Implementing a Convolutional Neural Network from Scratch for MNIST Classification

Cooper Hawley, Ria Singh, Daryon Roshanzaer, Anjali Dev
Foundations of Deep Learning
University of California, Santa Barbara
`chawley@ucsb.edu, riasingh@ucsb.edu, daryon@ucsb.edu, adev@ucsb.edu`

Abstract

This paper presents the development of a convolutional neural network (CNN) implemented entirely from scratch to classify handwritten digits from the MNIST dataset. By manually building each layer without relying on high-level deep learning libraries, we aim to provide a deep understanding of CNN internals, data augmentation effects, and model performance dynamics.

1 Introduction

Image classification is a foundational task in computer vision, and the MNIST handwritten digit dataset is a widely used benchmark for evaluating machine learning models. While many high-level libraries like TensorFlow and PyTorch provide optimized tools for building and training deep learning models, our goal is to implement a convolutional neural network (CNN) completely from scratch, without relying on external APIs. By building each layer manually, from convolution and pooling to fully connected and output layers, we aim to deeply understand the internal mechanics of CNNs and how their architectural choices influence learning and performance.

This project focuses on classifying handwritten digits in the MNIST dataset using a fully custom CNN pipeline. Our work involves low-level data preprocessing, custom layer implementations, and a full training and evaluation loop. We explore the effects of various data augmentation techniques, like image jitter and label corruption, and analyze how features like symmetry and pixel intensity influence model learning. Additionally, we visualize learned filters and activation paths to gain insight into the internal representations formed during training.

2 Background

Convolutional neural networks (CNNs) have been a cornerstone of modern image recognition tasks due to their ability to capture spatial hierarchies in visual data. One of the earliest and most influential architectures is LeNet-5, introduced by Yann LeCun in the 1990s. LeNet-5 pioneered the layered structure of convolution, pooling, and fully connected layers that remains standard in deep learning models today. It achieved high accuracy on MNIST by leveraging small filters and local receptive fields to progressively extract abstract features from the input image.

While many modern implementations of LeNet and its variants are built using high-level frameworks, such abstractions often obscure the mathematical and algorithmic details underlying model training and inference. In contrast, our project unpacks these components by reimplementing them manually, thus providing a ground-up understanding of CNN behavior. Our approach also incorporates contemporary techniques such as data augmentation and advanced visualizations, bridging classical theory with modern experimentation.

3 Methods

3.1 Dataset and Preprocessing

We used the MNIST dataset, which consists of 60,000 training and 10,000 test grayscale images of handwritten digits, each of size 28×28 pixels. In the preprocessing stage, images were reshaped to the appropriate input dimensions and normalized to fall within a specified value range (e.g., [0, 1] or scaled to match LeNet-5 norms). To improve model robustness, we applied data augmentation techniques, including:

- **Pixel jittering:** Randomly turning on and off pixel values.
- **Label corruption:** Randomly scrambling the labels for 10% of the training set to simulate noisy supervision.
- **Symmetry calculation:** Computing horizontal symmetry by flipping images and comparing absolute pixel-wise differences.
- **Intensity metric:** Calculating the average pixel intensity for each image as a potential auxiliary input feature.

These enhancements aimed to enrich the diversity of training examples and test the network's generalization capabilities.

3.2 Model Architecture

The core architecture follows a LeNet-style structure, with minor adjustments for experimentation. The layers are as follows:

- **Conv1:** A convolutional layer with 6 filters of size 5×5, producing an output of shape $24 \times 24 \times 6$.
- **ReLU:** Applied element-wise to introduce non-linearity.
- **Pool1:** A 2×2 max-pooling layer reducing the spatial resolution to $12 \times 12 \times 6$.
- **Conv2:** A second convolutional layer with 16 filters of size 5×5, producing $8 \times 8 \times 16$ output.
- **ReLU:** Non-linearity after Conv2.
- **Pool2:** A second 2×2 max-pooling layer, reducing to $5 \times 5 \times 16$.
- **Flatten:** Flattening the output volume to a 256-dimensional vector.
- **FC1:** Fully connected layer mapping from 256 to 120 neurons.
- **ReLU**
- **FC2:** Fully connected layer from 120 to 84 neurons.
- **ReLU**
- **FC3:** Final fully connected layer mapping from 84 to 10 output classes.
- **Softmax:** Converts output logits into probability distributions.

Each convolution and pooling layer was implemented manually, following kernel sliding and aggregation rules. Loss gradients for these layers were also backpropagated from scratch.

