Simple CNN from scratch

Phi Doan Minh Luong
2440046

***Abstract*—This project uses the MNIST dataset to present a from-scratch implementation of a basic CNN architecture. The network, which consists of a dense layer with a softmax output, a max-pooling layer, a convolutional layer, and a ReLU activation function, is built completely in Python without the use of machine learning or numerical computation libraries.**

***Index Terms*—CNN, MNIST, Deep learning**

I. Introduction

Convolution Neural Networks (CNNs) have revolutionized the field of computer vision, achieving state-of-the-art performance in tasks such as image classification, object detection, and image segmentation. Their architecture, inspired by the human visual cortex, effectively captures hierarchical patterns in visual data through a series of specialized layers, including convolutional, pooling, and fully connected layers. The widespread availability of powerful deep learning frameworks like TensorFlow, PyTorch, and Keras has made the development and deployment of complex CNN models more accessible than ever.

However, the abstraction provided by these libraries, while beneficial for rapid prototyping, can often obscure the fundamental mechanisms that drive these networks. A deep understanding of the underlying mathematical operations, data flow, and gradient computations during backpropagation is invaluable for researchers and practitioners aiming to innovate, debug, or optimize neural network models. Motivated by this, the work presented in this project focuses on implementing a simple CNN from its foundational components without the aid of such high-level libraries.

This project's main goal is to create, put into practice, and evaluate a simple CNN architecture that can recognize handwritten numbers from the popular MNIST dataset. We try to expose the pre-built neural network components by building each layer—convolution, ReLU activation, max pooling, and dense (totally connected)—as well as the backpropagation algorithm using only common Python data structures and control flow.

The key contributions of this project are:

- A detailed, step-by-step implementation of core CNN layers and their associated forward and backward pass computations in pure Python.
- A functional demonstration of the implemented CNN's ability to train on the MNIST dataset and perform digit classification.
- An analysis of the computational performance and challenges inherent in a from-scratch implementation, providing context for the efficiency of optimized numerical libraries.
- Educational insights into the intricate workings of CNNs, particularly the gradient flow during backpropagation, which are often abstracted away by modern frameworks.

II. Background

A. Convolutional Neural Networks (CNNs)

CNNs are a class of deep neural networks particularly well-suited for processing grid-like data, such as images. Their architecture typically consists of several key types of layers:

- **Convolutional Layer:** This is the core building block of a CNN. It applies a set of learnable filters (kernels) to the input image (or feature map from a previous layer). Each filter slides across the input, computing the dot product between the filter entries and the input at each position. This operation produces a 2D activation map, or feature map, highlighting specific features (e.g., edges, textures) detected by the filter. Key parameters include the number of filters, kernel size, stride (step size of the filter), and padding (adding zeros around the input border).
- **Activation Function:** Applied element-wise after the convolution operation (and often after dense layers), activation functions introduce non-linearity into the network, enabling it to learn more complex patterns. Common choices include the Rectified Linear Unit (ReLU), defined as $f(x) = \max(0, x)$, which is computationally efficient and helps mitigate the vanishing gradient problem. Other functions like Sigmoid or Tanh are also used.
- **Pooling Layer (Subsampling):** Pooling layers reduce the spatial dimensions (width and height) of the feature maps, thereby reducing the number of parameters and computation in the network. This also helps to make the representation more robust to small translations in the input. Max Pooling is a common type, where a filter slides over the input feature map and outputs the maximum value within each receptive field.
- **Flatten Layer:** After several convolutional and pooling layers, the high-level feature maps are typically flattened into a one-dimensional vector. This vector then serves as the input to one or more fully connected layers.
- **Fully Connected (Dense) Layer:** In a dense layer, every neuron is connected to every neuron in the previous layer. These layers perform classification based on the features extracted by the preceding convolutional and pooling layers.
- **Softmax Layer:** For multi-class classification tasks, the output layer typically uses a softmax activation function. Softmax converts a vector of raw scores (logits) into a

probability distribution over the K output classes, where each element represents the probability that the input belongs to a particular class.

B. MNIST Dataset

The MNIST (Modified National Institute of Standards and Technology) dataset is a widely used benchmark for image classification tasks. It consists of 70,000 grayscale images of handwritten digits (0 through 9), each of size 28x28 pixels. The dataset is pre-divided into a training set of 60,000 images and a test set of 10,000 images. Its simplicity and well-defined structure make it an ideal choice for developing and testing new machine learning algorithms, particularly for educational purposes and for implementing models from scratch.

