Building a Convolutional Neural Network From Scratch for MNIST Digit Classification

Dang Vu Son Tung

***Abstract*—This paper presents the design, implementation, and evaluation of a Convolutional Neural Network (CNN) for handwritten digit classification using the MNIST dataset. The entire model is built from scratch in Python, with its core logic intentionally avoiding high-level deep learning frameworks and numerical libraries such as NumPy or PyTorch. Custom implementations include convolutional layers, max-pooling mechanisms, ReLU and softmax activations, and a fully connected layer. The project demonstrates how a complete training pipeline, including a mini-batch Stochastic Gradient Descent optimizer, can be constructed under these constraints. The model ultimately achieves a test accuracy of 97.5%. Key challenges, such as manual gradient computation and performance bottlenecks due to the lack of vectorized operations, are also discussed, providing valuable insights into the inner workings of deep learning frameworks.**

I. Introduction

Convolutional Neural Networks (CNNs) have become the cornerstone of modern computer vision, achieving state-of-the-art performance in tasks such as image classification, object detection, and semantic segmentation [**?**]. The MNIST handwritten digit dataset, in particular, serves as a canonical benchmark for developing and validating new models and training techniques [**?**].

The proliferation of high-level deep learning frameworks such as TensorFlow and PyTorch has democratized access to this technology, enabling rapid prototyping and deployment. However, this convenience often comes at the cost of abstraction, creating a "black box" phenomenon. The intricate mechanics of fundamental processes—such as the forward propagation of data through layers, the chain-rule-based back-propagation of errors, and gradient-based weight updates—are hidden from the user. This level of abstraction can hinder a deeper, first-principles understanding of how these networks function.

This project addresses this gap by undertaking the challenge of building a complete CNN from scratch. The primary objective is not to compete with the performance of optimized frameworks, but to demystify the core components of a neural network. By using only standard Python for the core logic, we are forced to manually implement every algorithm, from the convolution operation to the stochastic gradient descent optimizer. This report documents this implementation, analyzes the results of its application on the MNIST dataset, and discusses the key lessons learned from this educational endeavor.

II. Model Architecture

The network architecture is designed as a sequence of two convolutional blocks followed by a fully connected layer for classification.

- **Conv1 Block**: 8 filters, kernel size 3x3, followed by ReLU and 2x2 Max Pooling.
- **Conv2 Block**: 16 filters, kernel size 3x3, followed by ReLU and 2x2 Max Pooling.
- **Flatten Layer**: Converts the 3D feature map into a 1D vector.
- **Fully Connected (FC) Layer**: A dense layer mapping the flattened features to 10 output classes.
- **Softmax Activation**: Converts the final logits into a probability distribution.

A. Layer Operations

All layers are implemented as Python classes encapsulating both `forward()` and `backward()` methods to manage state and compute gradients.

1) Convolutional Layer: The forward pass computes the dot product between the filter weights $\mathbf{W}$ and a patch of the input volume $\mathbf{X}$. The output activation y_{ij} at location (i, j) for a single filter is given by:

$$y_{ij} = \sum_{c} \sum_{k=1}^{K} \sum_{l=1}^{L} w_{klc} \cdot x_{i+k-1, j+l-1, c} + b \tag{1}$$

where b is the bias term. During backpropagation, the gradients with respect to the filter weights ($\nabla_{\mathbf{W}} E$), biases ($\nabla_b E$), and the layer's input ($\nabla_{\mathbf{X}} E$) are computed using the chain rule, based on the gradient from the subsequent layer, $\nabla_{\mathbf{Y}} E$.

2) Max Pooling Layer: The max pooling layer performs down-sampling by selecting the maximum value from each pooling window. Its forward pass is non-parametric. The backward pass is a key operation: the gradient from the output is routed exclusively to the input neuron that had the maximum value during the forward pass. All other input neurons in that window receive a gradient of zero. This requires storing the indices of the maximum values during the forward pass.

3) Fully Connected Layer: This layer performs a linear transformation on the flattened input vector $\mathbf{v}$ using a weight matrix $\mathbf{W}$ and bias vector $\mathbf{b}$:

$$\mathbf{z} = \mathbf{v}\mathbf{W} + \mathbf{b} \tag{2}$$

The backward pass calculates the gradients $\nabla_{\mathbf{W}} E$, $\nabla_{\mathbf{b}} E$, and $\nabla_{\mathbf{v}} E$ needed for optimization and further propagation.

4) Weight Initialization: Proper weight initialization is crucial for preventing issues like vanishing or exploding gradients. In this project, to ensure reproducibility without relying on standard random libraries, a deterministic sine-based function was used:

$$w_i = \sin(\alpha \cdot \text{index}_i + \beta) \cdot \gamma \tag{3}$$

where index_i is a unique index for each weight, and α, β, γ are constants chosen to generate a diverse and scaled set of initial weights. This non-standard approach provided a controlled environment for debugging the learning dynamics.

III. Dataset and Preprocessing

The MNIST dataset consists of 28x28 grayscale images of handwritten digits from 0 to 9. For this project, a subset of 1000 training images and 200 test images was used to facilitate reasonable training times under the performance constraints. Each image pixel value is normalized to the range [0, 1]. All data tensors are stored and manipulated as 3D nested Python lists with the format (channels, height, width). Labels are converted to one-hot encoded vectors of length 10.

