
Time taken to train LeNet-5: 8.436 seconds
Final Test Acc: 93.60%
Time taken to test LeNet-5: 0.043 seconds
```

Figure 6: Results of training on dataset with 300 real samples and 1000 augmented samples

```
Training on dataset with 300 real samples and 2000 augmented samples...
Epoch 1, Loss: 0.8693, Acc: 71.57%
Epoch 2, Loss: 0.4140, Acc: 86.77%
Epoch 3, Loss: 0.3208, Acc: 89.73%
Epoch 4, Loss: 0.2629, Acc: 91.56%
Epoch 5, Loss: 0.2253, Acc: 92.57%
Epoch 6, Loss: 0.2039, Acc: 93.28%
Epoch 7, Loss: 0.1809, Acc: 93.99%
Epoch 8, Loss: 0.1590, Acc: 94.65%
Epoch 9, Loss: 0.1424, Acc: 95.23%
Epoch 10, Loss: 0.1303, Acc: 95.50%
Final Train Acc: 96.71%
Time taken to train LeNet-5: 14.402000000000001 seconds
Final Test Acc: 95.55%
Time taken to test LeNet-5: 0.040999999999999995 seconds
```

Figure 7: Results of training on dataset with 300 real samples and 2000 augmented samples

```
Training on dataset with 700 real samples and 1000 augmented samples...
Epoch 1, Loss: 0.9054, Acc: 71.09%
Epoch 2, Loss: 0.3995, Acc: 87.28%
Epoch 3, Loss: 0.2993, Acc: 90.36%
Epoch 4, Loss: 0.2450, Acc: 92.09%
Epoch 5, Loss: 0.2076, Acc: 93.36%
Epoch 6, Loss: 0.1820, Acc: 93.94%
Epoch 7, Loss: 0.1581, Acc: 94.71%
Epoch 8, Loss: 0.1398, Acc: 95.38%
Epoch 9, Loss: 0.1169, Acc: 95.94%
Epoch 10, Loss: 0.1164, Acc: 95.93%
Final Train Acc: 97.36%
Time taken to train LeNet-5: 10.523 seconds
Final Test Acc: 95.25%
Time taken to test LeNet-5: 0.043 seconds
```

Figure 8: Results of training on dataset with 700 real samples and 1000 augmented samples