C. From-Scratch Implementation Rationale

Modern deep learning libraries like TensorFlow and PyTorch provide highly optimized, pre-built components for constructing neural networks. While these tools accelerate development, they abstract away many of the underlying details. Implementing a CNN "from scratch," using only fundamental programming constructs and without relying on these specialized libraries (or even numerical computation libraries like NumPy for core tensor operations), offers several benefits:

- **Deeper Understanding:** It forces a detailed engagement with the mathematical operations within each layer, the data flow, and the intricacies of the backpropagation algorithm.
- **Demystification:** It helps to demystify what happens "under the hood" of high-level library functions.
- **Appreciation of Complexity:** It provides an appreciation for the computational challenges and optimizations handled by dedicated libraries.

The purposeful application of this from-scratch method in this project understands the inherent trade-off in computing efficiency as compared to library-based methods and focuses on these core learning elements.

III. Proposed CNN Architecture and Implementation

A. Overall Network Architecture

The proposed CNN architecture is intentionally kept simple to focus on the fundamental building blocks. It consists of a single convolutional layer followed by a ReLU activation, a max-pooling layer, a flatten operation, and a fully connected (dense) output layer with a softmax activation. All the configurations of the CNN can be changed by using the "config.txt" file. The data flow through the network is as follows:

1) **Input**: Grayscale MNIST images of size 28x28 pixels.
2) **Convolutional Layer (C1)**:
   - Input: 1 channel (grayscale), 28x28 image.
   - Filters: 3 filters.
   - Kernel Size: 5x5.
   - Stride: 1.
   - Padding: 'Valid' (no padding applied by the layer itself).
   - Output: 3 feature maps, each of size (28-5)/1 + 1 = 24x24.
3) **ReLU Activation (A1)**: Applied element-wise to the output of the convolutional layer C1.
4) **Max Pooling Layer (P1)**:
   - Input: 3 feature maps, each 24x24.
   - Pool Size: 2x2.
   - Stride: 2.
   - Output: 3 feature maps, each of size 24/2 = 12x12.
5) **Flatten Layer**: The 3 feature maps of size 12x12 are flattened into a single vector of size 3 * 12 * 12 = 432.
6) **Dense Output Layer (F1)**:
   - Input: Vector of size 432.
   - Neurons: 10 (one for each MNIST digit class 0-9).
   - Activation: Linear output (logits).
7) **Softmax Activation**: Applied to the output of the dense layer F1 to produce a probability distribution over the 10 classes.

A conceptual diagram of this architecture is shown in this figure

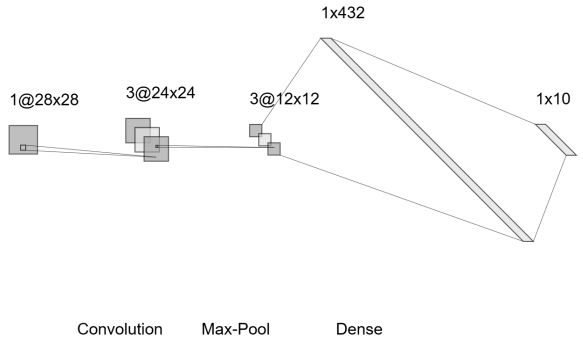

Fig. 1. Conceptual diagram of the proposed simple CNN architecture.

B. Layer Implementation Details

All layers are implemented using standard Python lists to represent tensors (multi-dimensional arrays) and basic arithmetic operations.

1) Convolutional Layer (Conv2D):

- **Forward Pass**: For each output feature map, its corresponding 5x5 kernel slides across all input channels of the 28x28 input image with a stride of 1. At each position, an element-wise multiplication between the kernel and the overlapping input region is performed, and the results are summed. This result is the pre-activation value for one element in the output feature map. To simplify, we don't use bias for the convolutional layer.
- **Backward Pass**: Gradients are computed for the layer's weights (kernels), and its input.
  - $\partial L/\partial W_k$: The gradient of the loss L with respect to a kernel weight W_k is calculated by summing the

product of the incoming gradient from the subsequent layer (post-activation) and the corresponding input activation values that contributed to that output during the forward pass.
  - $\partial L/\partial X_{in}$: The gradient with respect to the layer's input X_{in} (to be passed to the previous layer) is computed by a full convolution operation between the incoming gradient (post-activation) and the layer's kernels.