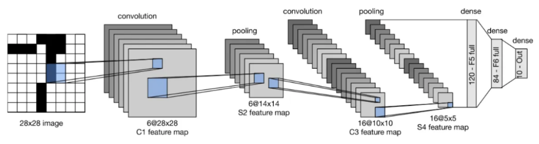

Figure 1: Architecture diagram of our LeNet-style CNN, illustrating the sequence of convolutional, pooling, and fully connected layers.

3.3 Training and Loss Calculation

For training, we minimized the negative log-likelihood (cross-entropy) loss using stochastic gradient descent (SGD) with momentum (0.9); momentum accelerates convergence by integrating a fraction of the previous update into each parameter step. Cross-entropy loss was computed between the softmax probabilities and true labels. All gradient updates were computed manually through backpropagation, with special care given to chain rule applications across matrix operations and activation functions.

We also explored the impact of residual connections, which were implemented as optional modules for select experiments. Weight initialization was conducted using the He method, and validation runs were repeated to measure accuracy variance (mean and standard deviation).

3.4 Visualization and Evaluation

To analyze the network's learning behavior, several visualizations were produced:

- Convolutional kernels after training, to examine the learned features.
- Backpropagation influence maps, highlighting which input pixels most affected classification outcomes.
- Accuracy trends over time across different model variants and corruption levels.

Multiple training sessions were conducted under varying data augmentation strategies, with performance differences documented for each configuration.

3.5 Reconstructing LeNet-5 in Pure Python

The project aimed to reconstruct LeNet-5 entirely in pure Python, eliminating reliance on PyTorch's automated components. This effort sought to provide both a deeper understanding of the inner workings of convolutional neural networks and a fully transparent, teachable implementation.

The initial step involved constructing a configurable LeNet-5 model in PyTorch. All random number generators (Python, NumPy, CPU, and CUDA) were seeded, and deterministic CuDNN settings were enabled to ensure reproducibility. A patience-2 early stopping policy was applied to identify stable training checkpoints.

With this stable PyTorch baseline in place, a phased replacement of components with custom Python implementations was undertaken. PyTorch's MaxPool2d was re-implemented using nested Python loops that manually slid a fixed window and recorded both maximum values and their corresponding indices. Additional components, dense layers, ReLU, convolution, and log-softmax, were implemented in pure Python and validated independently through finite-difference checks.

Once all layers passed their respective unit tests, the full model was assembled into a complete PythonLeNet class. This included forward and backward passes and an SGD-with-momentum update step. Initial end-to-end training runs revealed negligible learning, suggesting a mismatch in weight initialization.

To address this, a PyTorch utility script was developed to export Kaiming/Xavier-initialized weights to a JSON format (weights/lenet_seed42.json). Matching He initialization logic was implemented in Python, but seeding inconsistencies required further alignment. By ensuring that both implementations shared identical initial weights, any divergence in training behavior could be attributed to implementation errors rather than randomness.

This process led to the establishment of "Phase 0," during which a fixed 4-sample batch was used to capture full snapshots of inputs, outputs, and gradients for each layer. These reference values were stored in phase0/_reference.json. During training, each custom layer could be compared against this reference to verify correctness. Discrepancies such as off-by-one errors in pooling or sign issues in bias gradients were isolated and corrected with the aid of this reference snapshot.

Following correction of all inconsistencies, the final PythonLeNet class was assembled and integrated into a full-batch training harness. This harness utilized the MNISTWithFeats loader for consistent dataset access and logged loss values to CSV in a format mirroring the original PyTorch script. Across multiple epochs, the Python and PyTorch implementations exhibited nearly identical loss trajectories, confirming the correctness and convergence of the pure-Python model.

This phased development approach, beginning with a PyTorch reference, progressing through stepwise component replacement and JSON-based weight mirroring, and concluding with per-layer validation, provided complete transparency into the learning dynamics of convolutional neural networks.