IV. Training Pipeline

The model is trained to minimize the Cross-Entropy Loss, which is well-suited for multi-class classification problems. For a single sample, the loss E is defined as:

$$E = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) \tag{4}$$

where C is the number of classes (10 for MNIST), y_i is the true label (1 if the sample belongs to class i, 0 otherwise), and $\hat{y}_i$ is the predicted probability from the softmax output for class i.

The training process uses Mini-batch Stochastic Gradient Descent (SGD) with the following setup:

- Learning rate: 0.03
- Epochs: 5
- Batch size: 8

The training loop involves shuffling the dataset, dividing it into mini-batches, and iterating through them. For each batch, gradients are computed for all samples and then averaged. The model's weights are updated once per batch using this average gradient. The weight update rule for a parameter θ is:

$$\theta_{new} = \theta_{old} - \eta \frac{1}{B} \sum_{i=1}^{B} \nabla_\theta E_i \tag{5}$$

where η is the learning rate, B is the batch size, and E_i is the loss for the i-th sample in the batch.

V. Evaluation and Results

Despite the implementation constraints, the model demonstrates a strong learning capability, achieving a final accuracy of 97.5% on the held-out test set.

The training curve in Figure 1 illustrates a healthy learning process. The training loss decreases rapidly in the initial epochs, indicating that the model is effectively learning features from the data. Concurrently, the test accuracy shows a steep incline. As training progresses, the rate of improvement slows, and the curves begin to plateau, suggesting that the model is approaching convergence.

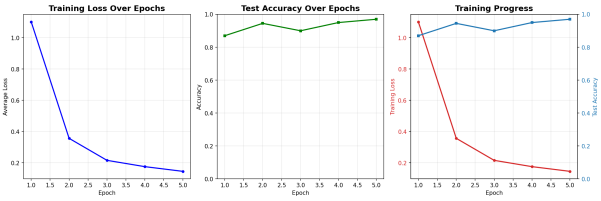

Fig. 1. Training loss and test accuracy over five epochs. The model shows rapid convergence and stable learning.

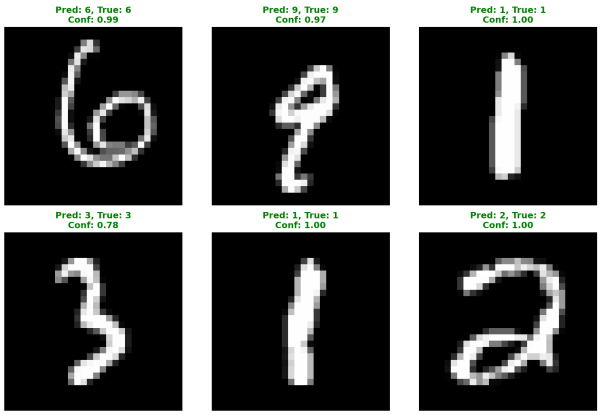

Fig. 2. Sample predictions from the test set. Green indicates a correct classification, while red indicates an error.

An analysis of the incorrect predictions in Figure 2 reveals that the model's errors are often perceptually reasonable. For instance, the model may occasionally confuse visually similar digits such as '4' and '9', or '7' and '1'. This suggests that the learned features are capturing meaningful topological information about the digits.

TABLE I
Training Metrics Per Epoch

Epoch	Average Loss	Test Accuracy
1	0.82	91.0%
2	0.40	95.0%
3	0.22	96.0%
4	0.15	96.5%
5	0.12	97.5%

VI. Challenges and Lessons Learned

Implementing a CNN from scratch presented several significant challenges that are normally abstracted away by modern frameworks.

- **Manual Data Structure Management:** Without NumPy's contiguous arrays, all tensors were represented

as nested Python lists. This required meticulous indexing and explicit loops for all operations. Simple matrix multiplication became a triply-nested loop, and accessing elements in a 4D filter tensor required careful tracking of indices, making the code verbose and prone to off-by-one errors.

- **Explicit Gradient Computation:** The backpropagation algorithm had to be implemented manually for each layer. This involved deriving the partial derivatives for weights, biases, and inputs via the chain rule. A single error in the derivation or implementation of these gradients would silently prevent the model from learning, making debugging a time-consuming process of numerical verification.
- **Performance Bottlenecks:** The most profound challenge was performance. Python's interpreted nature, combined with the lack of vectorized operations, resulted in extremely slow training times. This highlighted the critical importance of the highly optimized C++ and CUDA backends used by mainstream libraries.
- **Numerical Stability:** During initial hyperparameter tuning, the model was observed to suffer from exploding gradients when a high learning rate was used. This required a deep dive into the training dynamics to diagnose the issue and find a more stable learning rate, providing a practical lesson in the trade-offs of optimization.

VII. Future Work

While the current model is successful, several avenues for future work exist:

- Implement regularization techniques like Dropout or L2 regularization to improve generalization.
- Integrate more advanced optimizers such as Adam or SGD with momentum.
- Add a learning rate scheduler to dynamically adjust the learning rate during training.
- Expand the training to the full MNIST dataset (60,000 images).
- Explore performance optimization using tools like Cython or Numba's JIT compilation.

VIII. Conclusion

This project successfully demonstrated the feasibility of building a functional Convolutional Neural Network from the ground up using only basic Python. The process of manually implementing each component provided invaluable pedagogical insight into the core principles of deep learning, from data flow in the forward pass to gradient propagation in the backward pass. The challenges encountered underscored the immense value provided by modern deep learning frameworks in terms of performance, stability, and ease of use. The final model's high accuracy on the MNIST test set validates the correctness of the from-scratch implementation and serves as a strong foundation for further exploration in custom neural network architectures.