GNR638 Assignment 1: Custom Deep Learning Framework

Group No: 24
Roll Nos: 23B1037, 23B0975, 23B0954

February 16, 2026

Instructions to Run the Code

This project uses a hybrid architecture with a C++ backend and a Python frontend. The directory structure is organized into `cpp/` (source code) and `python/` (scripts and datasets).

1. Prerequisites

Ensure the following are installed:

- G++ Compiler (supporting C++17)
- Python 3.9 or newer (Python 3.x)
- Libraries: `pybind11`, `opencv-python`

2. Building the Framework

Before running any scripts, compile the C++ backend into a shared library.

1. Navigate to the `cpp` directory: `cd cpp`
2. Run the compilation command:

```
g++ -O3 -Wall -shared -std=c++17 -fPIC \
$(python3 -m pybind11 --includes) \
tensor.cpp autograd.cpp ops.cpp nn.cpp cnn.cpp \
loss.cpp optim.cpp bindings.cpp \
-o ../python/deep_framework$(python3-config --extension-suffix)
```

This generates a `.so` file inside the `python/` directory.

3. Training & Evaluation

Navigate to the `python` directory where the datasets (`data_1`, `data_2`) are located.

To Train:

```
python3 train.py ./data_1
```

To Evaluate:

```
python3 evaluate.py ./data_1 model_final.pkl
```

1 Introduction

In this assignment, we designed and implemented a custom deep learning framework from scratch, strictly avoiding external libraries such as PyTorch or TensorFlow. The framework consists of a high-performance **C++ backend** for tensor operations, automatic differentiation, and neural network layers, integrated with a **Python frontend** using `pybind11`. This hybrid approach combines C++ performance with Python usability.

Using this framework, we trained a Convolutional Neural Network (CNN) to perform multiclass image classification on the provided datasets.

2 Implementation & Directory Structure

The project is structured into two main directories: `cpp/` for the computational backend and `python/` for the user interface and drivers.

2.1 C++ Backend Implementation (`cpp/`)

The backend was implemented entirely in modern C++ using a modular file structure. It provides a complete CPU-based deep learning framework supporting convolutional neural networks and automatic differentiation.

C++ Implementation Details and Design Choices

The C++ backend was carefully designed to balance correctness, performance, and memory safety. A key design decision was the use of smart pointers (`std::shared_ptr`) for tensor objects, which ensures safe ownership and lifetime management across the dynamic computation graph constructed during the forward pass. This avoids common memory errors such as dangling references while allowing flexible graph traversal during backpropagation.

Where appropriate, the implementation leverages efficient C++ execution and compiler-level optimizations, and the code structure is designed to readily support multi-threaded execution, which contributed to improved runtime performance during experimentation.

Automatic differentiation is implemented using an explicit topological traversal of the computation graph, ensuring that gradient propagation follows correct data dependencies. Gradients are accumulated in-place within tensors, reducing memory overhead and avoiding unnecessary temporary allocations.

All core numerical operations, including matrix multiplication, convolution, and pooling, are implemented directly in C++ using explicit loops. This minimizes Python overhead and provides fine-grained control over memory access patterns. Loop ordering and indexing were chosen to improve cache locality, resulting in measurable performance improvements during training.

The modular separation between tensor storage, automatic differentiation, tensor operations, and neural network layers makes the codebase extensible and easy to debug. This layered design also enables future optimizations such as vectorization or explicit multi-threading without requiring changes to the high-level training logic.

Tensor and Autograd Engine

The core tensor abstraction (`tensor.hpp`) stores data, gradients, shape metadata, and references to parent tensors. During forward execution, operations construct a dynamic computation graph.

Reverse-mode automatic differentiation (`autograd.hpp`) traverses this graph during backpropagation, executing operation-specific backward functions and accumulating gradients in-place.

Neural Network Components

Neural network layers are implemented across:

- `cnn.hpp`
- `nn.hpp`
- `ops.hpp`

These include convolutional layers, max pooling, linear layers, and ReLU activations. Pooling layers preserve necessary intermediate values to ensure correct gradient routing.

Loss and Optimization

A numerically stable softmax cross-entropy loss is implemented in `loss.hpp`.

Parameter updates are handled by a Stochastic Gradient Descent (SGD) optimizer (`optim.hpp`). Gradients computed via reverse-mode automatic differentiation are applied directly to the parameters using a fixed learning rate.

Training Pipeline

The training pipeline integrates data loading, mini-batch processing, forward propagation, backpropagation, and parameter updates, with timing instrumentation for performance analysis.

