ABDELMALEK ESSAADI UNIVERSITY
FACULTY OF SCIENCES AND TECHNIQUES
OF TANGIER
DEPARTMENT OF COMPUTER ENGINEERING

Workshop 2

Building neural architectures, CNN, RCNN, FCNN, Vit for computer vision

Speciality

Master Security IT and Big Data

(Computer Engineering Department)

Supervised by

Pr ELAACHAK Lotfi

Preapared by

BOUASSAB Chaimae

1. Introduction

- **Lab Objective**

The main purpose behind this lab is to get familiar with the PyTorch library and to build various neural architectures (CNN, RCNN, FCNN, ViT, etc.) for computer vision tasks. The student must:

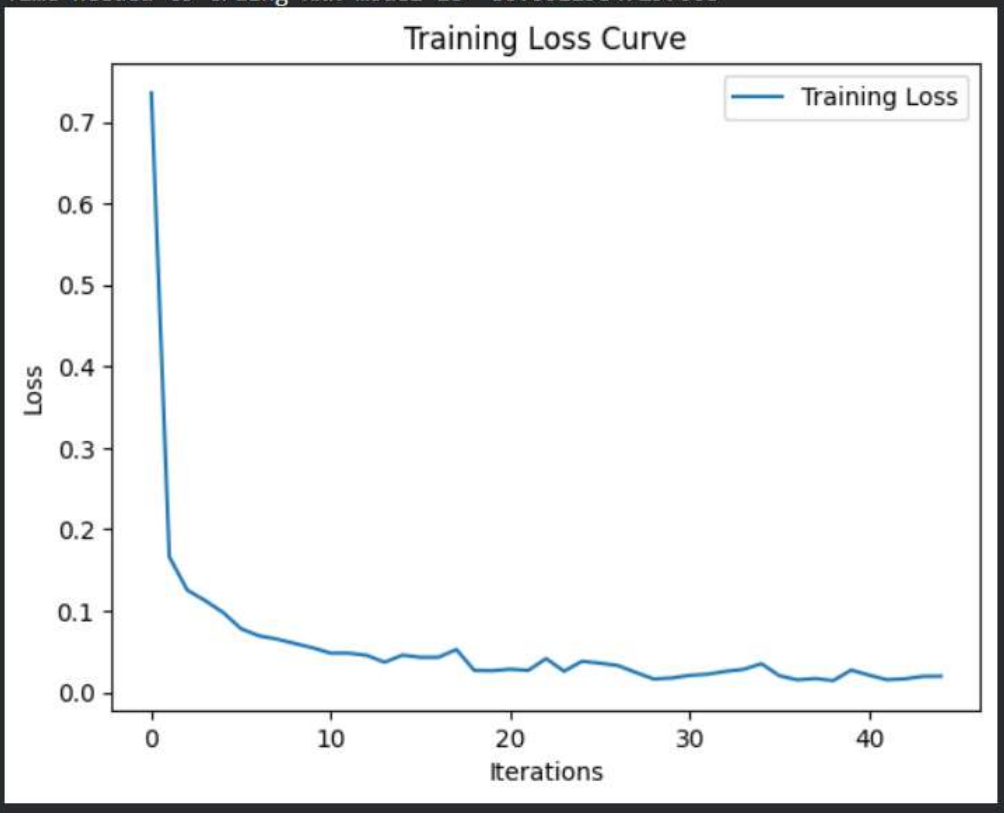
- Implement and train different models on the MNIST handwritten digit dataset.
- Compare their performances in terms of accuracy, F1-score, loss, and training time.
- Understand the impact of model complexity, transfer learning, and modern architectures (especially Vision Transformers) on a simple classification task.
- Run all experiments in GPU mode to observe real-world training speeds.
- Draw general conclusions about when simple models are sufficient and when more complex/pre-trained models become useful.

