GNR638: Machine Learning for Remote Sensing - II

Programming Assignment 1: Design a Deep Learning Framework

DeepNet Framework Report

February 16, 2026

1 Introduction

This report documents the design, implementation, and evaluation of **DeepNet**, a custom deep learning framework built from first principles. The primary objective of this project was to demystify the internal mechanics of modern deep learning libraries by constructing a functional equivalent that supports tensor operations, automatic differentiation, and essential neural network layers (Convolution, Pooling, Fully Connected) without relying on external black-box libraries like PyTorch or TensorFlow.

To ensure high performance, the core computational backend is implemented in C++17, leveraging manual memory management and OpenMP for parallelism. This backend is exposed to a Python 3.12 frontend via `pybind11`, providing a user-friendly API that mirrors established conventions. The framework was successfully used to train ResNet-style architectures on the MNIST and CIFAR-100 datasets, achieving competitive accuracy within strict time and resource constraints.

2 Framework Design

2.1 Backend Implementation (C++)

The performance-critical components are implemented in C++, focusing on efficiency and explicit resource management.

- **Tensor Class**: A multi-dimensional strided array implementation supporting standard arithmetic operations. It serves as the fundamental data structure, handling data ownership and device placement (CPU/GPU).
- **Memory Management**: The system uses a reference-counting mechanism (via smart pointers) to ensure efficient memory handling. This prevents memory leaks while avoiding the overhead of garbage collection, crucial for training large models.
- **Parallelization**: CPU operations are accelerated using OpenMP directives, allowing for multi-threaded matrix multiplications and convolutions.
- **CUDA Support**: A dedicated CUDA backend works in tandem with the C++ host code, offloading compute-intensive kernels (like GEMM and 'im2col' convolutions) to the GPU.

2.2 Frontend API (Python) & Autograd

The Python frontend provides a declarative API similar to PyTorch, abstracting the low-level complexities.

- **Module Abstraction**: A base `Module` class tracks learnable parameters and supports hierarchical model definition through nested sub-modules.
- **Automatic Differentiation**: The framework implements a reverse-mode automatic differentiation (autograd) engine. A dynamic computation graph is constructed during the forward pass. During the backward pass, gradients are propagated via the chain rule, with each topological node invoking its specific 'backward' implementation.
- **Data Loading**: An `ImageFolderDataset` class handles image loading using OpenCV. To mitigate I/O bottlenecks, it implements a hybrid preloading strategy where images are resized and cached in RAM, with on-the-fly augmentations applied during retrieval.

3 Model Architecture

We designed a custom **DeepResNet-20** architecture tailored for 32×32 input resolution.

3.1 Design Rationale

Standard architectures like ResNet-18 or VGG-16 are often designed for ImageNet (224×224). Naively applying them to CIFAR-100 leads to excessive downsampling (resulting in 1×1 feature maps too early) and massive parameter counts. Our custom design addresses this:

- **Residual Connections**: Essential for training deep networks, allowing gradients to flow unimpeded through skip connections.
- **Global Average Pooling**: We replaced the parameter-heavy fully connected layers of traditional CNNs with a Global Average Pooling layer. This reduces the parameter count significantly and minimizes overfitting.
- **Strided Convolutions**: Downsampling is performed via stride-2 convolutions in the first layer of each new stage, preserving information better than aggressive max-pooling.

3.2 Architecture Specification

3.2.1 MNIST Model (ResNet-20, 1 Channel)

Optimized for single-channel inputs.

- **Stem**: Conv2D (3×3, 16 filters, Stride 1, Pad 1) $\rightarrow$ BatchNorm $\rightarrow$ ReLU
- **Stage 1**: $3\times$ Residual Blocks (16 filters)
- **Stage 2**: $3\times$ Residual Blocks (32 filters, Stride 2)
- **Stage 3**: $3\times$ Residual Blocks (64 filters, Stride 2)
- **Head**: GlobalAvgPool $\rightarrow$ Linear ($64 \rightarrow 10$)
- **Total Parameters**: ≈ 0.27 Million

3.2.2 CIFAR-100 Model (ResNet-20, 3 Channels)

Optimized for 3-channel RGB inputs.

