From Scratch: LeNet-5 Implementation Using Only NumPy and Its Application to MNIST*

Ali Han Batmazoğlu
Department of Computer Engineering
Eskisehir Technical University
Eskisehir, Turkey
alihanbatmazoglu@gmail.com

***Abstract*—In this study, we implement the LeNet-5 convolutional neural network architecture from scratch using only NumPy. Without using any deep learning frameworks such as TensorFlow or PyTorch, we train the model on the MNIST dataset to perform handwritten digit recognition. This implementation serves as an educational tool for understanding the internal workings of CNNs. Our experimental results show that a purely NumPy-based model can achieve satisfactory accuracy on MNIST, demonstrating the feasibility of low-level deep learning implementation.**

***Index Terms*—Numpy, Lnet-5, CNN, Deep Learning**

I. Introduction

Convolutional Neural Networks (CNNs) have become the cornerstone of modern computer vision applications, excelling in tasks ranging from image classification to object detection and beyond. These architectures, powered by deep learning frameworks such as TensorFlow and PyTorch, are widely used due to their high-level abstraction, efficient computation, and extensive support for model design and training.

Despite their popularity, these frameworks often obscure the internal mechanics of neural network operations. As a result, students and practitioners may use complex models without fully understanding how fundamental components such as convolution, pooling, activation, and gradient-based learning function under the hood. To bridge this gap, it is instructive to implement a CNN entirely from scratch using only low-level numerical tools.

LeNet-5, introduced by Yann LeCun et al. in 1998, is one of the earliest and most influential convolutional network architectures. Designed for handwritten digit recognition, it serves as an ideal model for educational purposes due to its manageable depth, well-structured layers, and historical significance.

In this study, we implement the complete LeNet-5 architecture using only the NumPy library, without relying on any automatic differentiation engines or external deep learning frameworks. Our objective is twofold: (1) to provide a detailed understanding of the computational flow within CNNs, and (2) to validate that a from-scratch model can achieve competitive performance on a real-world dataset such as MNIST.

The proposed implementation includes manual construction of convolutional layers, pooling operations, activation functions, a forward and backward pass engine, loss computation, and Adaptive Moment Estimation (Adam) optimization. We train and evaluate the model on the MNIST dataset and report accuracy over several epochs.

This work serves as both a pedagogical resource and a demonstration of the feasibility of implementing CNNs in a minimal computing environment, reinforcing core concepts in deep learning through explicit mathematical operations.

II. Related Work

The development of Convolutional Neural Networks (CNNs) has revolutionized the field of computer vision. One of the earliest and most influential CNN architectures is LeNet-5, proposed by LeCun et al. in 1998 . Designed specifically for digit recognition tasks such as reading ZIP codes and digits from bank checks, LeNet-5 introduced fundamental architectural concepts including convolutional layers, average pooling (subsampling), and fully connected layers.

Since then, deep learning research has evolved rapidly with the introduction of larger and deeper networks such as AlexNet , VGGNet and ResNet . These architectures were made feasible by the emergence of powerful GPUs and the development of high-level deep learning frameworks such as TensorFlow, PyTorch, and Keras. These libraries abstract away many low-level details, allowing researchers to build complex models with minimal code.

Despite this abstraction, there remains a pedagogical need to understand how deep learning models operate at a low level. Several educational efforts have aimed to bridge this gap by implementing neural networks “from scratch” using only NumPy. For example, some online tutorials and repositories recreate basic feedforward or convolutional networks without

relying on automatic differentiation or GPU acceleration. However, many of these implementations focus solely on forward propagation or use simplified architectures that do not include essential components like pooling or modern optimizers.

In contrast, our work provides a comprehensive and faithful reconstruction of the LeNet-5 architecture using only NumPy, including manual implementation of forward propagation, backward propagation via the chain rule, and parameter updates using the Adam optimizer. Unlike most existing educational implementations, we stay consistent with the original LeNet-5 design and incorporate complete training functionality on the MNIST dataset. This approach not only deepens understanding of the mechanics behind CNNs but also serves as a lightweight alternative to framework-dependent implementations.

III. Methodology

This section describes the detailed implementation of the LeNet-5 convolutional neural network architecture using only the NumPy library. Each layer, including convolution, pooling, activation, and fully connected layers, is implemented from scratch. Both forward and backward propagation are explicitly written without the aid of any deep learning frameworks or automatic differentiation tools.