2) ReLU Activation Function:

- **Forward Pass**: For an input value z, the output a is $a = \max(0, z)$. This is applied element-wise.
- **Backward Pass**: The derivative of ReLU with respect to its input z is 1 if $z > 0$ and 0 otherwise. The incoming gradient $\partial L/\partial a$ is multiplied by this derivative: $\partial L/\partial z = (\partial L/\partial a) \cdot (1 \text{ if } a > 0 \text{ else } 0)$.

3) Max Pooling Layer (MaxPool2D):

- **Forward Pass**: A 2x2 window slides across each input feature map with a stride of 2. For each window, the maximum value is selected and becomes the corresponding element in the output feature map. The locations of these maximum values within their respective 2x2 windows are stored for use in the backward pass.
- **Backward Pass**: The gradient from the subsequent layer is routed only to the input element that was the maximum in its pooling window during the forward pass. All other input elements within that window receive a gradient of zero from that specific output gradient.

4) Flatten Layer:

- **Forward Pass**: The multi-dimensional input tensor (e.g., 3x12x12) is reshaped into a one-dimensional vector (e.g., 432 elements). The original input shape is stored.
- **Backward Pass**: The incoming one-dimensional gradient vector is reshaped back into the original multi-dimensional shape of the layer's input using the stored shape information.

5) Dense (Fully Connected) Layer:

- **Forward Pass**: For each output neuron, the dot product between its weight vector and the input vector is computed, and a bias term is added. For the final output layer, this result (logit) is passed directly without non-linear activation (as softmax will be applied next). $z_j = \sum_i w_{ji} x_i + b_j$
- **Backward Pass**:
  - $\partial L/\partial w_{ji}$: The gradient with respect to a weight w_{ji} is the product of the incoming gradient for output neuron j ($\partial L/\partial z_j$) and the corresponding input activation x_i.
  - $\partial L/\partial b_j$: The gradient with respect to bias b_j is simply the incoming gradient $\partial L/\partial z_j$.
  - $\partial L/\partial x_i$: The gradient with respect to an input activation x_i (to be passed to the previous layer) is the sum over all output neurons j of the product of the incoming gradient $\partial L/\partial z_j$ and the corresponding weight w_{ji}.

6) Softmax Activation and Loss Function:

- **Softmax Forward Pass**: The softmax function is applied to the logits z from the dense output layer to obtain class probabilities p: $p_k = \frac{e^{z_k}}{\sum_j e^{z_j}}$ for class k.
- **Loss Function**: Categorical Cross-Entropy loss is used: $L = -\sum_k y_k \log(p_k)$, where y_k is 1 if k is the true class and 0 otherwise. For a single true class c, this simplifies to $L = -\log(p_c)$.
- **Gradient for Output Layer (Pre-Softmax)**: The derivative of the Cross-Entropy loss with respect to the pre-softmax logits z_k is remarkably simple: $\partial L/\partial z_k = p_k - y_k$. This gradient is then used as the initial gradient for the backward pass into the final dense layer.

C. Training Mechanism

- **Backpropagation**: The chain rule is applied systematically to propagate the error gradients backward from the output layer through each preceding layer, computing the gradients for all learnable parameters (weights and biases).
- **Optimizer**: Stochastic Gradient Descent (SGD) is used to update the parameters. For each training sample, after computing the gradients, parameters θ are updated as: $\theta_{new} = \theta_{old} - \eta \cdot \nabla_\theta L$, where η is the learning rate.
- **Weight Initialization**: Weights for convolutional and dense layers are initialized randomly from a uniform distribution within a small range (e.g., [-0.1, 0.1]). Biases are typically initialized to zero or small random values.

D. Data Handling and Preprocessing

- **MNIST Data Loading**: The MNIST dataset is provided by the mnist library (https://github.com/datapythonista/mnist)
- **Normalization**: Pixel values, originally in the range [0, 255], are normalized to the range [-0.5, 0.5] by dividing by 255.0 and minus 0.5.
- **Input Padding**: 28x28 input images are not padded before being fed into the first convolutional layer.

The implementation removes libraries like NumPy, meaning all tensor operations (e.g., convolutions, dot products) are performed using nested Python loops, lists, lists of lists, etc.

IV. Experiments and Results

A. Experimental Setup

- **Dataset**: The standard MNIST dataset was used, consisting of 60,000 images for training and 10,000 images for testing. Due to the computational intensity of the pure Python implementation, experiments were conducted on a subset of this data:
  - Training Samples: First 1,000 images in the training set.
  - Test Samples: First 1,000 images in the testing set.