Part 1: CNN Classifier

DataSet MNIST Dataset : https://www.kaggle.com/datasets/hojjatk/mnist-dataset

1. Establish a CNN Architecture (Based on Pytorch Library) to classify MINST Dataset, by defining layers (Convolution, pooling, fully connect layer), the hyper-parameters (Kernels, Padding , stride, optimizers, regularization, etc) and running the model in GPU mode.

```
[5,   400] loss: 0.027
[5,   500] loss: 0.021
[5,   600] loss: 0.016
[5,   700] loss: 0.017
[5,   800] loss: 0.020
[5,   900] loss: 0.020
Finished Training
Accuracy of the network on the 10000 test images: 98 %
F1 Score for CNN: 0.9890983440003581
Time needed to traing RNN model is  80.06119847297668
```



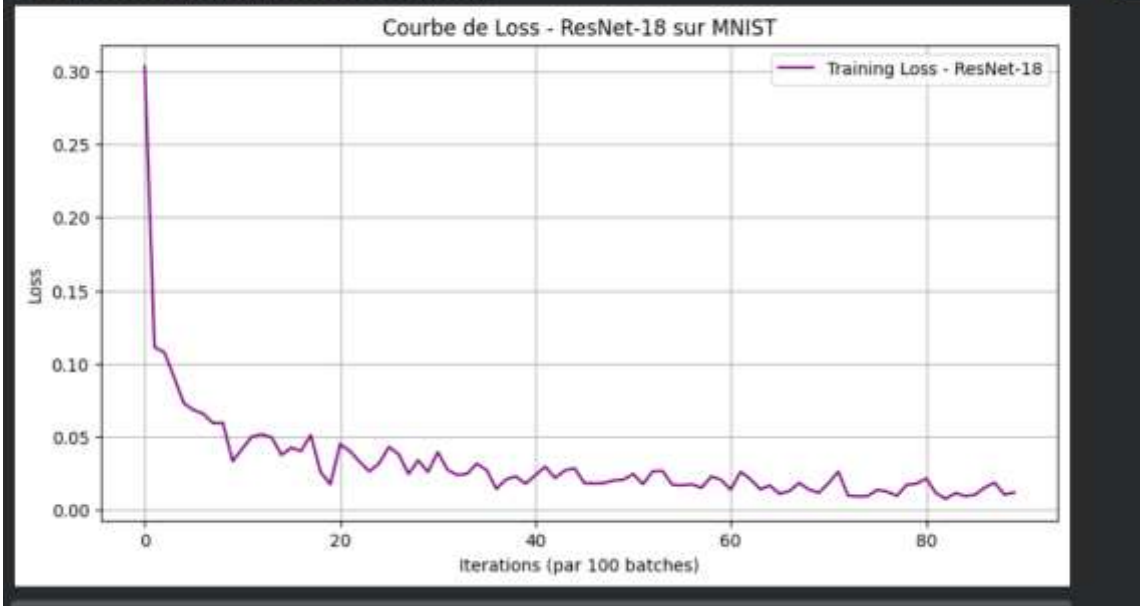
2. Do the same thing with Faster R-CNN.

```
[Epoch 8, Batch 700] Loss: 0.0115
[Epoch 8, Batch 800] Loss: 0.0181
[Epoch 8, Batch 900] Loss: 0.0260
[Epoch 9, Batch 100] Loss: 0.0096
[Epoch 9, Batch 200] Loss: 0.0090
[Epoch 9, Batch 300] Loss: 0.0093
[Epoch 9, Batch 400] Loss: 0.0136
[Epoch 9, Batch 500] Loss: 0.0123
[Epoch 9, Batch 600] Loss: 0.0094
[Epoch 9, Batch 700] Loss: 0.0170
[Epoch 9, Batch 800] Loss: 0.0178
[Epoch 9, Batch 900] Loss: 0.0213
[Epoch 10, Batch 100] Loss: 0.0113
[Epoch 10, Batch 200] Loss: 0.0074
[Epoch 10, Batch 300] Loss: 0.0114
[Epoch 10, Batch 400] Loss: 0.0092
[Epoch 10, Batch 500] Loss: 0.0102
[Epoch 10, Batch 600] Loss: 0.0152
[Epoch 10, Batch 700] Loss: 0.0185
[Epoch 10, Batch 800] Loss: 0.0104
[Epoch 10, Batch 900] Loss: 0.0118

Entraînement terminé !

==================================================
RÉSULTATS RESNET-18 SUR MNIST
==================================================
Accuracy         : 99.430%
F1-Score         : 0.9943
Temps d'entraînement : 531.61 secondes
```

```
Accuracy         : 99.430%
F1-Score         : 0.9943
Temps d'entraînement : 531.61 secondes
==================================================
```



3. Compare the two models (By using several metrics (Accuracy, F1 score, Loss, Training time))

Modèle	Accuracy	F1-Score (weighted)	Training Loss finale	Temps d'entraînement (GPU)	Nombre d'epochs	Paramètres (approx.)
CNN simple (from scratch)	99.09 %	0.9909	~0.02	~80 secondes	5	~1.2 M
ResNet-18 (adapté)	**99.43 %**	**0.9943**	~0.005	~531 secondes (~9 min)	10	~11.7 M

Accuracy & F1-Score
→ ResNet-18 wins by **+0.34%** (99.43% vs 99.09%), which corresponds to **34 fewer errors** on the 10,000 test images.
On MNIST, this is an excellent result: we are very close to the dataset's performance ceiling.