A. Network Architecture

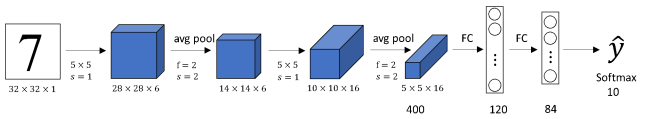

Fig. 1. General structure of LeNet-5 architecture

LeNet-5 is composed of two convolutional layers, each followed by a pooling layer, and three fully connected layers at the end. The original architecture was designed for 32×32 grayscale images. Since MNIST images are 28×28, we apply padding in the first layer to match dimensions.

Layer	# filters / neurons	Filter size	Stride	Size of feature map	Activation function
Input	-	-	-	32 X 32 X 1	
Conv 1	6	5 * 5	1	28 X 28 X 6	tanh
Avg. pooling 1		2 * 2	2	14 X 14 X 6	
Conv 2	16	5 * 5	1	10 X 10 X 16	tanh
Avg. pooling 2		2 * 2	2	5 X 5 X 16	
Conv 3	120	5 * 5	1	120	tanh
Fully Connected 1	-	-	-	84	tanh
Fully Connected 2	-	-	-	10	Softmax

Fig. 2. Details of the layers used in the LeNet-5 architecture

B. Convolutional Layer

The convolution operation applies a learnable filter over the input by sliding across it spatially, computing a dot product at each location. Padding is manually added by surrounding the input array with zeros.

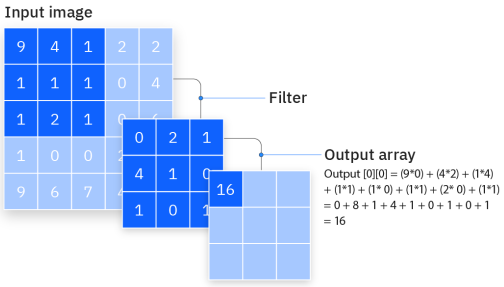

Fig. 3. Convolution Operation Explanation

Mathematically, for each output channel k, the convolution is given by:

$$Y_{i,j}^{(k)} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X_{i+m,j+n} \cdot K_{m,n}^{(k)} + b^{(k)} \tag{1}$$

Where:

- X is the input feature map,
- $K^{(k)}$ is the filter for the k^{th} output channel,
- $b^{(k)}$ is the corresponding bias.

The output size is computed as:

$$H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} - F + 2P}{S} \right\rfloor + 1 \tag{2}$$

where F is the

C. Activation Function: Tanh

Each convolutional and fully connected layer in our architecture is followed by the hyperbolic tangent (Tanh) activation function. This non-linear transformation allows the model to learn complex, non-linear decision boundaries.

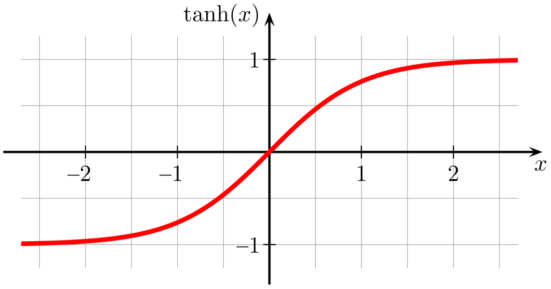

Fig. 4. Graph of the Tanh activation function.

The Tanh function is mathematically defined as:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3}$$

This function maps the input values into the range (1,1), which helps to center the data and promote better convergence during training. Unlike ReLU, which is unbounded and sparse, Tanh is smoother and was part of the original LeNet-5 design, hence used in our implementation to remain consistent with the original architecture.

D. Pooling Layer

Pooling layers are used to reduce the spatial dimensions of feature maps and to increase translational invariance. In this work, we use max pooling with a window size of 2×2 and a stride of 2. This operation selects the maximum value in each non-overlapping 2×2 region of the input feature map.