```

Training on dataset with 700 real samples and 2000 augmented samples...
Epoch 1, Loss: 0.7848, Acc: 74.98%
Epoch 2, Loss: 0.3749, Acc: 87.95%
Epoch 3, Loss: 0.2801, Acc: 90.90%
Epoch 4, Loss: 0.2306, Acc: 92.55%
Epoch 5, Loss: 0.1992, Acc: 93.62%
Epoch 6, Loss: 0.1742, Acc: 94.27%
Epoch 7, Loss: 0.1490, Acc: 95.11%
Epoch 8, Loss: 0.1379, Acc: 95.33%
Epoch 9, Loss: 0.1276, Acc: 95.70%
Epoch 10, Loss: 0.1128, Acc: 96.10%
Final Train Acc: 96.76%
Time taken to train LeNet-5: 16.639 seconds
Final Test Acc: 95.75%
Time taken to test LeNet-5: 0.042 seconds

```

Figure 9: Results of training on dataset with 700 real samples and 2000 augmented samples

Final Accuracy Table:				
Aug\Real	300	700	1000	
0	92.75%	95.05%	95.25%	
1000	93.60%	95.25%	95.85%	
2000	95.55%	95.75%	95.70%	
3000	95.25%	95.60%	95.85%	

Figure 10: Final Accuracy Table for different number of real and augmented samples



Final Accuracy Table:

		Real Data		
		300	700	1,000
No. of Augmented examples	0	92.75%	95.05%	95.25%
	1,000	93.60%	95.25%	95.85%
	2,000	95.55%	95.75%	95.70%
	3,000	95.25%	95.60%	95.85%

Comments and conclusion:

The table presents classification accuracy across different combinations of real and augmented training samples. As expected, increasing the amount of real data generally leads to improved performance, with the model achieving **95.25% accuracy** using **1000 real samples** and no augmentation. However, the key insight comes from the impact of augmentation: for all real-data sizes, introducing generated (augmented) data results in **improved or at least maintained accuracy**.

In the case of only **300 real samples**, performance improved significantly with augmentation from **92.75% to a peak of 95.55%** with 2000 augmented examples. This highlights the value of augmentation in low-data regimes, where it effectively boosts model generalization by providing diverse training variations. For larger real datasets (700 and 1000), the improvement from augmentation is smaller but still meaningful, pushing accuracy slightly higher or helping it plateau more consistently.

Interestingly, beyond 2000 generated examples, the benefit of adding more data diminishes or stabilizes, suggesting a **saturation point** where more data no longer yields significant gains. This diminishing return is particularly evident for the 700 and 1000 real-sample settings.

A deeper look at the training output for one case—**300 real + 1000 augmented samples**—shows the model learning effectively, improving from **63.26% to 95.31%** training accuracy over 10 epochs, with a final training accuracy of **95.99%**. However, the **test accuracy was slightly lower at 93.60%**, indicating **mild overfitting**, a common occurrence when models adapt strongly to training data. Nevertheless, the training time was short (~8.4 seconds), showcasing the efficiency of this setup for rapid experimentation.

In conclusion, **data augmentation proves most effective when real data is limited**, significantly narrowing the performance gap between small and large datasets. While its benefits plateau at scale, these results confirm that **strategic augmentation is a practical and efficient method for improving model robustness**, especially in data-scarce environments.

## Problem 3: Using GAN to generate Synthetic Data

The two networks presented — the Generator and the Discriminator — form the core components of a Generative Adversarial Network (GAN), where the generator creates data and the discriminator evaluates it. Both are implemented using PyTorch and are designed to work with grayscale images of size  $28 \times 28$ , such as those in the MNIST dataset.

The Discriminator is a binary classifier that aims to distinguish between real and fake images. It takes a  $28 \times 28 \times 1$  image as input and processes it through two convolutional layers. The first layer transforms the input into 64 feature maps using a  $5 \times 5$  kernel with stride 2 and padding 2, followed by a LeakyReLU activation and dropout to prevent overfitting. The second convolutional layer increases the depth to 128 using the same kernel settings, followed again by LeakyReLU and dropout. The resulting  $7 \times 7 \times 128$  feature map is flattened into a vector of length 6272, which is passed through a linear layer to produce a single output, indicating the probability of the image being real. The Sigmoid activation function squashes the output between 0 and 1.

The Generator performs the inverse operation. It starts from a 100-dimensional latent vector (random noise) and aims to generate a realistic image. The input is passed through a fully connected layer that expands it to a  $7 \times 7 \times 256$  tensor. This tensor is reshaped and processed through three transposed convolutional layers, gradually increasing spatial dimensions while reducing the depth. The first transposed convolution maintains the  $7 \times 7$  size while reducing the depth to 128. The second increases the size to  $14 \times 14$  and reduces depth to 64. The final transposed convolution outputs a  $28 \times 28 \times 1$  image. Batch normalization and LeakyReLU are used after each layer except the last, which uses a Tanh activation to output pixel values in the range  $[-1, 1]$ . The final output is normalized to  $[0, 1]$ .

Together, these networks engage in a competitive process: the generator improves to produce more convincing images, while the discriminator sharpens its ability to distinguish real from fake.

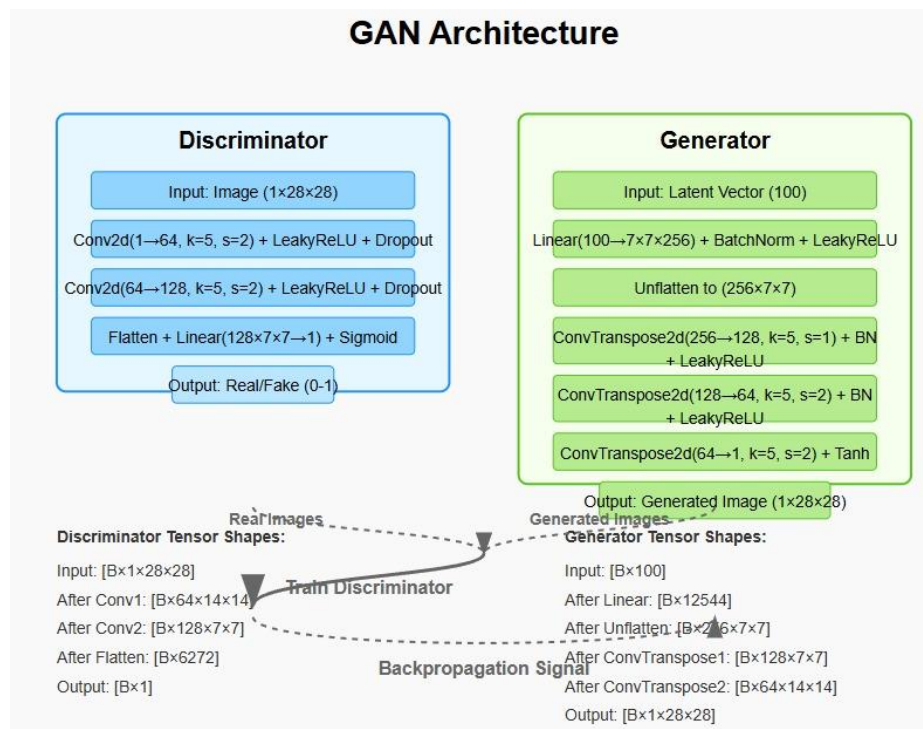


Figure 11: Generator and discriminator arch.

## Program Flow:

The program performs a comprehensive evaluation of how synthetic data generated using a GAN impacts classification accuracy on a reduced MNIST dataset using a modified LeNet-5 convolutional neural network. Initially, it loads or creates a reduced MNIST dataset consisting of 1000 training and 200 testing samples per digit. To simulate different training conditions, the program explores 12 combinations of real and generated data, where real samples per digit (**size**) are chosen from {300, 700, 1000} and generated samples per digit (**generate**) are selected from {0, 1000, 2000, 3000}. The synthetic datasets, previously generated using a GAN, are stored in nine **.npz** files named following the pattern "gen\_image\_{size}\_{generate}.npz".

For each of the 12 scenarios, the program randomly selects **size** real samples per digit and, if **generate > 0**, loads the corresponding synthetic dataset. It then merges the real and synthetic samples to form an augmented training set. Both the training and test data are converted into PyTorch tensors and loaded into data loaders. A new instance of LeNet-5 is initialized for each run, trained for 10 epochs using the training loader, and evaluated on the testing set. Training and testing accuracies, along with the time taken, are printed after each run.

Finally, the script compiles and displays a results table summarizing the classification accuracy for each of the 12 training configurations. This flow allows a clear comparative analysis of how the quantity of real and synthetic data influences model performance, ultimately helping to assess the efficacy of GAN-generated data in improving digit classification tasks.

### *Some Snippets of the generated digits:*



Figure 12: Snippets of generated digits from 300 real data

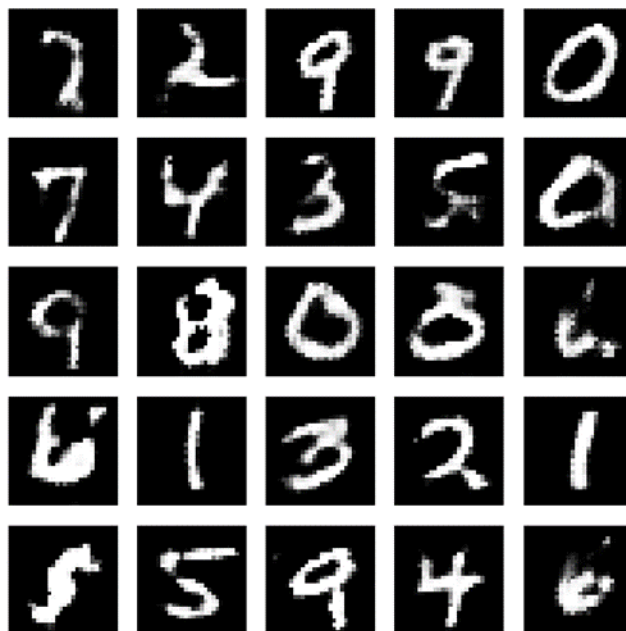


Figure 13: Snippets of generated digits from 700 real data

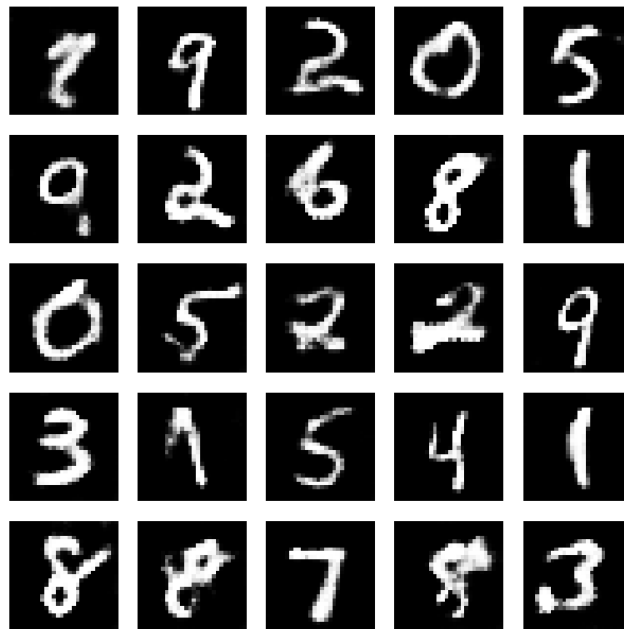


Figure 14: Snippets of generated digits from 1000 real data

### Simulation Results:

```
Training on dataset with 300 real samples and 1000 generated samples per digit...
Epoch 1, Loss: 0.5052, Acc: 84.12%
Epoch 2, Loss: 0.1474, Acc: 95.35%
Epoch 3, Loss: 0.1060, Acc: 96.69%
Epoch 4, Loss: 0.0780, Acc: 97.77%
Epoch 5, Loss: 0.0632, Acc: 97.85%
Epoch 6, Loss: 0.0518, Acc: 98.39%
Epoch 7, Loss: 0.0516, Acc: 98.27%
Epoch 8, Loss: 0.0368, Acc: 98.81%
Epoch 9, Loss: 0.0340, Acc: 98.80%
Epoch 10, Loss: 0.0343, Acc: 98.83%
Final Train Acc: 99.08%
Time taken to train LeNet-5: 8.3 seconds
Final Test Acc: 91.50%
Time taken to test LeNet-5: 0.051 seconds
```

Figure 15: Results of training on dataset with 300 real samples and 1000 generated samples

```
Training on dataset with 300 real samples and 2000 generated samples per digit...
Epoch 1, Loss: 0.3499, Acc: 88.87%
Epoch 2, Loss: 0.0947, Acc: 97.14%
Epoch 3, Loss: 0.0663, Acc: 97.93%
Epoch 4, Loss: 0.0538, Acc: 98.25%
Epoch 5, Loss: 0.0425, Acc: 98.62%
Epoch 6, Loss: 0.0339, Acc: 98.87%
Epoch 7, Loss: 0.0297, Acc: 99.01%
Epoch 8, Loss: 0.0247, Acc: 99.19%
Epoch 9, Loss: 0.0257, Acc: 99.20%
Epoch 10, Loss: 0.0238, Acc: 99.22%
Final Train Acc: 99.40%
Time taken to train LeNet-5: 14.905 seconds
Final Test Acc: 91.20%
Time taken to test LeNet-5: 0.044000000000000004 seconds
```

*Figure 16: Results of training on dataset with 300 real samples and 2000 generated samples*

```
Training on dataset with 700 real samples and 1000 generated samples per digit...
Epoch 1, Loss: 0.4728, Acc: 84.54%
Epoch 2, Loss: 0.1749, Acc: 94.31%
Epoch 3, Loss: 0.1230, Acc: 95.99%
Epoch 4, Loss: 0.0989, Acc: 96.67%
Epoch 5, Loss: 0.0751, Acc: 97.45%
Epoch 6, Loss: 0.0712, Acc: 97.58%
Epoch 7, Loss: 0.0572, Acc: 98.09%
Epoch 8, Loss: 0.0505, Acc: 98.24%
Epoch 9, Loss: 0.0469, Acc: 98.41%
Epoch 10, Loss: 0.0390, Acc: 98.65%
Final Train Acc: 98.54%
Time taken to train LeNet-5: 11.311 seconds
Final Test Acc: 94.25%
Time taken to test LeNet-5: 0.057 seconds
```

*Figure 17: Results of training on dataset with 700 real samples and 1000 generated samples*

```

Training on dataset with 700 real samples and 2000 generated samples per digit...
Epoch 1, Loss: 0.3695, Acc: 87.99%
Epoch 2, Loss: 0.1315, Acc: 95.60%
Epoch 3, Loss: 0.0917, Acc: 97.09%
Epoch 4, Loss: 0.0741, Acc: 97.60%
Epoch 5, Loss: 0.0607, Acc: 98.03%
Epoch 6, Loss: 0.0494, Acc: 98.36%
Epoch 7, Loss: 0.0443, Acc: 98.51%
Epoch 8, Loss: 0.0356, Acc: 98.81%
Epoch 9, Loss: 0.0355, Acc: 98.74%
Epoch 10, Loss: 0.0316, Acc: 98.85%
Final Train Acc: 99.14%
Time taken to train LeNet-5: 17.752 seconds
Final Test Acc: 94.55%
Time taken to test LeNet-5: 0.048 seconds

```

Figure 18: Results of training on dataset with 700 real samples and 2000 generated samples

Final Accuracy Table:				
Gen\Real	300	700	1000	
0	92.20%	95.00%	95.55%	
1000	91.50%	94.25%	94.50%	
2000	91.20%	94.55%	94.95%	
3000	91.20%	94.50%	94.95%	

Figure 19: Final Accuracy Table for different number of real and generated samples

Final Accuracy Table:

		Real Data		
		300	700	1,000
No. of Generated examples	0	92.20%	95.00%	95.55%
	1,000	91.50%	94.25%	94.50%
	2,000	91.20%	94.55%	94.95%
	3,000	91.20%	94.50%	94.95%

Comments and conclusion:

The accuracy results across various combinations of real and GAN-generated training data reveal key insights into the practical effectiveness of synthetic data. When using only real data, the model performs well—achieving up to 95.55% accuracy with 1000 real samples. However, unlike augmentation-based data, the addition of GAN-generated samples generally led to a **drop in performance**.

For all real data sizes (300, 700, and 1000), increasing the amount of GAN-generated data caused a slight but consistent decrease in accuracy. For instance, with 1000 real samples, performance dropped from 95.55% to 94.50% and plateaued, regardless of whether 1000, 2000, or 3000 generated samples were added. This trend suggests that the GAN-generated digits may not be diverse or realistic enough to enhance the model’s generalization — and might even introduce noisy or confusing patterns that hinder learning.

Furthermore, the results show **diminishing returns**: once 1000 GAN samples are added, adding more (2000 or 3000) does not yield further improvement, which reinforces the idea that **data quality** matters more than quantity. The fact that accuracy is consistently lower than the augmentation results also underscores the importance of **controlling the quality and variability** of synthetic data during GAN training.

**In conclusion**, while GANs hold promise for data generation, their practical impact in this task was limited — and potentially harmful — compared to traditional augmentation. The findings highlight that **GAN-generated data should be carefully validated** before being used to replace or supplement real data in training deep learning models.

## Reducing the Need for Real Data: Augmentation and GAN Synthesis at 300 Real Samples