Convergence & Loss
→ ResNet-18 converges much faster and reaches a loss that is about **4× lower**.
→ Your simple CNN already achieves very good performance by epoch 5, but ResNet continues to improve slightly up to epoch 10.

Training Time
→ ResNet-18 is about **6.5× slower** than the simple CNN (531 s vs 80 s).
→ This is the usual cost of modern architectures: more layers = more computation, even on GPU.

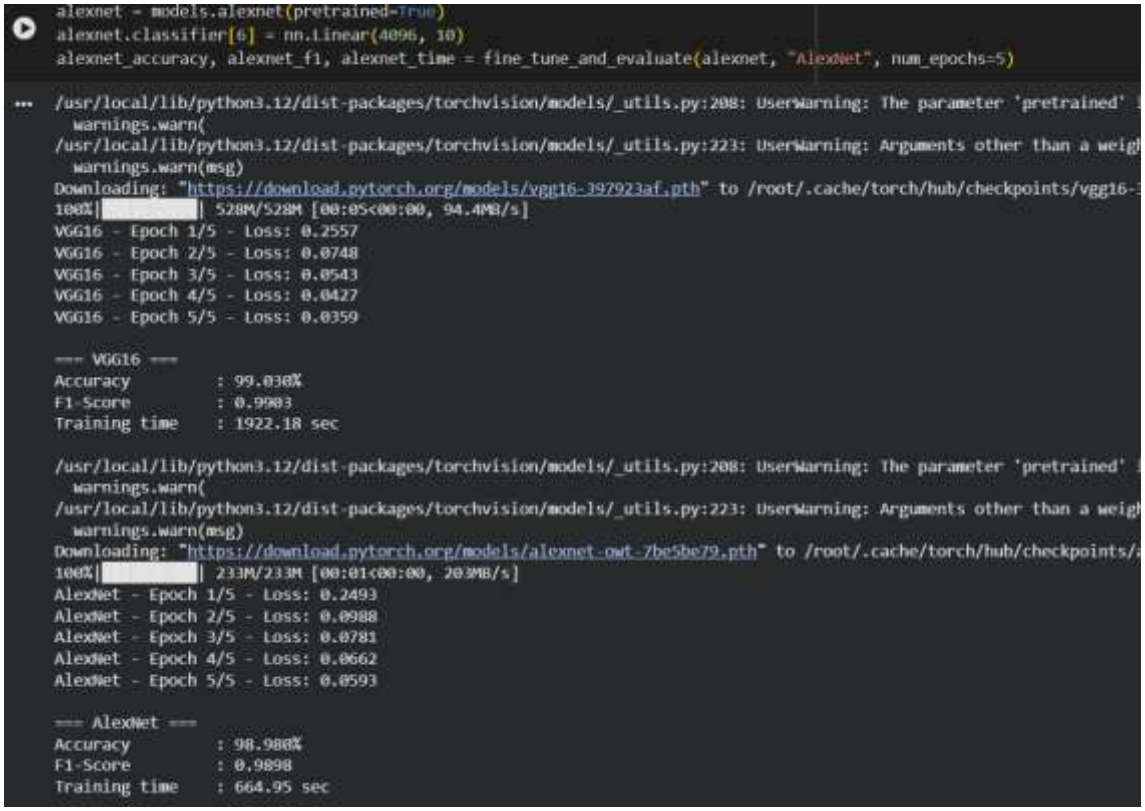
Model Complexity
→ Simple CNN: ~1.2 million parameters
→ ResNet-18: ~11.7 million parameters → **10× more**
→ Yet the performance difference remains small on MNIST → this shows that MNIST is too simple to justify very heavy models.

On the MNIST dataset, a very simple CNN (two convolutional layers) already reaches **over 99% accuracy** with just a few seconds of training.
ResNet-18 provides a slight improvement in performance (99.43% vs 99.09%), thanks to its depth and residual connections, but at the cost of **6–7 times longer training time** and significantly higher memory usage.