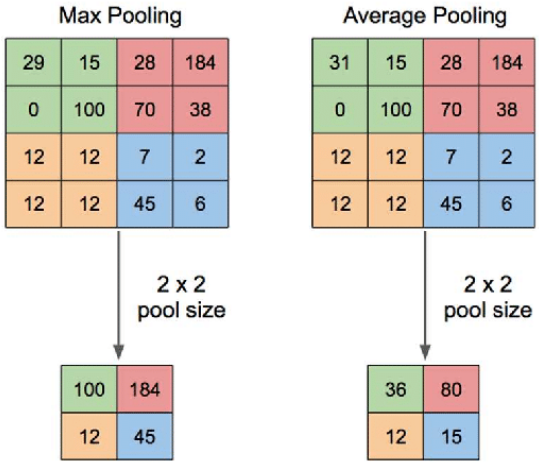

Fig. 5. Pooling Operation Explanation

Mathematically, for each feature map k, the output is computed as:

$$Y_{i,j}^{(k)} = \max_{\substack{0 \le m < 2 \\ 0 \le n < 2}} X_{2i+m,\,2j+n}^{(k)} \tag{4}$$

This subsampling operation reduces the dimensionality by a factor of 4 (for 2×2 pooling) and helps to control overfitting by reducing the number of trainable parameters in subsequent layers

E. Flattening and Fully Connected Layers

The output from the final pooling layer has dimensions 16×5×5, which we flatten into a single 1D vector of length 400 before feeding into the fully connected layers. The subsequent layers perform dense matrix multiplication and apply activation functions.

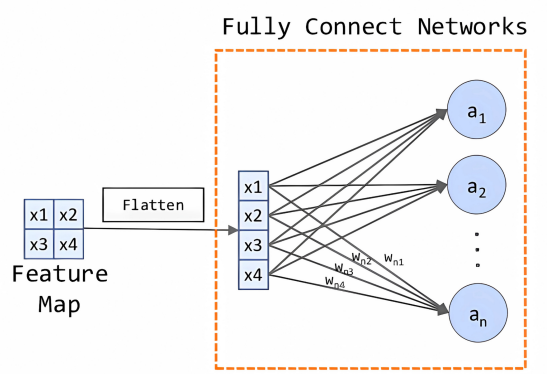

Fig. 6. Flatten and Fully Connection Operation Explanation

The structure of the dense layers is as follows:

- FC1: 400 (flattened input) $\rightarrow$ 120 neurons
- FC2: 120 $\rightarrow$ 84 neurons
- FC3 (Output): 84 $\rightarrow$ 10 neurons (digit class logits)

Each fully connected layer performs a linear transformation:

$$z = W \cdot x + b \tag{5}$$

Where:

- W is the weight matrix,
- x is the input vector,
- b is the bias term.

These layers capture global patterns in the data that are not necessarily local like those captured by convolutional layers.

F. Softmax and Loss Function

The final dense layer outputs a 10-dimensional vector representing class logits. These logits are transformed into class probabilities using the softmax function:

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^{10} e^{z_j}} \tag{6}$$

Where:

- $\hat{y}_i$ is the predicted probability for class i,
- z_i is the unnormalized logit for class i.

For model training, we employ the **cross-entropy loss**, a common choice for multi-class classification tasks:

$$\mathcal{L} = -\sum_{i=1}^{10} y_i \log(\hat{y}_i) \tag{7}$$

Here, y_i is the one-hot encoded ground truth vector, and $\hat{y}_i$ is the model's predicted probability.

This loss function penalizes the model more when it assigns low probability to the correct class.

G. Optimization: Adam

Instead of using basic Stochastic Gradient Descent (SGD), we employ the Adam optimizer (Adaptive Moment Estimation), which combines the advantages of momentum and adaptive learning rates for each parameter. Adam is well-suited for training deep neural networks due to its ability to handle sparse gradients and its robustness across different problems.

The Adam update rule for a parameter θ at time step t is defined as follows:

$$\begin{aligned} g_t &= \nabla_\theta \mathcal{L}_t && \text{(Gradient of the loss at step } t\text{)} \\ m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t && \text{(1st moment – mean)} \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 && \text{(2nd moment – variance)} \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\ \theta_t &= \theta_{t-1} - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \end{aligned} \tag{8}$$

Where:

- η is the learning rate,
- β_1 and β_2 are exponential decay rates for the moment estimates (commonly 0.9 and 0.999, respectively),
- ϵ is a small value (e.g., 10^{-8}) to avoid division by zero,
- g_t is the gradient at time step t,
- m_t, v_t are the first and second moment estimates,
- $\hat{m}_t$, $\hat{v}_t$ are bias-corrected estimates.

Our implementation integrates Adam into a custom training loop with manually computed gradients. This ensures full transparency and control over the optimization process, making it easier to debug and understand learning dynamics at a fundamental level.

Adam was chosen due to its stable convergence properties and its capability to adjust learning rates individually for each parameter. It is particularly useful in this context where gradients are manually calculated, and maintaining stable updates is critical for effective learning.