4 Training Dynamics and Visualizations

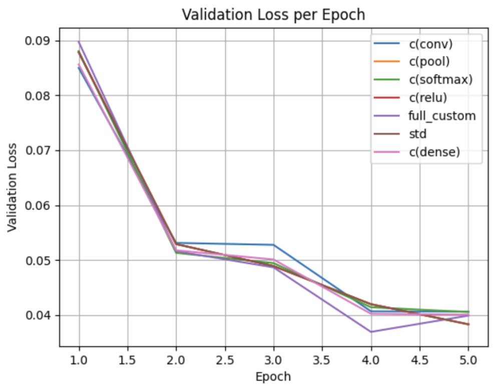

Figure 2: Validation loss per epoch

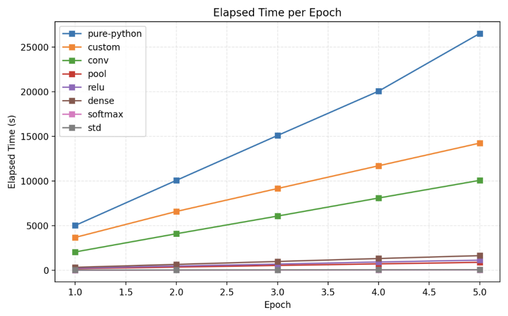

Figure 3: Elapsed time per epoch.

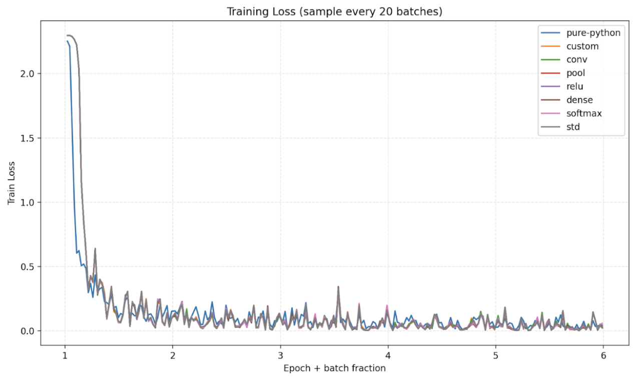

Figure 4: Training loss sampled every 20 batches

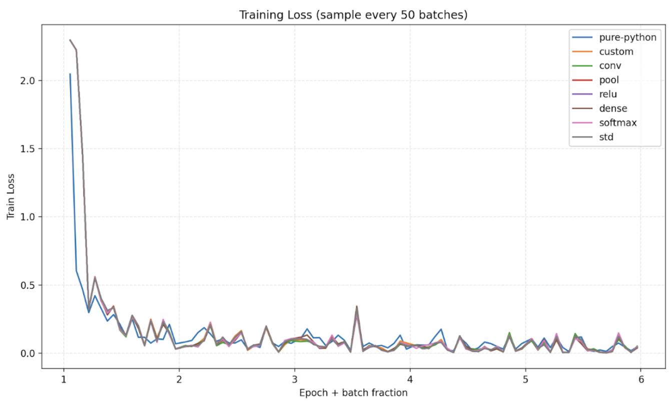

Figure 5: Training loss sampled every 50 batches

5 Results and Findings

Inference Comparison

Running inference with the pure-Python LeNet checkpoint (best_model_full_python.json), we processed all 40 test batches in roughly 6.6 seconds each and achieved a test accuracy of 98.17%. Likewise, the PyTorch LeNet model loaded from checkpoints/best_model.pt completed 79 batches at about 193 batches/second, yielding 98.36% accuracy on the same MNIST test set. Both results exceed our 98% target, so we're very pleased with the performance of both implementations.

Robustness Under Corruption and Noise

Our experimental results demonstrate a surprising degree of robustness in classification performance, even when supplied with corrupted input features and training labels. Models trained with up to 20% pixel-level jitter and label noise were able to achieve validation accuracies exceeding 97%. However, these high accuracies also came with significantly elevated validation loss values.

This apparent contradiction, accurate predictions paired with high loss, can be explained by the behavior of the cross-entropy loss function. This loss penalizes predictions based not only on correctness but also on confidence. As corruption levels increased, the network's output distributions became more uncertain, leading to higher loss even as classification accuracy remained high. These results suggest that input corruption primarily affects the calibration of the model's confidence rather than its decision boundaries.

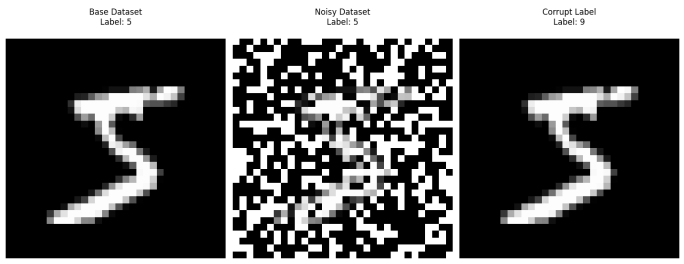

Figure 6: Base, Noisy, and Corrupt Images

Feature Space Consistency and Sensitivity

To further investigate how corruption influences the internal representations of our model, we conducted an additional analysis comparing the learned feature spaces across different augmentation levels. Specifically, we computed the ℓ_2 distance, relative change in activations, and cosine similarity between the learned weights of each corrupted model and a clean baseline across all ten layers.

