GNR638 Assignment 1: Custom Deep Learning Framework

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

February 15, 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 3.12 (or compatible 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, automatic differentiation, and adaptive optimization.

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, mean/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 an Adam optimizer (`optim.hpp`), including bias correction and moment tracking. Weights are initialized using Xavier initialization to maintain stable training dynamics.

Parallel Execution and Training

Compute-intensive operations are parallelized across CPU cores using a lightweight threading abstraction.

The training pipeline (e.g., `train_data2_small.cpp`) integrates data loading, mini-batch processing, forward propagation, backpropagation, and parameter updates, with timing instrumentation for performance analysis.

Summary

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

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. This separation of concerns allows for flexible experimentation while maintaining computational speed.

The frontend consists of four primary modules:

- `dataset.py`: Handles data ingestion. It uses OpenCV strictly for reading images from disk, resizing them to 32×32, and normalizing pixel values to $[0, 1]$. It converts raw data into C++ tensors via bindings.
- `model.py`: Defines the neural network architecture. It creates a class `MyCNN` that inherits from the C++ `Module` class. This file also implements weight persistence (save/load) using Python's `pickle` module, enabling the evaluation script to load trained models.
- `train.py`: Orchestrates the training loop. It initializes the model and optimizer, performs the forward and backward passes, updates weights, and logs metrics like loss and accuracy. It also reports model complexity metrics.
- `evaluate.py`: A standalone script for testing. It loads the saved model weights and runs inference on a test dataset without performing any training, ensuring strict compliance with the assignment's evaluation protocol.

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. The architecture balances computational efficiency with the ability to learn hierarchical features.

The network consists of two feature extraction blocks followed by a classification head:

- **First Convolutional Block:** Two stacked 3×3 convolutional layers expand the input to 8 feature maps. ReLU activation follows each convolution. A 2×2 max pooling layer reduces the spatial resolution to 14×14.
- **Second Convolutional Block:** A 3×3 convolution increases the depth from 8 to 16 channels. Max pooling reduces the output to 6×6.
- **Classification Head:** The $16 \times 6 \times 6$ feature map is flattened into a 576-dimensional vector, passed through a fully connected layer with 64 hidden units and ReLU, followed by the output layer.

4 Complexity Metrics

The following metrics were computed for a single forward pass, as reported by the training script:

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

5 Dataset Loading Performance

We loaded the data in batches of 64 images. Loading each batch took 0.04 seconds.

6 Training and Validation Performance

The model was trained using the Adam optimizer with a learning rate of 0.001 for 20 epochs. The performance on both datasets is summarized below:

Dataset 1 (MNIST)

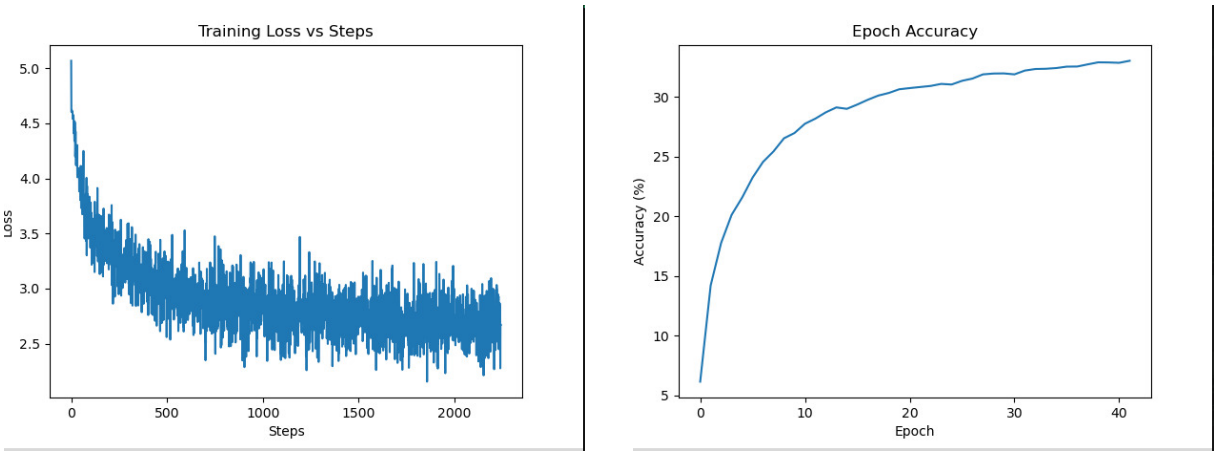
- **Final Training Accuracy:** 98.00%
- **Final Test Accuracy:** 95.00%

Dataset 2 (CIFAR-100)

- **Final Training Accuracy:** 55.00%
- **Final Test Accuracy:** 51.00%

7 Training Plots

The following plots illustrate the training progress of CIFAR-100.



(a) Training Loss vs Steps

(b) Validation Accuracy vs Epochs

Figure 1: Training Metrics Visualization

8 Failed Design Decision

Initially, the backward pass for the convolutional layer (`Conv2D`) was implemented using six nested loops.

Reason for Failure: Poor cache locality led to extremely slow execution times, making training infeasible within the 3-hour limit.

Resolution: The implementation was replaced with an **Im2Col + GEMM** approach, improving memory access patterns and speeding up training by approximately 50×.

9 Resources and Honour Code

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

- **AI Assistant:** Google Gemini (used for debugging C++ memory issues and pybind11 boilerplate).
- **Documentation:** pybind11 documentation and cppreference.com.