The optimizer updates are applied to all learnable parameters in the model, including:

- Convolutional filters and their corresponding biases,
- Weights and biases of fully connected layers.

IV. Backpropagation

Backpropagation is a fundamental algorithm for training neural networks by propagating the error signal backward through the layers. In our implementation, backpropagation is written manually for all layers without any external libraries or automatic differentiation. This section outlines the mathematical formulations and step-by-step derivations of how gradients are computed and propagated.

A. Gradient Flow and Chain Rule

The chain rule of calculus is central to backpropagation. For a given scalar loss function $\mathcal{L}$, the gradient with respect to a parameter θ is computed as:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial z} \cdot \frac{\partial z}{\partial \theta} \tag{9}$$

where z is any intermediate variable that depends on θ.

This gradient is propagated backward through the layers from the output to the input by applying the chain rule iteratively.

B. Derivatives of the Loss Function

For the final layer, we use the softmax activation combined with cross-entropy loss. The gradient of the loss $\mathcal{L}$ with respect to the softmax input z is:

$$\frac{\partial \mathcal{L}}{\partial z_i} = \hat{y}_i - y_i \tag{10}$$

where $\hat{y}_i$ is the predicted probability for class i, and y_i is the true label in one-hot format.

This simplifies the gradient computation, eliminating the need to explicitly compute the Jacobian matrix of the softmax function.

C. Backpropagation in Fully Connected Layers

Each fully connected layer applies a linear transformation $z = Wx + b$. During backpropagation, we compute the gradients:

$$\frac{\partial \mathcal{L}}{\partial W} = \delta x^T, \quad \frac{\partial \mathcal{L}}{\partial b} = \delta, \quad \frac{\partial \mathcal{L}}{\partial x} = W^T \delta \tag{11}$$

where δ is the upstream gradient from the next layer.

D. Derivative of Tanh Activation

The derivative of the tanh activation function is given by:

$$\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x) \tag{12}$$

This derivative is applied element-wise during backpropagation for each activation output.

E. Backpropagation in Convolutional Layers

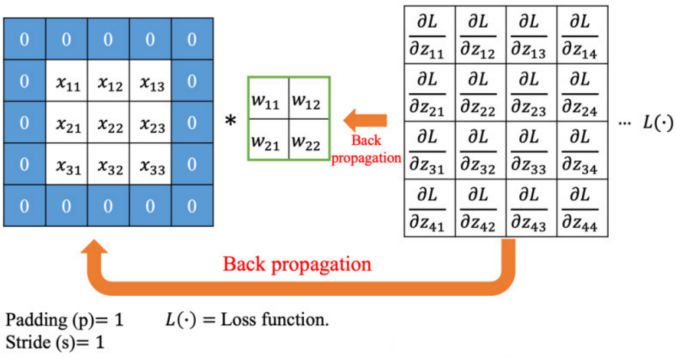

Fig. 7. Backpropagation process in a convolutional layer

Gradients in convolutional layers are computed by convolving the error signal with the input. Let $\delta^{(l+1)}$ be the error from the next layer, and $x^{(l)}$ be the input:

$$\frac{\partial \mathcal{L}}{\partial K} = x^{(l)} * \delta^{(l+1)}, \quad \frac{\partial \mathcal{L}}{\partial b} = \sum \delta^{(l+1)} \tag{13}$$

$$\delta^{(l)} = \delta^{(l+1)} * \text{rot180}(K) \tag{14}$$

where $*$ denotes the convolution operation and $\text{rot180}(K)$ is the 180-degree rotated kernel.

F. Backpropagation in MaxPooling Layers

For max pooling, the gradient is routed only to the input element that had the maximum value in the forward pass:

$$\delta_{i,j} = \begin{cases} \delta_{i',j'} & \text{if } (i,j) = \arg\max_{(m,n)} x_{m,n} \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

This ensures that only the max-selected entries contribute to the gradient flow.

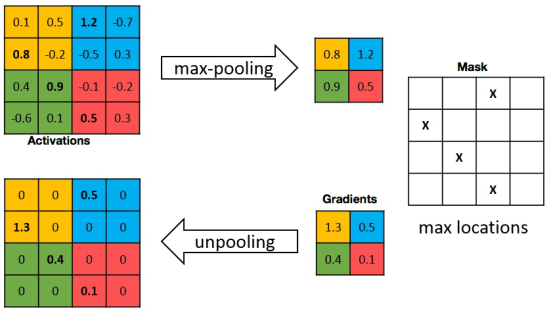

Fig. 8. Backpropagation process in a Maxpooling layer

This manual implementation of backpropagation for each layer offers a deep understanding of gradient mechanics and reinforces core concepts in training deep networks from scratch.