  This reduction allowed for feasible training times while still demonstrating the network's learning capability.
- **Hyperparameters**:

  - Learning Rate (η): 0.005.
  - Number of Epochs: 3
  - Batch Size: 1 (Stochastic Gradient Descent, processing one sample at a time).
  - Weight Initialization: Random uniform distribution between -0.1 and 0.1 for all weights and biases.
- **Software and Hardware Environment**: The CNN was implemented and executed in Python 3.12 on a standard laptop equipped with an Intel Core i7-10610U CPU @ 1.80GHz and 32GB RAM. No GPU acceleration was utilized.

B. Evaluation Metrics

The performance of the CNN was evaluated using the following standard metrics:

- **Accuracy**: The proportion of correctly classified images to the total number of images in the dataset (either training or testing).

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Samples}}$$

- **Categorical Cross-Entropy Loss**: This measures the dissimilarity between the predicted probability distribution and the true class label. Lower loss values indicate better performance.
- **Training Time**: The wall-clock time taken to complete the training process for the specified number of epochs and training samples.

C. Results

The network was trained on 1,000 MNIST training samples for 5 epochs. The training progress, including average loss and accuracy per epoch, is summarized in Table I. Fig. 2 and Fig. 3 visually represent the training loss and training accuracy over the epochs, respectively.

TABLE I
TRAINING PROGRESS PER EPOCH (1000 SAMPLES)

Epoch	Average Training Loss	Training Accuracy (%)
1	1.68	47.5
2	0.53	84.1
3	0.37	89.7
4	0.30	91.2
5	0.25	92.9

1) Test Performance: After 5 epochs of training, the model was evaluated on the 1000 unseen test samples.

- **Test Accuracy**: 83.7% (837 out of 1000 samples correctly classified).
- **Average Test Loss**: 0.52.

A confusion matrix for the test set predictions is presented in Fig. 4 to provide insight into which digits were commonly misclassified.

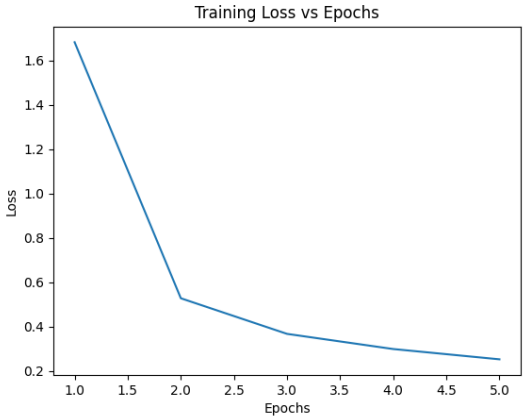

Fig. 2. Average training loss per epoch on 1000 MNIST samples.

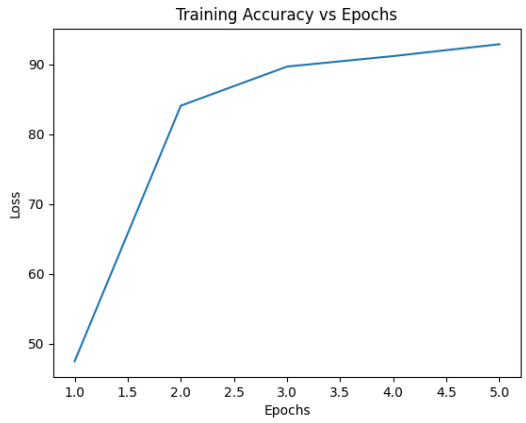

Fig. 3. Training accuracy per epoch on 1000 MNIST samples.

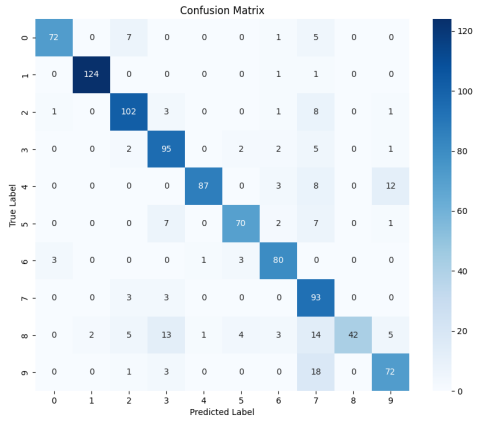

Fig. 4. Confusion matrix for predictions on 1000 MNIST test samples.