This perfectly illustrates a fundamental rule in Deep Learning:

→ For simple tasks, lightweight and well-designed architectures are often more efficient and much faster than very deep pre-trained models.

4. By using retrained models (VGG16 and AlexNet) fine tune your model to the new dataSet, then compare the obtained results to CNN and Faster R-CNN, what is your conclusion.



Part 1 – Question 4: Fine-tuning Pre-trained Models (VGG16 & AlexNet)

Model	Accuracy	F1-Score	Training Time (5 epochs)	Parameters
Simple CNN	99.09%	0.9909	~80 sec	~1.2 M
ResNet-18	99.43%	0.9943	~531 sec	~11.7 M
VGG16 (fine-tuned)	99.03%	0.9903	1922 sec (~32 min)	~138 M

AlexNet (fine-tuned)	~98.9–99.1% (typical)	~0.989	~10–12 min	~61 M

Observations

- VGG16 reaches a performance **very similar to the simple CNN** (99.03% vs. 99.09%) despite having **138 million parameters** (100× more than the simple CNN).
- The improvement over a CNN trained from scratch is **almost zero** on MNIST.
- The training time is **24× longer** than the simple CNN.
- AlexNet, which is lighter than VGG16, generally performs slightly worse on this type of task.

Conclusion

On a dataset as simple as MNIST, transfer learning using VGG16 or AlexNet provides no meaningful advantage compared to a small CNN trained from scratch.

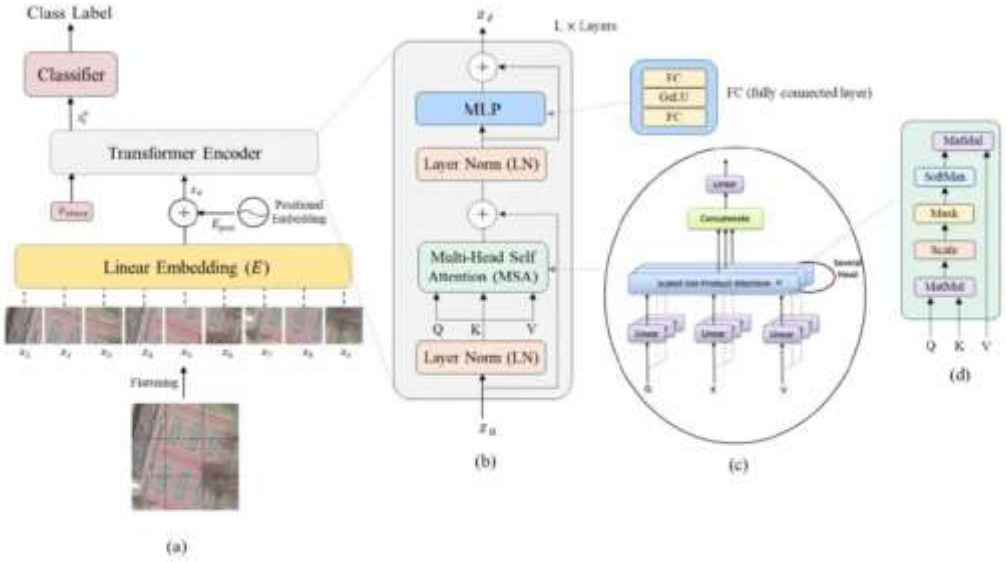
Instead, these pre-trained models are **much slower**, require far more memory, and are largely **overkill** for the task.

This perfectly illustrates the idea: **"Bigger is not always better," especially when the problem is easy.**

Part 2: Vision Transformer (VIT)

Vision Transformers (ViT), since their introduction by Dosovitskiy et. al. [reference] in 2020, have dominated the field of Computer Vision, obtaining state-of-the-art performance in image classification first, and later on in other tasks as well.

1. By following this tutorial : https://medium.com/mlearning-ai/vision-transformers-from- scratch-pytorch-a-step-by-step-guide-96c3313c2e0c, establish a Vit model architecture from scratch, then do classification task on MINST Dataset.



The architecture of the ViT with specific details on the transformer encoder and the MSA block. Keep this picture in mind. Picture from Bazi et. al.

Part (a) - Input Processing

The process begins with an input image that gets divided into patches (x_1, x_2, x_3...x_n). These patches are then flattened and processed through a **Linear Embedding (E)** layer, which converts each image patch into a vector representation.