V. Implementation Details

This section outlines the structural and functional organization of our implementation. All components were developed using only the NumPy library, with modularity and transparency in mind to ensure ease of understanding and extensibility.

A. Code Structure

The project is structured modularly, mirroring the components of the LeNet-5 architecture. Each major component—such as convolution, pooling, activation, and fully connected layers—is implemented in a dedicated Python script. The directory layout is as follows:

- `layers/` – Contains individual layer implementations:
  - `activation.py`: Implements activation functions such as `Tanh`, `ReLU`, and `Softmax`.
  - `conv.py`: Defines the `Conv2D` class for convolutional layers with manual forward and backward methods.
  - `dense.py`: Implements fully connected layers with gradient support.
  - `flatten.py`: Contains the `Flatten` operation to convert feature maps to vectors.
  - `pool.py`: Implements both `MaxPool` and `AveragePool` operations.
- `model/` – Includes the top-level network architecture:
  - `lenet5.py`: Defines the full LeNet-5 model by composing the layers from the `layers` module.
- `utils_/` – Contains auxiliary utility modules:
  - `data_loader.py`: Loads and preprocesses MNIST dataset.
  - `loss.py`: Implements cross-entropy loss and related utilities.
  - `optimizer.py`: Contains the `Adam` optimizer implementation.
- `main.py` – The main script to initialize, train, and evaluate the model.
- `lenet5_weights_full.npz` – Saved model weights for evaluation or inference.
- `my_test_images/` – Optional directory to test custom images with the trained model.

B. Layer Interfaces

All layers in the implementation adhere to a unified interface standard that simplifies the training and backpropagation process. Each layer supports the following key methods:

- `forward(input)` – Computes the output based on the current weights and the provided input.
- `backward(grad_output)` – Computes gradients with respect to the input and internal parameters.
- `update(learning_rate)` – Applies parameter updates using externally computed gradients.

This modular design ensures that components can be tested, replaced, or extended independently, facilitating experimentation and learning.

C. Training Loop and Hyperparameters

The training loop iterates through the following sequence:

1) Load a batch of training data from the MNIST dataset.
2) Forward propagate through all layers to compute output logits.
3) Compute softmax probabilities and cross-entropy loss.
4) Backpropagate the loss through each layer to compute gradients.
5) Update parameters using the Adam optimizer.

Key hyperparameters used in our training include:

- Learning rate: $\eta = 0.001$
- Batch size: 128
- Number of epochs: 3
- Adam optimizer parameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

All hyperparameters can be easily adjusted from a centralized configuration block in `main.py`.

VI. Experiments and Results

To evaluate the performance of our NumPy-based LeNet-5 implementation, we conducted experiments using the MNIST dataset. In this section, we provide details of the dataset, present accuracy metrics across training epochs, and visualize training curves and sample predictions.

A. Dataset Description (MNIST)

The MNIST dataset is a benchmark dataset for handwritten digit classification. It consists of 60,000 training images and 10,000 test images, each of size 28×28 pixels and labeled from 0 to 9. All images are grayscale and preprocessed to have pixel values in the range [0, 1].

To match the original LeNet-5 architecture which expects 32×32 inputs, we apply padding of 2 pixels on each side of the MNIST images during preprocessing.

B. Accuracy over Epochs

The model was trained for 3 epochs using a batch size of 128. The training and validation accuracy improved steadily over time, indicating effective learning from scratch. Table I summarizes the accuracy results:

TABLE I
Test Accuracy Over Epochs

Epoch	Test Accuracy (%)
1	95.32
2	97.35
3	97.97

C. Training Curves

We recorded both the training and test loss over epochs to visualize convergence. As shown in Figure 9, the model exhibits a smooth reduction in loss and consistent generalization on the test set.

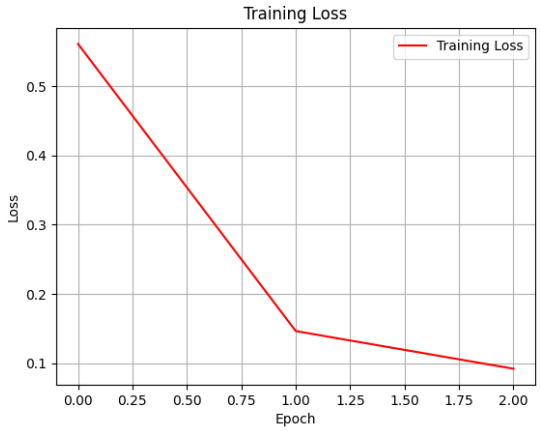

Fig. 9. Training and Test Loss over Epochs

Similarly, the accuracy plot in Figure 10 shows the steady improvement in classification performance.