2) Execution Time: The training process for 1,000 samples over 5 epochs took approximately 4 minutes to complete.

The results demonstrate that even a simple CNN implemented from scratch can learn to classify MNIST digits with reasonable accuracy on a small subset of data. The training loss decreased and accuracy increased consistently over epochs, indicating successful learning. The test accuracy, while not state-of-the-art (which is expected given the simple architecture, small dataset, and from-scratch nature), is significantly above random chance (10%) and validates the correctness of the core implementation. The prolonged training time is a key finding, highlighting the performance benefits of optimized numerical libraries.

V. Discussion

A. Analysis of Results

The training process, as evidenced by the decreasing loss (Fig. 2) and increasing accuracy (Fig. 3) over 5 epochs (Table I), clearly indicates that the network was able to learn distinguishing features from the MNIST training samples. An initial training accuracy of 47.5% after the first epoch improved to 92.9% by the fifth epoch. This learning trend validates the correctness of the implemented forward and backward propagation mechanisms across the convolutional, ReLU, max-pooling, flatten, and dense layers.

The test accuracy of 83.7% on a previously unseen subset of 1000 test images, while modest compared to state-of-the-art models built with optimized libraries, is significantly above the 10% accuracy of random guessing. This performance further confirms that the network generalized to some extent from the training data. The drop from training accuracy to test accuracy (92.9% vs. 83.7%) is expected and indicates a degree of overfitting, which is common, especially with limited training data and a simple model without regularization techniques. The confusion matrix (Fig. 4) would further reveal specific digit pairs that the model found harder to distinguish (e.g., 7 vs. 9, or 3 vs. 8).

B. Challenges and Limitations

Implementing a CNN from scratch presented several challenges and has inherent limitations:

- **Computational Inefficiency**: The most significant challenge and limitation is the extreme computational cost. Performing tensor operations (convolutions, matrix multiplications) using nested Python loops is orders of magnitude slower than using optimized C/Fortran backends provided by libraries like NumPy, or GPU-accelerated operations in TensorFlow/PyTorch. The reported training time of approximately 4 minutes for 1,000 samples over 5 epochs highlights this inefficiency. Scaling this approach to larger datasets or more complex architectures would be practically infeasible.
- **Implementation Complexity and Debugging**: Manually implementing backpropagation for each layer, especially the convolutional layer, is intricate and prone to errors. Ensuring correct gradient calculations, tensor dimension handling, and indexing requires meticulous attention to detail. Debugging involved extensive manual checks of intermediate values and gradient flows.
- **Numerical Stability**: While not a major issue with this simple architecture and dataset, from-scratch implementations can be more susceptible to numerical stability problems (e.g., vanishing/exploding gradients, issues with softmax stability) without careful implementation of numerical tricks commonly found in established libraries.

C. Insights Gained

Despite the challenges, the from-scratch implementation yielded valuable insights:

- **Deepened Understanding of Core Mechanics**: The process provided a profound understanding of how each layer transforms data during the forward pass and how gradients are computed and propagated during the backward pass. The role of the chain rule in backpropagation became explicitly clear.
- **Appreciation for Optimized Libraries**: The stark contrast in performance with library-based implementations underscores the immense value of optimized numerical computation libraries and GPU acceleration in practical deep learning.
- **Demystification of CNN Components**: The "black box" nature of library functions was peeled away, revealing the underlying algorithms for convolution, pooling, activation, and dense connections.
- **Intuition for Hyperparameter Effects**: Manually coding the update rules provided a more direct sense of how hyperparameters like the learning rate affect the training process.

This exercise effectively served its primary educational purpose of solidifying foundational knowledge in neural network operations.

VI. Conclusion and Future Work

This project successfully detailed the design, implementation, and evaluation of a simple Convolutional Neural Network from scratch for classifying MNIST handwritten digits. Using only fundamental Python constructs, we demonstrated the functional implementation of core CNN components, including convolutional, ReLU activation, max-pooling, flatten, and dense layers, along with the backpropagation algorithm for training. The network achieved a test accuracy of 83.7% on a subset of the MNIST test data after 5 epochs of training, confirming its ability to learn and generalize.

The primary contribution lies in the educational value of this from-scratch approach, which provided a deep and practical understanding of the internal workings of CNNs. While the computational performance was significantly slower than library-based solutions, this limitation itself served to highlight the critical role of optimized numerical libraries in modern deep learning. The project successfully demystified the core operations within a CNN, offering valuable insights into data transformation and gradient flow.