**Final Project Report**

**Manual Implementation of MLP and CNN for MNIST Classification**

**1. Introduction**

This project focuses on building and training two fundamental types of neural networks — a Multi-Layer Perceptron (MLP) and a Convolutional Neural Network (CNN) — entirely from scratch, using only NumPy. The purpose was to strip away the abstractions of high-level deep learning libraries like PyTorch or TensorFlow and get hands-on experience with the underlying mathematics and logic that power deep learning systems. Through this manual implementation, we aimed to reinforce our theoretical understanding while encountering and solving real-world implementation challenges.

We selected the MNIST dataset for this project due to its ubiquity and clarity as a benchmark task in computer vision. MNIST comprises 60,000 training and 10,000 test images of handwritten digits (0–9), each of size 28x28 pixels. It offers a sufficiently complex pattern recognition task while remaining computationally lightweight for manual implementations. Our objective was to design, train, and evaluate the models without relying on any automatic differentiation tools or predefined layers. Instead, we wrote every function manually: activation functions, forward passes, gradient computation, and weight updates.

Additionally, PyTorch’s torchvision.datasets and torch.utils.data.DataLoader were used strictly for loading and batching MNIST images, in accordance with project guidelines. All neural network logic — including convolution, matrix multiplication, backpropagation, and loss computation — was coded manually.

This report outlines the architecture and implementation of both models, mathematical derivations used in backpropagation, training procedures, experiment results, and key lessons learned.

**2. Methodology**

**2.1 MLP Architecture and Implementation**

Our MLP consists of an input layer (784 units), one hidden layer (128 units), and an output layer (10 units). Input images are flattened into vectors before entering the model. We used a Sigmoid activation function between the layers and Softmax for the output to produce probability distributions over the 10 digit classes.

The core of the MLP’s training lies in manually executing the forward and backward passes. During the forward pass, we compute:

Z1 = np.dot(X, W1) + b1

A1 = sigmoid(Z1)

Z2 = np.dot(A1, W2) + b2

A2 = softmax(Z2)

Here, W1 and W2 are the weight matrices for the first and second layer, respectively. The backward pass calculates gradients using the chain rule. Because we're using Softmax and cross-entropy loss, the gradient of the loss with respect to the logits simplifies to:

∂L∂Z2=y^−y\frac{\partial L}{\partial Z\_2} = \hat{y} - y

Then, for the hidden layer:

δ1=((y^−y)⋅W2T)⋅A1(1−A1)\delta\_1 = ((\hat{y} - y) \cdot W\_2^T) \cdot A\_1 (1 - A\_1)

Each parameter is updated using stochastic gradient descent:

W=W−η⋅∇WW = W - \eta \cdot \nabla W

All weights and biases were initialized using Gaussian noise scaled appropriately. Training used mini-batches of size 128 for 10 epochs, and the learning rate was tuned to 0.1. The modular class structure — with forward(), backward(), train() and evaluate() methods — ensured logical separation and extensibility.

**2.2 CNN Architecture and Implementation**

The CNN implementation was significantly more complex due to the need to manually implement the 2D convolution operation. The model architecture includes:

* One convolutional layer with a single 3×3 kernel and no padding
* ReLU activation following the convolution
* A flattening step to transform the feature map into a 1D vector
* A fully connected layer to project the flattened output into 10 logits

The convolution operation was coded using nested loops, as shown:

def convolve2d(image, kernel):

output = np.zeros((ih - kh + 1, iw - kw + 1))

for i in range(output.shape[0]):

for j in range(output.shape[1]):

output[i, j] = np.sum(image[i:i+kh, j:j+kw] \* kernel)

return output

Backpropagation for CNN was handled manually. After computing the gradient of the loss w.r.t. output logits, we backpropagate through the fully connected layer and reshape the result to match the feature map. The convolutional kernel’s gradient is obtained by summing over all input patches multiplied by the backpropagated error at each location.

Despite being a single-filter model, the CNN performed meaningful feature extraction and classification. However, due to the nested-loop convolution implementation, it was significantly slower than the MLP.

**2.3 Loss Function and Logging**

Both models use categorical cross-entropy loss:

L=−∑yilog⁡(y^i)L = - \sum y\_i \log(\hat{y}\_i)

Loss computation was implemented with NumPy, ensuring numerical stability through log clipping. Logging was integrated at every critical stage: model initialization, forward passes, activations, loss computation, and backward passes. Logs were written to mlp\_training\_log.txt and cnn\_training\_log.txt to allow full traceability of the training process.

**3. Evaluation**

**Experimental Setup**

We tested both models on a macOS laptop (CPU-only, no GPU acceleration). The models were implemented in Python 3.9 and used only NumPy for all numerical operations. The MNIST dataset was loaded using torchvision. MLP training took around 1–2 minutes, while CNN training took approximately 15–20 minutes for 5 epochs due to the cost of nested-loop convolution.

**Results and Observations**

The MLP achieved excellent performance:

* Total training loss decreased from ~411 to ~68 across 10 epochs
* Final test accuracy: **95.6%**, consistent with expectations for a shallow MLP on MNIST

The CNN, although slower, showed a consistent learning trend:

* Training loss decreased from ~1084 to ~933
* Final test accuracy: **~60–65%**, which is reasonable for a single-filter CNN without pooling or padding

The performance gap reflects architectural differences and the limited representational power of a single 3×3 kernel. Nonetheless, both models demonstrated proper learning behavior. The log files confirmed the stability and correctness of all operations and made it easy to verify that forward and backward flows were executed as intended.

**4. Learning Outcomes**

This project was an in-depth exercise in manually designing and training neural networks. We went beyond just using models — we built every critical piece ourselves, from the first dot product to the last gradient update.

Through this, we gained a deep appreciation of how frameworks like PyTorch simplify training. For example, implementing the CNN’s convolution and its gradient required carefully indexing each spatial position and calculating per-pixel errors. Similarly, debugging MLP gradients reinforced our understanding of matrix calculus and dimensional consistency.

We learned about the importance of stable numerical computation, especially in the softmax function and loss calculation. We also built a logging system that not only facilitated debugging but also gave us visibility into how each layer processed inputs. It became clear how much goes on behind-the-scenes in real frameworks, and why such tooling is vital for debugging and reproducibility.

**Team Contributions and Reflections:**

* One member led the implementation of the MLP architecture and training pipeline.
* The other developed the CNN and handled complex gradient propagation.
* Both collaborated on debugging, writing utility functions, and refining the report.

**Challenges Faced:**

* Manually calculating and verifying CNN gradients without autograd
* Maintaining dimension alignment through all layers
* Training time for CNN due to lack of optimized ops

This experience gave us practical exposure to neural network internals and reinforced our ability to reason through algorithms, shape constraints, and edge cases. We are now more confident in our ability to work at both high-level abstraction and low-level system design in deep learning.