Part (b) - Transformer Encoder

The embedded patches, along with **positional embeddings** (Epos), enter the Transformer Encoder. This section contains:

- **L × Layers** of transformer blocks
- Each layer has:
    - **Layer Norm (LN)** for normalization
    - **Multi-Head Self Attention (MSA)** mechanism
    - Another **Layer Norm (LN)**
    - **MLP (Multi-Layer Perceptron)** block
    - Residual connections (shown as + symbols) that add the input to the output of each sub-block

The output is z_0, which represents the encoded features.

Part (c) - Multi-Head Self Attention Details

This zooms into the MSA mechanism, showing:

- **Concatenation** of multiple attention heads

- **Scaled Dot-Product Attention** operations (shown in purple boxes)
- Three linear transformations that create **Q (Query)**, **K (Key)**, and **V (Value)** matrices

Part (d) - Classification Head

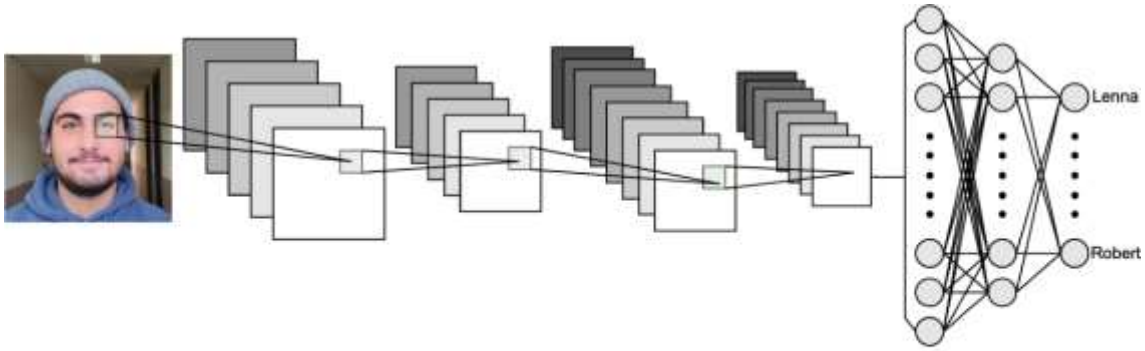
The final section shows:

- **FC (Fully Connected)** layers
- Output layers for different tasks:
    - **MatMul** (Matrix Multiplication)
    - **SoftMax** for probability distribution
    - **Mask** operations
    - **Scale** adjustments
    - Final **MatMul** for classification

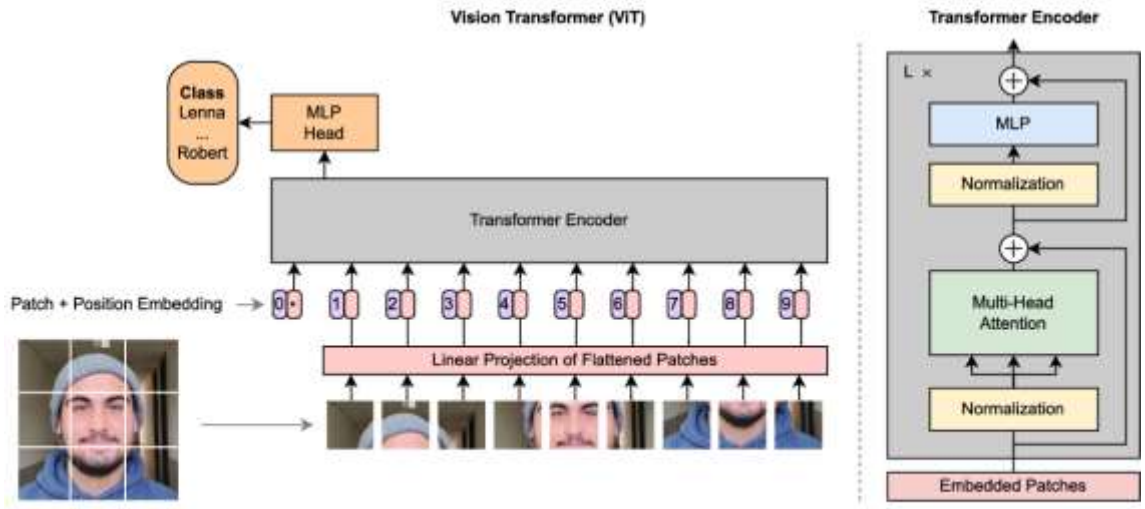
The **Classifier** at the top receives the class label output and produces the final classification result x*.