Experiment	L2 Distance	Rel Change	Cos Sim
corrupt_20pct	13.990	0.5478	0.8474
corrupt_15pct	13.500	0.5173	0.8585
corrupt_10pct	12.613	0.4824	0.8684
noise_20pct	12.354	0.4819	0.8966
noise_15pct	12.134	0.4570	0.9116
corrupt_5pct	11.683	0.4110	0.9082
noise_10pct	10.930	0.3904	0.9350
noise_5pct	8.755	0.2985	0.9526

Table 1: Feature-space deviation metrics relative to clean baseline across various levels of noise and corruption.

As shown, higher levels of noise or label corruption lead to increasingly large ℓ_2 shifts in weight space and reduced cosine similarity to the clean model. However, even under substantial corruption (e.g. 20% label scrambling), cosine similarity remained above 0.84, indicating that the overall structure of the learned feature space remains largely preserved.

These results highlight the resilience of the network in maintaining a consistent internal representation, despite facing significant data disturbances. Interestingly, label corruption induces greater divergence than input noise, suggesting that supervision quality plays a more disruptive role than input degradation alone.

Implications for Evaluation Metrics

Together, these findings emphasize the limitations of relying solely on accuracy as a measure of performance. While classification rates may appear robust, the internal dynamics, like feature drift, confidence calibration, and sensitivity to label noise, can vary substantially. Our results underscore the value of incorporating additional diagnostic metrics such as cosine similarity, gradient visualization, and confidence margin distributions to holistically assess model reliability, especially in low-level or hand-crafted CNN implementations.

Gradient Analysis and Attribution Clarity

To further understand how different training conditions affect model interpretability, we visualized input gradients for representative test samples. These gradient plots are shown in Figures 7, 8, and 9 respectively.

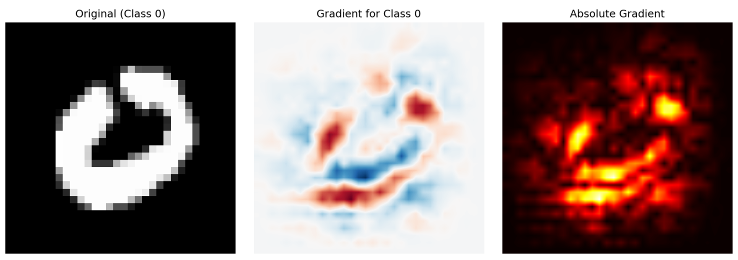

Figure 7: Input gradients for baseline model (no corruption).

Baseline (No Corruption). The baseline model produces crisp, interpretable gradients. The raw gradient emphasizes the curved stroke of the digit with well-aligned red and blue bands. Its absolute gradient highlights a focused attention loop tightly tracking the digit's structure. This suggests that the model's predictive decision is anchored in the true visual features of the input, with minimal distraction.

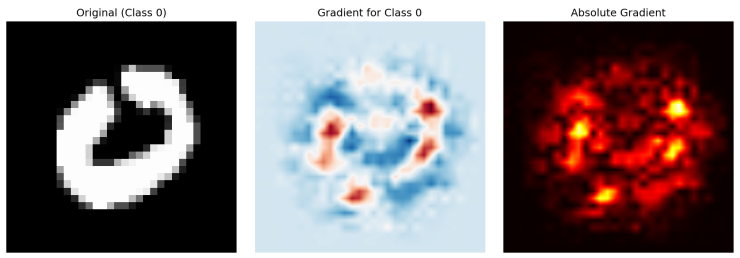

Figure 8: Input gradients for model trained with corrupted labels.

Corrupted Labels. In contrast, the model trained on randomly switched labels shows highly fragmented gradient patterns. The raw gradient becomes erratic, with red/blue patches appearing in irrelevant interior regions. The absolute gradient is scattered, lacking clear structural alignment with the digit. The model memorizes false patterns from incorrect supervision, reducing interpretability and generalization.