```
Evaluating with 1000 real samples only...
Epoch 1, Loss: 0.0653, Acc: 97.97%
Epoch 2, Loss: 0.0382, Acc: 98.76%
Epoch 3, Loss: 0.0204, Acc: 99.36%
Epoch 4, Loss: 0.0142, Acc: 99.50%
Epoch 5, Loss: 0.0085, Acc: 99.73%
Epoch 6, Loss: 0.0040, Acc: 99.92%
Epoch 7, Loss: 0.0202, Acc: 99.36%
Epoch 8, Loss: 0.0251, Acc: 99.11%
Epoch 9, Loss: 0.0270, Acc: 99.14%
Epoch 10, Loss: 0.0324, Acc: 99.07%
Final Train Acc (1000 real): 99.74%
Time taken to train LeNet-5 (1000 real): 6.364 seconds
Final Test Acc (1000 real): 96.45%
Time taken to test LeNet-5 (1000 real): 0.045 seconds
```

Figure 20: Results of training on dataset with 1000 real samples and 0 generated samples and 0 augmented samples

Final Accuracy Table:					
Gen\Aug	0	1000	2000	3000	
0	92.25%	94.45%	95.55%	95.90%	
1000	91.80%	94.20%	94.05%	94.95%	
2000	91.35%	94.15%	94.95%	95.80%	
3000	89.65%	94.60%	94.05%	95.45%	

Figure 21: Final Accuracy Table for different number of augmented and generated samples used with 300 real samples



## Comments and conclusion:

This analysis explores the extent to which data augmentation and GAN-generated data can reduce the need for large real datasets by evaluating different combinations of synthetic and augmented samples while keeping the number of real samples fixed at **300**. The goal is to determine whether synthetic techniques can achieve comparable performance to using the full **1000 real samples**, which yielded a test accuracy of **96.45%**.

From the results table, we observe that using **only 300 real samples** (with no augmentation or GANs) achieves **92.25% accuracy**, which is noticeably lower than the 1000-real baseline. However, once augmented data is introduced, accuracy improves substantially. For instance, combining 300 real samples with **3000 augmented examples** (and no GAN data) achieves **95.90%**, just **0.55% below** the 1000-real baseline. This shows that carefully constructed augmented data can nearly replicate the benefit of having over three times more real data.

When comparing this to combinations that include **GAN-generated data**, the results are more mixed. For example, using **300 real + 3000 augmented + 2000 GAN** samples reaches **95.80%**, very close to the augmentation-only peak. However, other combinations with heavy reliance on GAN data tend to perform worse, especially when GAN data replaces augmentation. For example, 300 real + 3000 GAN (without augmentation) results in just **89.65%** accuracy, indicating that GAN data on its own does not effectively substitute for real or augmented data.

The best-performing setup that includes both augmentation and GANs is **300 real + 3000 augmented + 2000 GAN**, with an accuracy of **95.80%**—only **0.65% below** the 1000-real case. This shows that when augmentation is the primary method of data expansion, GANs can contribute positively as a supplementary source.

## Conclusion

We conclude that **the need for a full real dataset can be significantly reduced**. With just **300 real samples**, the model can nearly match the accuracy of training on 1000 real samples by leveraging **data augmentation alone**, or in combination with moderate amounts of **GAN-generated data**. The most effective strategy remains heavy augmentation, with GAN data offering marginal improvements when added cautiously. This confirms that in scenarios where real data is limited or expensive to collect, a synthetic data strategy—especially one focused on augmentation—can be a powerful and efficient alternative.

## Problem 4: Understanding the Impact of Attention Mechanisms (Part a)

Build basic CNN network to classify images from the ReducedMNIST dataset.

- Three convolutional layers (32, 64, and 128 filters)
- MaxPooling after the first two conv layers
- Fully Connected Layer
- Dropout for regularization

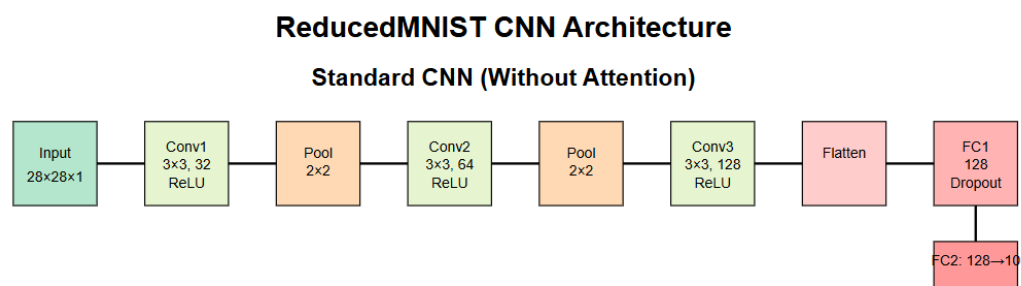


Figure 22: Block Diagram for the Basic CNN network

```
Training Basic CNN...
Epoch 1/10
71/71 ██████████ 5s 40ms/step - accuracy: 0.5469 - loss: 1.3292 - val_accuracy: 0.0000e+00 - val_loss: 11.9169
Epoch 2/10
71/71 ██████████ 3s 35ms/step - accuracy: 0.9322 - loss: 0.2275 - val_accuracy: 0.0000e+00 - val_loss: 13.1923
Epoch 3/10
71/71 ██████████ 3s 35ms/step - accuracy: 0.9610 - loss: 0.1259 - val_accuracy: 0.0000e+00 - val_loss: 13.3940
Epoch 4/10
71/71 ██████████ 3s 36ms/step - accuracy: 0.9675 - loss: 0.1044 - val_accuracy: 0.0000e+00 - val_loss: 12.6948
Epoch 5/10
71/71 ██████████ 3s 36ms/step - accuracy: 0.9739 - loss: 0.0800 - val_accuracy: 0.0000e+00 - val_loss: 13.7029
Epoch 6/10
71/71 ██████████ 2s 34ms/step - accuracy: 0.9803 - loss: 0.0622 - val_accuracy: 0.0000e+00 - val_loss: 14.0429
Epoch 7/10
71/71 ██████████ 2s 34ms/step - accuracy: 0.9838 - loss: 0.0495 - val_accuracy: 0.0000e+00 - val_loss: 12.5586
Epoch 8/10
71/71 ██████████ 3s 37ms/step - accuracy: 0.9850 - loss: 0.0472 - val_accuracy: 0.0000e+00 - val_loss: 12.7911
Epoch 9/10
71/71 ██████████ 3s 36ms/step - accuracy: 0.9876 - loss: 0.0359 - val_accuracy: 0.0000e+00 - val_loss: 14.5048
Epoch 10/10
71/71 ██████████ 2s 33ms/step - accuracy: 0.9867 - loss: 0.0386 - val_accuracy: 0.0000e+00 - val_loss: 15.5455

Evaluating Basic CNN...
Basic CNN test accuracy: 0.8810
Basic CNN training time: 28.40 seconds
Basic CNN testing time: 0.46 seconds
```

Figure 23: The Results of the Basic CNN network to classify images from the ReducedMNIST dataset.

Second version of the CNN that includes a spatial attention mechanism.

- Similar architecture but with spatial attention mechanisms
- Attention is applied after each of the first two convolutional blocks

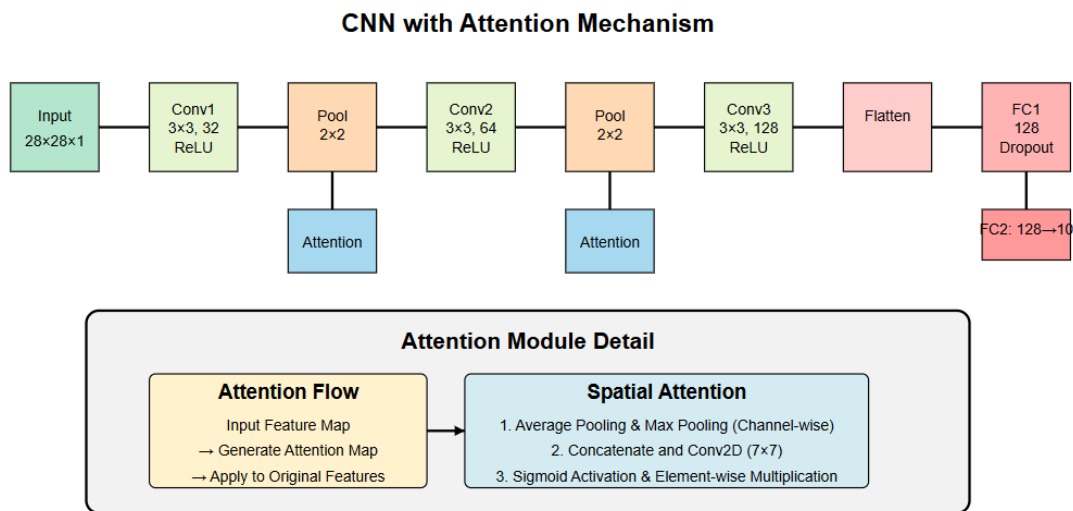


Figure 24: Block Diagram for the CNN network which includes a spatial attention

