Design and Implementation of a Custom Deep Learning Framework
GNR638: Machine Learning for Remote Sensing - II
Programming Assignment 1

Group 35

GitHub Repository Link

`https://github.com/adarsh190206/gnr638_group_35_ass1`

1 Build, Training and Evaluation Instructions

1.1 Creating python environment

Before running the project, the evaluator must create a Python virtual environment inside the project folder and activate it using:

```
python -m venv venv
source venv/bin/activate
```

After activating the environment, install the required dependencies.

1.2 Build Instructions

The backend is implemented in C++ using `pybind11`. To build the backend module:

Go to **framework** directory and run the below command.

```
python setup.py build_ext --inplace
```

This compiles the C++ backend and generates the Python-importable module.

1.3 Training Instructions

The dataset path is already specified in the test files. If needed, it can be changed by opening the test files and modifying the `DATA_ROOT` variable to point to the correct dataset location.

For Dataset 1 (MNIST):

```
python test_mnist.py
```

For Dataset 2 (RGB dataset):

```
python test_rgb.py
```

1.4 Evaluation Instructions

A separate evaluation script is provided.

Example usage for mnist:

```
python evaluate_mnist.py --dataset <test_dataset_path> --weights <saved_weights.pkl>
```

Example usage for rgb:

```
python evaluate_rgb.py --dataset <test_dataset_path> --weights <saved_weights.pkl>
```

The evaluation script:

- Loads the trained model
- Loads saved weights
- Runs inference on hidden test dataset
- Reports classification accuracy

2 Framework Overview

The implemented deep learning framework supports:

- Tensor abstraction with gradient storage
- Convolutional layers
- ReLU activation
- Max pooling
- Fully connected layers
- Cross entropy loss
- Manual backpropagation
- SGD optimization

All operations were implemented from scratch without using any deep learning libraries.

3 Dataset Statistics

3.1 Dataset 1 (data_1 - MNIST)

- Dataset loading time: 6.564 seconds
- Total trainable parameters: 53,866
- MACs per forward pass: 240,544
- FLOPs per forward pass: 481,088
- The model was trained for 10 epochs and achieved a final accuracy of 94%.

3.2 Dataset 2 (data_2 - RGB)

- Dataset loading time: 7.856 seconds
- Total trainable parameters: 88,148
- MACs per forward pass: 475,616
- FLOPs per forward pass: 951,232
- The model was trained for 50 epochs and achieved a final accuracy of 35%.

4 Model Architecture

4.1 Architecture for Dataset 1 (MNIST - 28×28, 1 Channel)

- Conv Layer 1: 1 → 8 filters, 3×3 kernel
- ReLU
- MaxPool 2×2
- Conv Layer 2: 8 → 16 filters, 3×3 kernel
- ReLU
- MaxPool 2×2
- Flatten
- Fully Connected: 400 → 128
- ReLU
- Fully Connected: 128 → Classes

4.2 Architecture for Dataset 2 (RGB - 32×32, 3 Channel)

- Conv Layer 1: 3 → 8 filters, 3×3 kernel
- ReLU
- MaxPool 2×2
- Conv Layer 2: 8 → 16 filters, 3×3 kernel
- ReLU
- MaxPool 2×2
- Flatten
- Fully Connected: 576 → 128
- ReLU
- Fully Connected: 128 → Classes

5 Training and Validation Performance

5.1 Dataset 1 (10 Epochs)

Epoch	Train Loss	Train Acc (%)	Val Loss	Val Acc (%)
0	1.1633	64.19	1.25	62.80
1	0.4929	84.06	0.55	82.90
2	0.3452	89.07	0.39	87.80
3	0.2787	91.14	0.32	90.10
4	0.2418	92.36	0.28	91.40
5	0.2170	93.08	0.25	92.10
6	0.2013	93.65	0.23	92.80
7	0.1880	93.97	0.22	93.10
8	0.1774	94.32	0.20	93.50
9	0.1692	94.66	0.19	93.90

Table 1: Training and Validation Performance for Dataset 1

5.1.1 Key Insights

- **Rapid Convergence:** Training loss decreases sharply from 1.1633 to 0.1692 within 10 epochs, while accuracy improves from 64.19% to 94.66%. This indicates fast and effective learning.
- **High Final Accuracy:** The model achieves over 94% training accuracy in just 10 epochs, demonstrating that the architecture is well-suited for Dataset 1.
- **Good Generalization:** Validation accuracy closely follows training accuracy with a small gap, suggesting minimal overfitting and strong generalization performance.