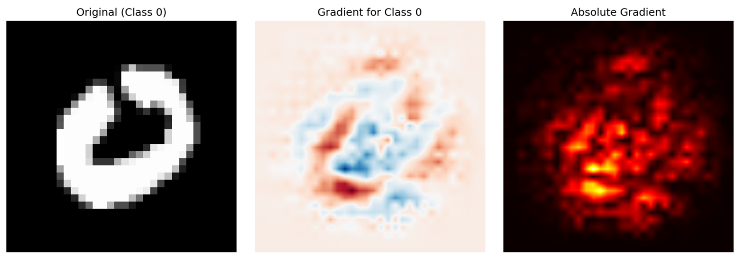

Figure 9: Input gradients for model trained with additive input noise.

Input Noise Augmentation. For models trained with pixel-level noise, gradient maps retain the overall shape of the digit, but with smoother and more diffuse activations. Raw gradients show a softened edge contrast, while absolute gradients remain spatially coherent but lack the sharp peaks seen in the baseline. This suggests a trade-off: noise regularization improves robustness and smooths attribution, though at the cost of some precision in identifying fine-grained features.

Condition	Gradient Clarity	Spatial Coherence	Noise Sensitivity
Baseline	High	High (sharp edges)	Low
Corrupted Labels	Low	Low (fragmented blobs)	Very High
Noise Augmentation	Medium	Medium (blurred structure)	Moderate

Table 2: Qualitative comparison of gradient attribution maps.

These gradient plots support what we saw earlier. Training with corrupted labels makes the model's attention messy and unreliable. Adding input noise, on the other hand, helps the model focus more broadly but still on the right areas. The baseline model gives the clearest and most accurate focus, but it may be more sensitive to changes or noise.

6 Collaborative Development Process

Our team adopted a modular and iterative approach to developing a custom LeNet-style CNN for MNIST digit classification. The process was designed to maximize both correctness and learning by isolating and testing each component individually before integrating them into a full model.

We began by implementing a baseline convolutional neural network in PyTorch, closely following the classic LeNet architecture. This model was trained and evaluated on the MNIST dataset to establish a reference for accuracy, loss, and training dynamics. The baseline served as a "gold standard" for subsequent comparisons, ensuring that any custom modifications could be directly measured against a known, working solution.

To encourage deep understanding and efficient debugging, each team member was assigned a specific layer or operation to re-implement from scratch in pure Python (e.g., convolution, pooling, activation, or dense/fully connected layers). These custom implementations were designed to match the interface and expected behavior of their PyTorch counterparts.

To verify correctness, we adopted a hybrid testing strategy:

- Each custom layer was inserted into the baseline PyTorch model, replacing the corresponding built-in layer.
- The resulting hybrid model was trained and evaluated on MNIST, using the same data splits and random seeds as the baseline.
- Performance metrics (accuracy, loss) and qualitative behaviors (e.g., convergence curves) were compared to the baseline to ensure the custom layer was functionally correct.

This approach allowed us to isolate bugs and performance issues to a single component, greatly simplifying debugging.

After all custom layers were individually validated, we assembled a fully custom model composed entirely of our own implementations. This model was then trained and evaluated on MNIST, and its performance was compared to the original PyTorch baseline. This final step provided both a rigorous test of our understanding and a clear demonstration of each layer's contribution to the overall model.

7 Contribution Statement

Ria was responsible for preprocessing the MNIST dataset, including normalization, data augmentation (pixel jittering and label corruption), and feature engineering (symmetry and pixel intensity). Daryon implemented the convolutional layers and corresponding loss computations, ensuring correct forward and backward passes. Cooper developed the PyTorch base model and logic for overriding layers, forward and backward for pooling layers, and created a Python-only training loop, integrating all layers into it. Anjali contributed the activation function implementations, fully connected (dense) layers and manually coded the cross-entropy loss calculation used during model training.

8 Github Repository

`https://github.com/daryonr/CS_190A_Project`

[1] [2] [4] [5] [3]

References

[1] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[2] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[3] The independent code. Convolutional neural network from scratch | mathematics & python code, May 2021. Accessed: 2025-06-09.

[4] Samson Zhang. Simple mnist nn from scratch (numpy, no tf/keras). `https://www.kaggle.com/code/wwsalmon/simple-mnist-nn-from-scratch-numpy-no-tf-keras`, June 2018. Kaggle Notebook.

[5] Samson Zhang. Building a neural network from scratch (no tensorflow/pytorch, just numpy & math), November 2020. Accessed: 2025-06-09.