```

Training Attention CNN...
Epoch 1/10
71/71 ██████████ 6s 51ms/step - accuracy: 0.4665 - loss: 1.5603 - val_accuracy: 0.0000e+00 - val_loss: 12.2320
Epoch 2/10
71/71 ██████████ 3s 43ms/step - accuracy: 0.9259 - loss: 0.2421 - val_accuracy: 0.0000e+00 - val_loss: 11.8268
Epoch 3/10
71/71 ██████████ 3s 40ms/step - accuracy: 0.9567 - loss: 0.1451 - val_accuracy: 0.0000e+00 - val_loss: 13.4867
Epoch 4/10
71/71 ██████████ 3s 47ms/step - accuracy: 0.9654 - loss: 0.1008 - val_accuracy: 0.0000e+00 - val_loss: 12.3444
Epoch 5/10
71/71 ██████████ 3s 43ms/step - accuracy: 0.9752 - loss: 0.0799 - val_accuracy: 0.0000e+00 - val_loss: 13.9147
Epoch 6/10
71/71 ██████████ 3s 43ms/step - accuracy: 0.9764 - loss: 0.0703 - val_accuracy: 0.0000e+00 - val_loss: 14.7926
Epoch 7/10
71/71 ██████████ 3s 42ms/step - accuracy: 0.9808 - loss: 0.0608 - val_accuracy: 0.0000e+00 - val_loss: 14.1901
Epoch 8/10
71/71 ██████████ 3s 41ms/step - accuracy: 0.9817 - loss: 0.0537 - val_accuracy: 0.0000e+00 - val_loss: 14.9586
Epoch 9/10
71/71 ██████████ 3s 41ms/step - accuracy: 0.9858 - loss: 0.0430 - val_accuracy: 0.0000e+00 - val_loss: 16.3332
Epoch 10/10
71/71 ██████████ 3s 42ms/step - accuracy: 0.9896 - loss: 0.0325 - val_accuracy: 0.0000e+00 - val_loss: 15.3090

Evaluating Attention CNN...
Attention CNN test accuracy: 0.8875
Attention CNN training time: 33.81 seconds
Attention CNN testing time: 0.47 seconds

```

Figure 25: The Results of CNN network with spatial attention to classify images from the ReducedMNIST dataset.

## Compare the two models

Model	Accuracy	Training Time	Testing Time
Basic CNN	88.10%	24.40 sec	0.46 sec
CNN with attention	88.75%	33.81 sec	0.47 sec

```
--- Model Comparison ---
```

```
Basic CNN accuracy: 0.8810, training time: 28.40 seconds, testing time: 0.46 seconds
```

```
Attention CNN accuracy: 0.8875, training time: 33.81 seconds, testing time: 0.47 seconds
```