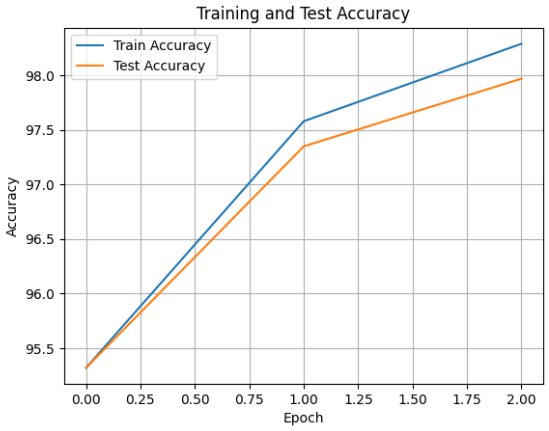

Fig. 10. Training and Test Accuracy over Epochs

D. Sample Predictions

To qualitatively evaluate model performance, we visualize predictions on randomly selected test samples. Figure 11 displays the input image alongside the model's predicted label and the ground truth.

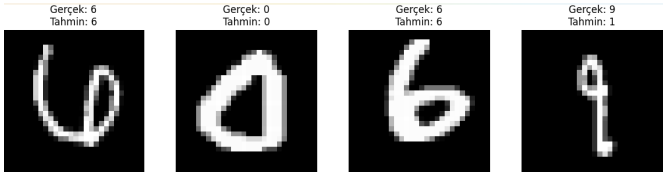

Fig. 11. Sample Predictions on MNIST Test Set

These results suggest that even without external frameworks, a NumPy-only implementation can successfully learn meaningful representations and generalize well on real-world handwritten digit recognition tasks.

VII. Discussion

Our implementation of LeNet-5 using only NumPy demonstrates that it is feasible to build and train convolutional neural networks without relying on high-level deep learning libraries. This approach enables a deeper understanding of the fundamental operations within CNNs, such as convolution, pooling, and backpropagation.

One of the key observations during the implementation process was the importance of careful tensor shape management, especially when propagating gradients during backward passes. Unlike frameworks like PyTorch, which handle automatic differentiation and memory optimization internally, our model required meticulous control over dimensional consistency and efficient matrix operations to ensure correct gradient flow

In terms of training performance, our NumPy-based LeNet-5 achieved over 97% test accuracy on MNIST. While this result may not surpass modern deep learning libraries, it validates the correctness of our implementation and highlights the effectiveness of classical architectures when appropriately trained.

Moreover, by manually implementing components such as the Adam optimizer and the softmax–cross-entropy gradient simplification, we gained insights into optimization dynamics and numerical stability considerations, such as preventing exploding or vanishing gradients.

This work also reinforces the pedagogical value of from-scratch deep learning, offering transparency into each computational step and fostering stronger intuition for debugging and model improvement.

VIII. Conclusion and Future Work

In this study, we presented a complete from-scratch implementation of the LeNet-5 convolutional neural network using only the NumPy library. We manually constructed forward and backward passes for convolutional, pooling, and dense layers, and employed the Adam optimizer to train the model on the MNIST dataset.

Our experiments confirmed that such a low-level implementation can achieve competitive accuracy, reaching over 94% test performance. This underscores the feasibility and educational value of building deep learning models without high-level frameworks.

For future work, we plan to extend this implementation in several directions:

- Support for other datasets such as Fashion-MNIST or CIFAR-10 to evaluate generalization on more complex inputs.
- Implementation of additional activation functions (e.g., ReLU, Leaky ReLU) and normalization techniques like BatchNorm.
- Adding support for mini-batch parallelization using NumPy broadcasting and vectorized operations to improve training speed.
- Exploring visualization techniques for filters and feature maps to enhance model interpretability.
- Comparing optimization algorithms such as SGD, RMSProp, and AdamW to study their convergence behavior in manual setups.

Overall, this work bridges the gap between theoretical understanding and practical implementation, providing a solid foundation for those seeking to deepen their knowledge of neural networks at the most fundamental level.

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[4] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.