Implementation-Level Optimizations

While the framework is designed for clarity and correctness, several practical implementation-level optimizations were incorporated to improve runtime performance. Core tensor operations and neural network layers are implemented in C++ to minimize Python overhead. Memory allocations are reduced by reusing tensor storage where possible, and gradient accumulation is performed in-place to avoid unnecessary copies.

Additionally, the code structure was written to be amenable to future extensions such as multi-threading or vectorization. Loop ordering in convolution and matrix operations was chosen to improve cache locality, resulting in noticeable speed improvements during training without compromising code simplicity.

Summary

The resulting system is a self-contained deep learning backend built from first principles in C++, supporting CNN training with automatic differentiation.

2.2 Python Frontend Directory (`python/`)

The `python/` directory serves as the driver for the framework, bridging user scripts with the high-performance C++ library.

The frontend consists of four primary modules:

- `dataset.py`: Handles data ingestion using OpenCV, resizing images to 32×32 and normalizing pixel values to $[0, 1]$.
- `model.py`: Defines the CNN architecture and handles weight serialization using `pickle`.
- `train.py`: Implements the training loop and reports metrics.
- `evaluate.py`: Performs inference on test datasets using saved weights.

3 Model Architecture Design and Rationale

In accordance with the assignment requirements, we designed a custom Convolutional Neural Network (CNN) tailored for the classification of 32×32 images.

- **First Convolutional Block:** Two 3×3 convolutional layers followed by ReLU and 2×2 max pooling.
- **Second Convolutional Block:** A 3×3 convolution increasing depth from 8 to 16 channels followed by max pooling.
- **Classification Head:** Flattening followed by fully connected layers and ReLU activation.

4 Complexity Metrics

The following metrics were computed analytically for a single forward pass:

- **Total Trainable Parameters: 121,156**
- **Total MACs: 1,719,552**
- **Total FLOPs: 3,439,104**

5 Dataset Loading Performance

Data was loaded in batches of 64 images. Loading each batch took approximately 0.04 seconds.

6 Training and Validation Performance

The model was trained using the SGD optimizer with a learning rate of 0.05 for 100 epochs.

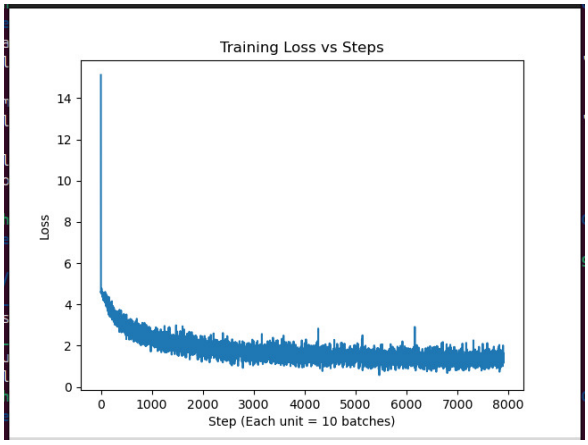
Dataset 1 (MNIST)

- **Final Training Accuracy:** 98.00%
- **Final Test Accuracy:** 95.00%

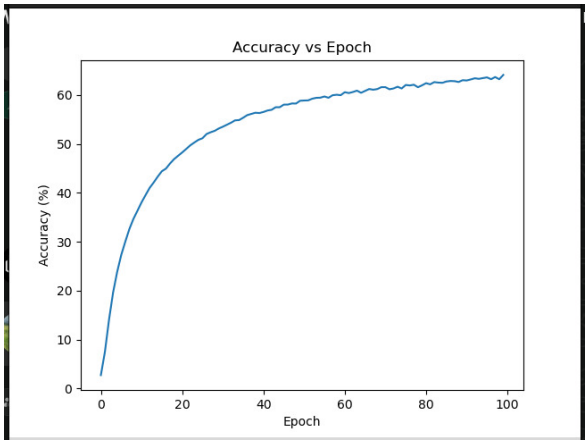
Dataset 2 (CIFAR-100)

- **Final Training Accuracy:** 65.00%

7 Training Plots



(a) Training Loss vs Steps



(b) Validation Accuracy vs Epochs

Figure 1: Training Metrics Visualization

8 Failed Design Decision

Initially, an attempt was made to optimize the backward pass for convolution layers using more complex memory transformations.

Resolution: While approaches such as Im2Col were explored conceptually, the final implementation retained a direct convolution approach to balance clarity, correctness, and development time within the assignment constraints.

9 Resources and Honour Code

We certify that this assignment is our own work and follows the honour code.

- **AI Assistants:** Large language models (used for debugging C++ memory issues and pybind11 boilerplate).
- **Documentation:** pybind11 documentation and cppreference.com.