```
Accuracy difference: 0.0065
```

```
Training time difference: 5.41 seconds
```

```
Testing time difference: 0.01 seconds
```

*Figure 26: Comparison between the two CNN models*

As we can observe from the results, the CNN model integrated with a spatial attention mechanism achieves higher accuracy compared to the basic CNN model. This outcome aligns with our expectations, as spatial attention enhances the model's ability to concentrate on the most relevant and informative regions within the feature maps. In contrast, a standard CNN treats all spatial locations equally, distributing its focus across the entire image. This approach may cause the model to process unimportant areas, such as blank background regions, with the same emphasis as the parts that contain meaningful features. On the other hand, the spatial attention mechanism helps the network to prioritize critical areas, such as regions where the actual digit appears in digit classification tasks, while downplaying or ignoring less significant regions. This selective focus results in better feature representation and ultimately improves classification performance.

However, it is also important to note that incorporating spatial attention comes with a trade-off in training efficiency. The CNN model with spatial attention requires more training time than the basic CNN model. This is primarily due to the additional computational layers introduced by the attention mechanism, which are responsible for generating the attention maps. These extra layers increase the number of parameters in the network and add complexity to both the forward and backward propagation processes during training. As a result, the model takes longer to train, although the performance benefits often justify this additional time cost.

## Future Improvements for the CNN network with attention mechanism

- **Self-Attention Mechanisms:** Borrow ideas from transformers by implementing self-attention mechanisms that can model long-range dependencies between different spatial locations in the feature maps.
- **Multi-Head Attention:** Instead of a single attention pathway, implement multiple attention heads that can focus on different aspects of the image simultaneously. Each head could learn to attend to different feature patterns, and their outputs could be combined for more comprehensive feature representation.

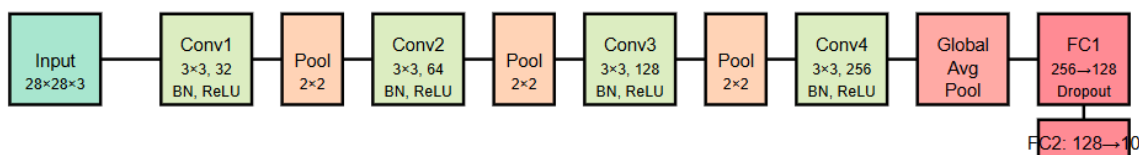
## Problem 4: Understanding the Impact of Attention Mechanisms (Part b)

### 1. Network Architecture:

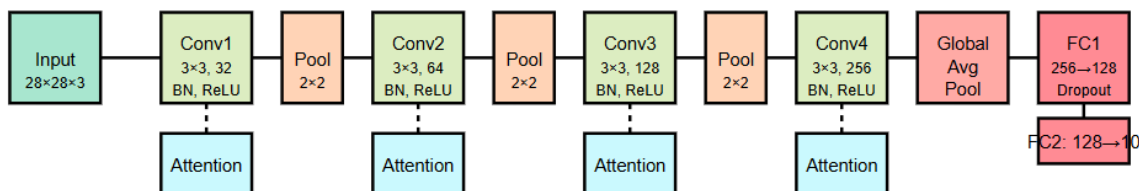
The base network is a basic CNN with four convolutional layers which are ReLU activated equipped with batch normalization each followed by a Max Pool layer except for the last layer which is followed by an Average Pool layer. At the end, there is a two layer fully connected neural network followed by a dropout layer for regularization. To add support for attention mechanism, an attention layer is embedded after each convolutional layer. This attention layer implements attention in two submodules which are Channel Attention and Spatial Attention. Channel Attention learns which channels are important using global average pooling and  $1 \times 1$  convolutions and outputs a channel-wise weight map, multiplied with the input. Spatial Attention learns where to focus spatially within the feature map and outputs a spatial weight map, applied elementwise. Both submodules implement sigmoid activation instead of traditional softmax used for attention and this is because sigmoid activation is more suited for CNNs than softmax which is more suitable for NLP. This is because sigmoid applies attention in a way independent of other pixels, rather than softmax which applies attention to favor parts of speech at the expense of other parts. This called a CBAM (Convolutional Block Attention Module). The architecture of the network is shown as follows:

#### AudioMNISTCNN Architecture

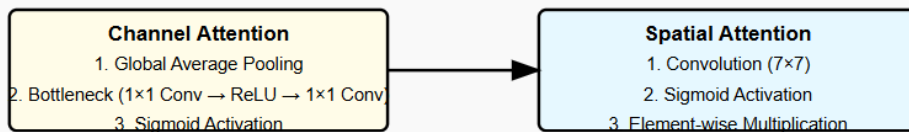
##### Standard CNN (Without Attention)



##### CNN with Attention Mechanism



##### Attention Module Detail



## 2. Training Process and Hyperparameters Choice:

Training is done using the same dataset used in assignment 2 for the audio recognition based on spectrograms of the 10 digits. The training is carried out using a learning rate of 0.0001 with 20 epochs which were enough as will be seen to reach convergence in training. Training is carried out using 50 audio samples per digit. The results of the training are outlined next.

## 3. Training Results

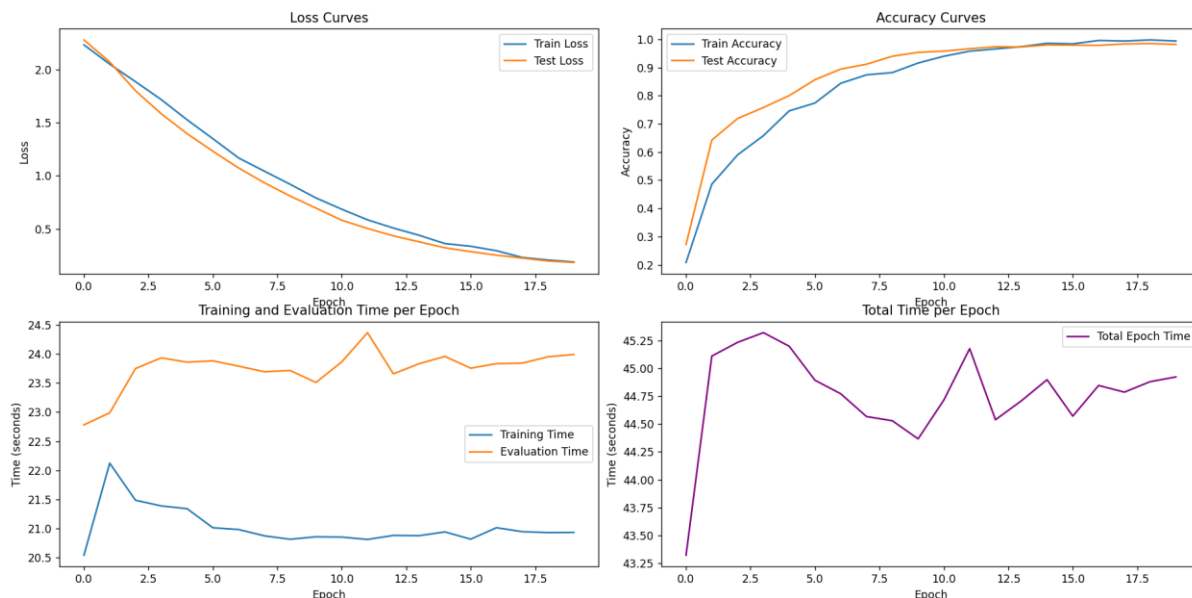


Figure 27: Loss and Accuracy curves along with time spent per epoch for training the CNN without attention

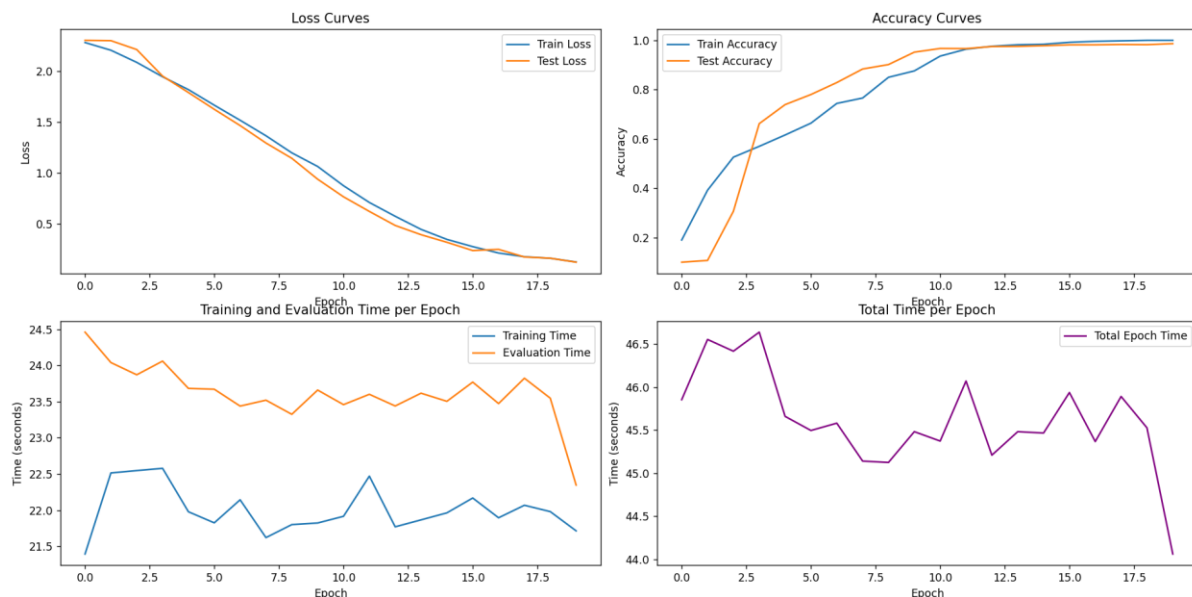


Figure 28: Loss and Accuracy curves along with time spent per epoch for training the CNN with attention