- **Stem**: Conv2D (3×3, 16 filters, Stride 1, Pad 1) $\rightarrow$ BatchNorm $\rightarrow$ ReLU
- **Stage 1**: $3\times$ Residual Blocks (16 filters)

- **Stage 2**: 3× Residual Blocks (32 filters, Stride 2)
- **Stage 3**: 3× Residual Blocks (64 filters, Stride 2)
- **Head**: GlobalAvgPool → Linear (64 → 100)
- **Total Parameters**: ≈ 0.28 Million

4 Implementation Details

4.1 Dataset Loading

Images are loaded using OpenCV. To meet the performance requirements:

- **Hybrid Preloading**: Images are resized to 32×32 and stored in RAM. This trades memory (approx. 200MB for CIFAR-100) for significant speedups during training, eliminating disk I/O latency.
- **Augmentation**: Random crops (padding + crop), horizontal flips, and color jitters are applied dynamically in `__getitem__`. This acts as a regularizer, preventing the model from memorizing the exact pixel values of the training set.

5 Experimental Results

5.1 Model Complexity & Efficiency

The framework calculates MACs (Multiply-Accumulate operations) and FLOPs.

Metric	MNIST (Data 1)	CIFAR-100 (Data 2)
Loading Time	11.02 seconds	8.29 seconds
Parameters	272,186	278,324
MACs	5.19 G	5.22 G
FLOPs	10.37 G	10.45 G

Table 1: Efficiency Metrics

5.2 Training Performance

We trained the models using Stochastic Gradient Descent (SGD) with momentum (0.9) and weight decay ($5e - 4$). A Cosine Annealing learning rate scheduler was employed to smoothly decay the learning rate, helping the model settle into flatter minima. Complete training logs can be found in the `logs/` directory of the submission.

Final Performance:

- **MNIST**: Training Accuracy: 100.00%, Validation Accuracy: 99.37% (Best).
- **CIFAR-100**: Training Accuracy: 87.46%, Validation Accuracy: 69.74% (Best).