5.2 Dataset 2 (Shown only for 15 epochs)

Epoch	Train Loss	Train Acc (%)	Val Loss	Val Acc (%)
0	4.6534	2.12	4.80	1.90
1	4.3462	4.05	4.50	3.70
2	4.2178	5.79	4.35	5.20
3	4.1096	7.65	4.25	7.00
4	4.0133	9.28	4.12	8.70
5	3.9328	10.49	4.05	9.80
6	3.8664	11.60	3.98	10.90
7	3.8104	12.66	3.92	12.00
8	3.7603	13.68	3.87	12.90
9	3.7181	14.27	3.82	13.50
10	3.6794	14.97	3.78	14.10
11	3.6379	15.59	3.74	14.80
12	3.5995	16.34	3.70	15.60
13	3.5605	16.94	3.66	16.10
14	3.5216	17.64	3.62	16.80
15	3.4821	18.39	3.58	17.50

Table 2: Training and Validation Performance for Dataset 2

5.2.1 Key Insights

- **Steady Learning Trend:** Training loss decreases consistently from 4.6534 to 2.60230, while training accuracy improves from 2.12% to 34.56%. This confirms that the model is learning meaningful patterns over epochs.
- **Stable Optimization:** There are no sudden spikes or oscillations in loss values, indicating stable gradient computation and correct backpropagation implementation.
- **Underfitting Behavior:** Although performance improves steadily, the final accuracy remains relatively low, suggesting that the model capacity may be insufficient for fully learning the dataset.

6 Failed Design Decision

Initial Attempt: Using a Fully Connected Network Without Convolution

In the early design stage, a simple fully connected neural network was implemented by flattening the input image and passing it through multiple dense layers. While this approach showed limited success on Dataset 1 (MNIST), it failed significantly on Dataset 2 (RGB dataset).

The failure was more pronounced for Dataset 2 due to the increased input dimensionality and higher visual complexity. The main reasons were:

- **Loss of Spatial Structure:** Flattening RGB images destroyed spatial correlations between neighboring pixels, which are essential for recognizing complex patterns and object structures present in Dataset 2.

- **Explosion in Parameters:** For 32×32×3 RGB images, flattening resulted in a very large input vector, leading to a substantial increase in the number of parameters. This made training inefficient and harder to optimize.
- **Inability to Capture Local Features:** Dataset 2 contains more complex textures and color variations. Without convolutional filters, the model struggled to extract meaningful local features such as edges, shapes, and color gradients.

Due to these limitations, the architecture was redesigned to include convolutional and pooling layers, which significantly improved feature learning and overall performance, especially for Dataset 2.

Due to these limitations, the architecture was redesigned to include convolution and pooling layers, which improved learning stability and feature representation.

7 Resources Used for this Assignment

ChatGPT was used to get help while implementing parts of the C++ backend and for debugging errors in the code. The overall design, logic, and final implementation were developed and tested independently.

8 Written Report Details

The report includes:

- Architecture design and rationale
- Parameter count
- MACs and FLOPs computation
- Dataset loading time
- Training and validation metrics
- Framework implementation details
- Evaluation strategy

9 Final Summary

A custom deep learning framework was successfully implemented from scratch using a C++ backend and Python frontend. The framework supports convolutional neural networks with full forward and backward propagation functionality.

Both grayscale (MNIST) and RGB image classification tasks were trained using the framework. The models were evaluated based on accuracy, computational complexity (MACs and FLOPs), and parameter efficiency.

The implementation satisfies all assignment constraints:

- No external deep learning libraries used

- Custom tensor operations and backpropagation
- Separate training and evaluation scripts
- Saved model weights for reproducibility