```

Epoch 1/20, Train Loss: 2.2328, Train Acc: 0.2080, Test Loss: 2.2794, Test Acc: 0.2720, LR: 0.000100, Train Time: 20.54s, Eval Time: 22.78s, Total Epoch Time: 43.32s
Epoch 2/20, Train Loss: 2.0512, Train Acc: 0.4860, Test Loss: 2.0714, Test Acc: 0.6425, LR: 0.000100, Train Time: 22.12s, Eval Time: 22.99s, Total Epoch Time: 45.11s
Epoch 3/20, Train Loss: 1.8864, Train Acc: 0.5900, Test Loss: 1.8809, Test Acc: 0.7190, LR: 0.000100, Train Time: 21.49s, Eval Time: 23.75s, Total Epoch Time: 45.23s
Epoch 4/20, Train Loss: 1.7173, Train Acc: 0.6580, Test Loss: 1.5837, Test Acc: 0.7580, LR: 0.000100, Train Time: 21.39s, Eval Time: 23.93s, Total Epoch Time: 45.32s
Epoch 5/20, Train Loss: 1.5276, Train Acc: 0.7460, Test Loss: 1.3965, Test Acc: 0.8003, LR: 0.000100, Train Time: 21.34s, Eval Time: 23.86s, Total Epoch Time: 45.20s
Epoch 6/20, Train Loss: 1.3495, Train Acc: 0.7740, Test Loss: 1.2319, Test Acc: 0.8567, LR: 0.000100, Train Time: 21.01s, Eval Time: 23.88s, Total Epoch Time: 44.89s
Epoch 7/20, Train Loss: 1.1680, Train Acc: 0.8440, Test Loss: 1.0734, Test Acc: 0.8945, LR: 0.000100, Train Time: 20.98s, Eval Time: 23.79s, Total Epoch Time: 44.77s
Epoch 8/20, Train Loss: 1.0435, Train Acc: 0.8740, Test Loss: 0.9340, Test Acc: 0.9117, LR: 0.000100, Train Time: 20.87s, Eval Time: 23.69s, Total Epoch Time: 44.57s
Epoch 9/20, Train Loss: 0.9199, Train Acc: 0.8820, Test Loss: 0.8095, Test Acc: 0.9400, LR: 0.000100, Train Time: 20.82s, Eval Time: 23.71s, Total Epoch Time: 44.53s
Epoch 10/20, Train Loss: 0.7915, Train Acc: 0.9160, Test Loss: 0.6961, Test Acc: 0.9538, LR: 0.000100, Train Time: 20.86s, Eval Time: 23.51s, Total Epoch Time: 44.37s
Epoch 11/20, Train Loss: 0.6855, Train Acc: 0.9400, Test Loss: 0.5816, Test Acc: 0.9582, LR: 0.000100, Train Time: 20.85s, Eval Time: 23.86s, Total Epoch Time: 44.72s
Epoch 12/20, Train Loss: 0.5856, Train Acc: 0.9580, Test Loss: 0.5042, Test Acc: 0.9667, LR: 0.000100, Train Time: 20.81s, Eval Time: 24.37s, Total Epoch Time: 45.18s
Epoch 13/20, Train Loss: 0.5084, Train Acc: 0.9660, Test Loss: 0.4349, Test Acc: 0.9738, LR: 0.000100, Train Time: 20.88s, Eval Time: 23.66s, Total Epoch Time: 44.54s
Epoch 14/20, Train Loss: 0.4402, Train Acc: 0.9740, Test Loss: 0.3786, Test Acc: 0.9730, LR: 0.000100, Train Time: 20.88s, Eval Time: 23.83s, Total Epoch Time: 44.71s
Epoch 15/20, Train Loss: 0.3622, Train Acc: 0.9860, Test Loss: 0.3226, Test Acc: 0.9800, LR: 0.000100, Train Time: 20.94s, Eval Time: 23.96s, Total Epoch Time: 44.90s
Epoch 16/20, Train Loss: 0.3368, Train Acc: 0.9840, Test Loss: 0.2863, Test Acc: 0.9795, LR: 0.000100, Train Time: 20.82s, Eval Time: 23.75s, Total Epoch Time: 44.57s
Epoch 17/20, Train Loss: 0.2947, Train Acc: 0.9960, Test Loss: 0.2529, Test Acc: 0.9787, LR: 0.000100, Train Time: 21.01s, Eval Time: 23.83s, Total Epoch Time: 44.85s
Epoch 18/20, Train Loss: 0.2315, Train Acc: 0.9940, Test Loss: 0.2255, Test Acc: 0.9837, LR: 0.000100, Train Time: 20.95s, Eval Time: 23.84s, Total Epoch Time: 44.79s
Epoch 19/20, Train Loss: 0.2073, Train Acc: 0.9980, Test Loss: 0.1982, Test Acc: 0.9852, LR: 0.000100, Train Time: 20.93s, Eval Time: 23.95s, Total Epoch Time: 44.88s
Epoch 20/20, Train Loss: 0.1889, Train Acc: 0.9940, Test Loss: 0.1842, Test Acc: 0.9822, LR: 0.000100, Train Time: 20.93s, Eval Time: 23.99s, Total Epoch Time: 44.92s
Total training time: 895.37 seconds

```

Figure 29: Detailed Logs for the Losses and accuracy evolution per epoch along with the training and evaluation time per epoch for training without attention