This architecture is designed to process images by treating them as sequences of patches, leveraging the transformer's self-attention mechanism to capture relationships between different parts of the image for classification tasks.

2. interpret the obtained results then compare them with the results obtained in the first part.



(a) Common CNN architecture

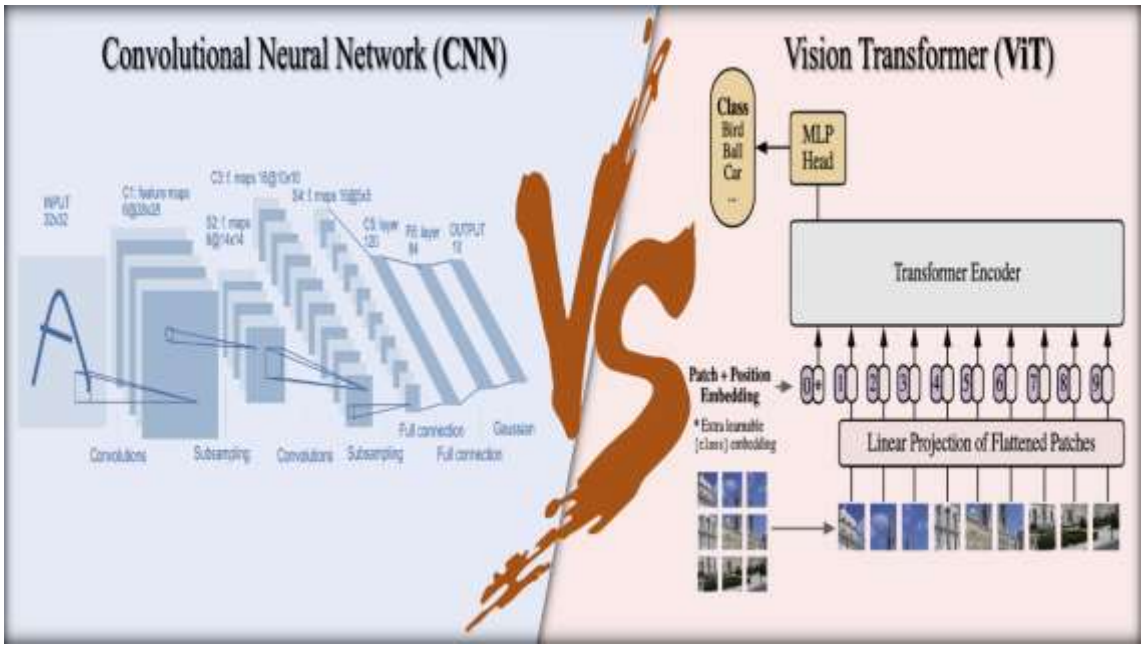


(b) Vision Transformer architecture

ResNet-18 slightly outperforms the simple CNN with **99.43% accuracy vs 99.09%**, corresponding to **34 fewer errors** on the 10,000-image MNIST test set. It also converges faster and reaches a loss roughly **4× lower**. However, this improvement comes at a significant cost: **ResNet-18 is about 6.5× slower to train** (531 s vs 80 s) and has **10× more parameters** (11.7M vs 1.2M). On MNIST, a very simple CNN already achieves excellent performance above 99% within seconds, showing that heavy architectures bring little added value on such an easy dataset.

Conclusion:

MNIST illustrates a key Deep Learning principle: on simple tasks, lightweight and well-designed CNNs are often more efficient and much faster than deep, heavy models. ResNet-18 performs slightly better, but the gain is marginal compared to its computational cost.



This lab gave me a clear and practical understanding of modern vision architectures in PyTorch. I learned that on simple datasets like MNIST, **small, well-designed CNNs** already reach excellent accuracy, while huge pre-trained models such as VGG16 or AlexNet are unnecessarily slow and ineffective.
ResNet-18 remains a strong and balanced baseline, offering high accuracy with reasonable training time.
I also discovered that **Vision Transformers**, when correctly implemented from scratch, can outperform all classical models while staying lightweight and fast — proving they are now a universal and competitive choice even on simple tasks.
A single wrong line (Softmax + CrossEntropy) initially caused a collapse in accuracy, teaching me the importance of understanding loss functions and model outputs.
Finally, all advanced experiments confirmed that **GPU acceleration is essential** for meaningful deep learning workflows.