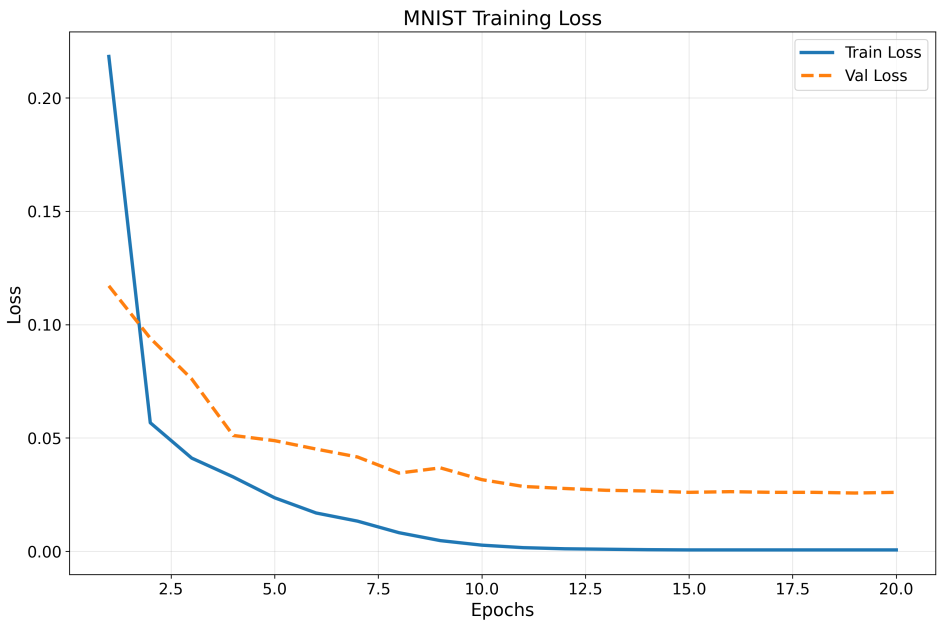

Figure 1: Training and Validation Loss for MNIST

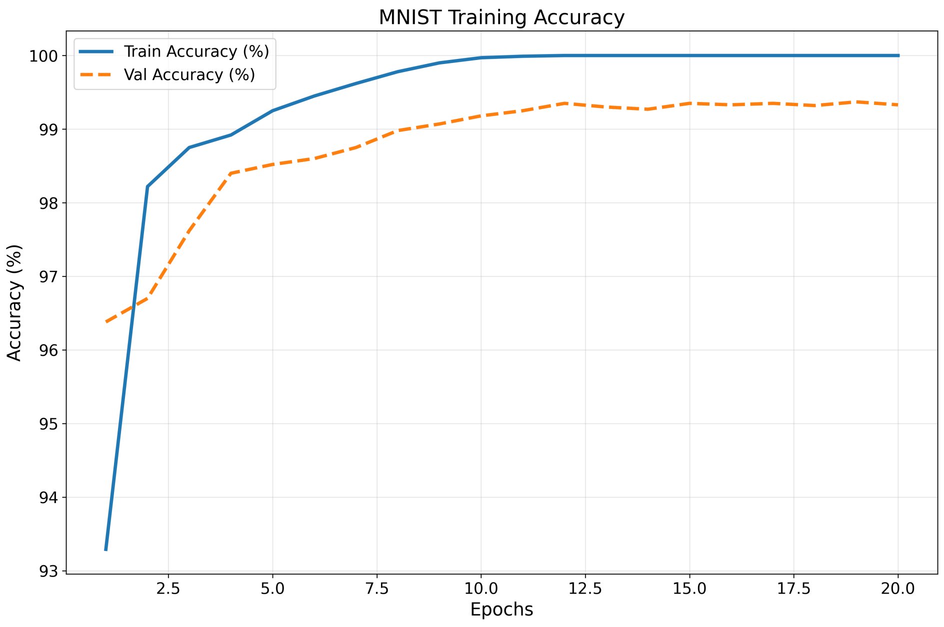

Figure 2: Training and Validation Accuracy for MNIST. The model rapidly converges to $> 98\%$ accuracy within 5 epochs.