```

Epoch 1/20, Train Loss: 2.2816, Train Acc: 0.1900, Test Loss: 2.3833, Test Acc: 0.1000, LR: 0.000100, Train Time: 21.39s, Eval Time: 24.46s, Total Epoch Time: 45.85s
Epoch 2/20, Train Loss: 2.2060, Train Acc: 0.3920, Test Loss: 2.3002, Test Acc: 0.1072, LR: 0.000100, Train Time: 22.51s, Eval Time: 24.04s, Total Epoch Time: 46.56s
Epoch 3/20, Train Loss: 2.0867, Train Acc: 0.5260, Test Loss: 2.2126, Test Acc: 0.3065, LR: 0.000100, Train Time: 22.55s, Eval Time: 23.87s, Total Epoch Time: 46.42s
Epoch 4/20, Train Loss: 1.9451, Train Acc: 0.5700, Test Loss: 1.9517, Test Acc: 0.6620, LR: 0.000100, Train Time: 22.58s, Eval Time: 24.06s, Total Epoch Time: 46.64s
Epoch 5/20, Train Loss: 1.8188, Train Acc: 0.6160, Test Loss: 1.7901, Test Acc: 0.7392, LR: 0.000100, Train Time: 21.98s, Eval Time: 23.68s, Total Epoch Time: 45.66s
Epoch 6/20, Train Loss: 1.6660, Train Acc: 0.6640, Test Loss: 1.6255, Test Acc: 0.7882, LR: 0.000100, Train Time: 21.82s, Eval Time: 23.67s, Total Epoch Time: 45.50s
Epoch 7/20, Train Loss: 1.5178, Train Acc: 0.7440, Test Loss: 1.4658, Test Acc: 0.8287, LR: 0.000100, Train Time: 22.14s, Eval Time: 23.44s, Total Epoch Time: 45.58s
Epoch 8/20, Train Loss: 1.3654, Train Acc: 0.7660, Test Loss: 1.2941, Test Acc: 0.8837, LR: 0.000100, Train Time: 21.62s, Eval Time: 23.52s, Total Epoch Time: 45.14s
Epoch 9/20, Train Loss: 1.1976, Train Acc: 0.8500, Test Loss: 1.1440, Test Acc: 0.9017, LR: 0.000100, Train Time: 21.80s, Eval Time: 23.33s, Total Epoch Time: 45.13s
Epoch 10/20, Train Loss: 1.0637, Train Acc: 0.8760, Test Loss: 0.9386, Test Acc: 0.9520, LR: 0.000100, Train Time: 21.82s, Eval Time: 23.66s, Total Epoch Time: 45.48s
Epoch 11/20, Train Loss: 0.8745, Train Acc: 0.9360, Test Loss: 0.7656, Test Acc: 0.9670, LR: 0.000100, Train Time: 21.91s, Eval Time: 23.46s, Total Epoch Time: 45.37s
Epoch 12/20, Train Loss: 0.7098, Train Acc: 0.9640, Test Loss: 0.6219, Test Acc: 0.9665, LR: 0.000100, Train Time: 22.47s, Eval Time: 23.60s, Total Epoch Time: 46.07s
Epoch 13/20, Train Loss: 0.5743, Train Acc: 0.9760, Test Loss: 0.4841, Test Acc: 0.9748, LR: 0.000100, Train Time: 21.77s, Eval Time: 23.44s, Total Epoch Time: 45.21s
Epoch 14/20, Train Loss: 0.4455, Train Acc: 0.9820, Test Loss: 0.3932, Test Acc: 0.9755, LR: 0.000100, Train Time: 21.87s, Eval Time: 23.62s, Total Epoch Time: 45.49s
Epoch 15/20, Train Loss: 0.3470, Train Acc: 0.9840, Test Loss: 0.3186, Test Acc: 0.9782, LR: 0.000100, Train Time: 21.96s, Eval Time: 23.50s, Total Epoch Time: 45.47s
Epoch 16/20, Train Loss: 0.2760, Train Acc: 0.9920, Test Loss: 0.2373, Test Acc: 0.9817, LR: 0.000100, Train Time: 22.17s, Eval Time: 23.77s, Total Epoch Time: 45.94s
Epoch 17/20, Train Loss: 0.2131, Train Acc: 0.9960, Test Loss: 0.2491, Test Acc: 0.9818, LR: 0.000100, Train Time: 21.89s, Eval Time: 23.47s, Total Epoch Time: 45.37s
Epoch 18/20, Train Loss: 0.1764, Train Acc: 0.9980, Test Loss: 0.1741, Test Acc: 0.9832, LR: 0.000100, Train Time: 22.07s, Eval Time: 23.82s, Total Epoch Time: 45.89s
Epoch 19/20, Train Loss: 0.1615, Train Acc: 1.0000, Test Loss: 0.1620, Test Acc: 0.9825, LR: 0.000100, Train Time: 21.98s, Eval Time: 23.55s, Total Epoch Time: 45.53s
Epoch 20/20, Train Loss: 0.1248, Train Acc: 1.0000, Test Loss: 0.1230, Test Acc: 0.9865, LR: 0.000100, Train Time: 21.71s, Eval Time: 22.35s, Total Epoch Time: 44.06s
Total training time: 912.36 seconds

```

Figure 30: Detailed Logs for the Losses and accuracy evolution per epoch along with the training and evaluation time per epoch for training with attention



#### 4. Performance comparison table

	<b>CNN without Attention</b>	<b>CNN with Attention</b>
<b>Final Testing Accuracy</b>	98.22%	98.65%
<b>Total Training Time</b>	419.61 seconds	440.17 seconds
<b>Total Evaluation Time</b>	475.76 seconds	472.19 seconds
<b>Total Running Time</b>	895.37 seconds	912.36 seconds
<b>Average Running Time per Epoch</b>	44.7685 seconds	45.618 seconds

It can be noticed that attention results in only a slight increase in the accuracy by about 0.43%. Also, attention takes a slightly longer time to train when incorporated in the CNN.

#### 5. Analysis of the Attention Mechanism:

- When attention is incorporated in CNNs, it has a positive impact on CNNs such that, CNNs treat all regions and channels of the feature maps equally unless explicitly told otherwise while attention changes that by using two submodules. Channel Attention learns which feature maps are important (e.g., pitch, energy, formants in spectrograms). It can suppress irrelevant filters (noise) and enhance useful ones. Spatial Attention learns where in the image to look (e.g., focusing on voiced segments or specific frequencies). It can highlight signal-rich zones and suppress silence or artifacts. This mimics how humans visually and cognitively "focus" attention.
- The actual boost in performance from attention varies based on task complexity, model size, and input resolution.
- Gains from attention in vision tasks usually fall in the range of 0-1% when it comes to simple tasks like MNIST and digit recognition and may increase to 2-5% when it comes to complex tasks like ImageNet. This explains why attention only led to slight increase in the accuracy of about 0.4% as compared to NLP applications where attention leads to huge accuracy improvement.
- When it also comes to the size of the used images, the shapes are all 28x28x3 which are not considered so large for the attention to lead to a significant improvement in accuracy. In addition to the strong base CNN as well as the simple classification task, attention does not show how it really shines as compared to NLP and language model applications.

#### 6. Some insights and observations:

- It can be noticed from the results and training accuracy curves that the attention-based CNN converges slowly compared to the normal CNN with attention.
- This is reasonable and can actually be explained as follows:
- Attention manifests itself as Extra Computation + Nonlinearities where CBAM introduces additional layers (e.g., MLPs, sigmoid activations, 7x7 convolutions). This increases model depth, and adds nonlinear dynamics that can make gradients harder to flow and slow the learning of optimal weights. (More layers = more to optimize = longer to settle into good minima)
- Also attention leads to Delayed Gradient Flow where CBAM uses sigmoid activations to compute attention maps. These maps modulate feature maps multiplicatively ( $x = x * \text{attention\_map}$ ). If the attention map starts off near 0 or 1, it can suppress gradients, especially in early training. The model might need several epochs before it even starts trusting or using attention.



## 7. Suggestions for future improvements:

Future improvement includes model Architecture Improvements:

- a-) Using Larger Input Resolutions such as spectrograms at  $64 \times 64$ ,  $96 \times 96$ , or  $128 \times 128$  instead of  $28 \times 28$ . This preserves frequency-time structure better and gives attention mechanisms more room to learn useful patterns without changing in the data itself.
- b-) Deeper CNN or Residual Blocks “such as those we learnt in the lecture” where we may add more convolutional layers or replace plain blocks with ResNet-style residual blocks which improves gradient flow and learning capacity.
- c-) Upgrading Attention Module where we used an example of an attention module embedded in a CNN (CBAM) while we may try stronger attention mechanisms such as (Squeeze-and-Excitation (SE) block, Self-Attention “e.g., non-local blocks or Vision Transformers” or Transformer layers “for time-frequency sequences”)