Final Project Report

Manual Implementation of MLP and CNN for MNIST Classification

1. Introduction

In this project, I manually implemented two foundational neural network architectures, Multi-Layer Perceptron (MLP) and Convolutional Neural Network (CNN), for the task of handwritten digit classification using the MNIST dataset. The primary objective was to eschew high-level deep learning libraries such as PyTorch or TensorFlow for the model logic, and instead construct both architectures entirely from first principles using only NumPy for numerical computations. This approach allowed me to gain direct insight into the mathematical mechanisms behind forward and backward propagation, weight updates, and loss calculations.

The MNIST dataset, consisting of 60,000 training images and 10,000 test images of handwritten digits (0–9), each of size 28×28 pixels, was chosen due to its balance of simplicity and relevance in computer vision. In accordance with course constraints, I used PyTorch's `torchvision.datasets` and `DataLoader` modules solely for data loading and batching, while ensuring that all learning mechanisms and architectural logic were implemented manually. The focus throughout the project was not merely on functional performance, but also on reinforcing a theoretical understanding of neural networks through hands-on implementation, debugging, and optimization.

2. Methodology

2.1 Multi-Layer Perceptron (MLP)

The design of the MLP followed a traditional two-layer fully connected architecture. The input images were flattened into 784-dimensional vectors, corresponding to the total number of pixels in a 28×28 image. The first layer projected the input vector to a 128-dimensional hidden representation, followed by a Sigmoid activation to introduce non-linearity. The output from the hidden layer was then passed through a second fully connected layer that mapped to 10 logits, which were subsequently converted into class probabilities via the Softmax function.

Weights were initialized using a Gaussian distribution with small variance to prevent saturation of activation functions early in training. Biases were initialized to zero. Forward propagation involved computing affine transformations followed by activation functions:

$$Z1 = XW1 + b1; A1 = Sigmoid(Z1) Z_1 = XW_1 + b_1 \quad ; \quad A_1 = \text{Sigmoid}(Z_1)$$

$$Z2 = A1W2 + b2; A2 = Softmax(Z2) Z_2 = A_1W_2 + b_2 \quad ; \quad A_2 = \text{Softmax}(Z_2)$$

During backpropagation, the gradient of the cross-entropy loss with softmax simplifies to the difference between predicted and actual labels, ∂L∂Z2=y^−y\frac{\partial L}{\partial Z_2} = \hat{y} - y. This enabled efficient computation of gradients with respect to all parameters:

$$\delta 2 = y^{-}y; \delta 1 = (\delta 2 W 2 T) \cdot A1 \cdot (1 - A1) \delta_2 = \hat{y} - y \quad ; \quad \delta_1 = (\delta_2 W_2^T) \cdot A_1 \cdot (1 - A_1)$$

$Gradients\ were\ averaged\ over\ the\ batch, and$ parameters were updated using stochastic gradient descent. The training loop processed data in mini-batches of size 128 for 10 epochs, with a fixed learning rate of 0.1. All numerical operations, including forward/backward passes and weight updates, were conducted using vectorized NumPy operations for performance and clarity. Logging was integrated throughout, recording model initialization, activations, loss values, and gradients at each epoch.

2.2 Convolutional Neural Network (CNN)

Designing the CNN required a significantly deeper engagement with low-level operations, particularly for the convolutional layer. The CNN architecture consisted of one convolutional layer with 8 learnable 3×3 kernels (no padding), followed by a ReLU activation. The resulting feature maps were then flattened and passed through a fully connected layer projecting to 10 logits, which were again converted to probabilities using the Softmax function.

Initially, I implemented convolution as a set of nested loops, iterating over spatial dimensions and kernels. However, this approach quickly proved to be computationally infeasible, especially when scaling to mini-batches. To address this, I re-engineered the forward convolution to leverage NumPy's `stride_tricks.as_strided`, which enabled efficient patch extraction and convolution via batched dot products. This optimization reduced training time per epoch from over 15 minutes to under one minute while preserving correctness.

In the backward pass, I manually computed gradients with respect to the convolutional kernels and biases by aligning each receptive field patch with the corresponding upstream gradient, applying ReLU derivatives where appropriate. Flattened feature maps were backpropagated through the fully connected layer in the usual way. Weight updates were again performed using gradient descent.

The modular class structure of the CNN enabled clear separation of concerns: convolution, activation, flattening, classification, and training were all encapsulated within logically distinct components. Logging was also implemented to monitor convolution outputs, activation maps, gradients, and loss progression.

2.3 Loss Function and Training Infrastructure

Both models used categorical cross-entropy loss, implemented manually to ensure numerical stability via log clipping. The loss function operated on one-hot encoded ground truth labels and Softmax probabilities:

$$L = -\sum i = 1Cyilo\ g(y^i)\ L = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

To prevent issues with log(0), predicted probabilities were clipped between 1×10−121 \times 10^{-12} and 1−1×10−121 - 1 \times 10^{-12}. A consistent training loop was implemented for both models, leveraging PyTorch DataLoader wrappers to convert tensors into NumPy arrays. Each model's train and evaluate routines were tested and debugged independently to ensure robustness.

3. Evaluation

All experiments were conducted on a macOS laptop with CPU-only execution. Both models were implemented in Python 3.9 using only NumPy for numerical computation. The models were evaluated using test accuracy and loss progression over training epochs.

MLP Performance:
The MLP model showed consistent convergence across epochs, with training loss decreasing from an initial value of 411.08 to 67.83 by epoch 10. The final test accuracy achieved was **95.62%**, which is consistent with expectations for a fully connected network on MNIST.

CNN Performance:
After vectorization, the CNN model trained efficiently and converged steadily. Training loss dropped from 566.76 in epoch 1 to 135.88 by epoch 5. The final test accuracy was **92.33%**, which is reasonable for a shallow CNN with a single convolutional layer and no pooling.

Both models exhibited correct gradient flow and stable convergence patterns, and logs from `mlp_training_log.txt` and `cnn_training_log.txt` confirmed consistent behavior across batches and epochs.

4. Learning Outcomes

Completing this project manually was a highly enriching experience that significantly deepened my understanding of how neural networks function at the most granular level. By deriving gradients and implementing backpropagation from scratch, I was able to internalize the chain rule in the context of matrix calculus and appreciate the importance of shape consistency in complex tensor operations.

One of the major technical challenges was the efficient implementation of convolution and its gradient computation. Unlike linear layers, convolution requires careful handling of spatial alignment and broadcasting. The decision to use `np.lib.stride_tricks` proved to be a turning point in performance optimization. Additionally, building ReLU and Softmax from scratch provided valuable exposure to numerical stability issues and the role of activation dynamics in gradient flow.

The logging infrastructure I built into both models also became an invaluable tool during debugging. By capturing detailed information at each stage, forward activations, gradients, parameter updates, I was able to trace issues systematically and verify correctness without relying on external visualization tools or debuggers.

This project was completed individually, and all components, MLP and CNN architectures, forward and backward logic, optimization routines, and report documentation, were independently developed. The experience has equipped me with the confidence to reason about and implement neural networks at both the algorithmic and code level, preparing me for future work involving custom architectures or low-level model engineering.