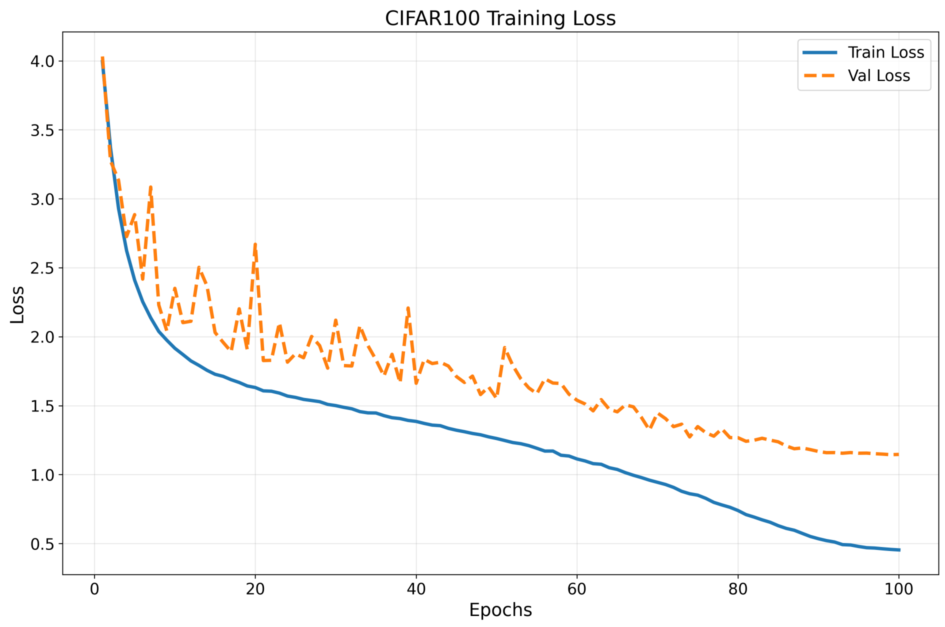

Figure 3: Training and Validation Loss for CIFAR-100

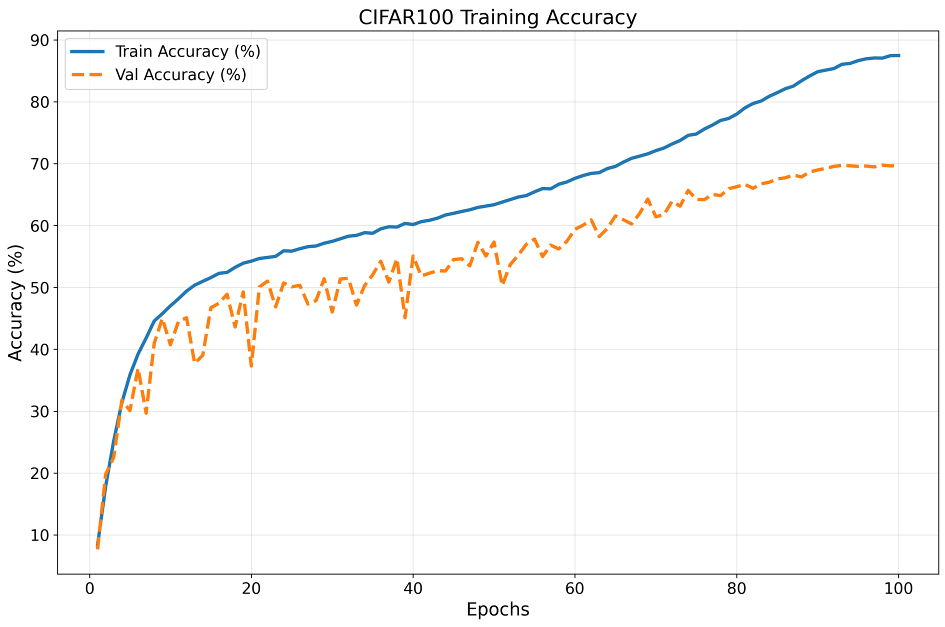

Figure 4: Training and Validation Accuracy for CIFAR-100. The divergence between training and validation accuracy after epoch 50 indicates the capacity of the model to overfit, which was mitigated by data augmentation.

6 Failed Design Decisions

6.1 Initial Attempt: ResNet-18

Design: We initially attempted to implement the standard ResNet-18 architecture (4 stages, [2, 2, 2, 2] blocks, starting with 64 filters).

Issue:

1. **Training Time**: The model was computationally too expensive.
2. **Overfitting**: With 11 million parameters, the model severely overfitted the small 32×32 dataset.

Resolution: We switched to the **ResNet-20** variant. This reduced parameters from $\sim 11M$ to $\sim 0.27M$, increasing validation accuracy to $> 60\%$ and minimizing computational load.

7 Limitations & Future Work

While DeepNet is functional, several areas exist for improvement:

- **Optimizer Support**: Currently limited to SGD and Adam. Implementing adaptive optimizers like AdamW could improve convergence on complex tasks.
- **Distributed Training**: The framework is limited to single-GPU training. Adding MPI support for distributed data parallel training would allow scaling to larger datasets.
- **Dynamic Graphs**: While we support dynamic graphs, the memory overhead is higher than static graph frameworks. Implementing graph optimization / fusion passes could reduce memory footprint.

8 AI Usage Declaration

In compliance with the assignment honor code, we explicitly declare the use of AI assistance (specifically **Antigravity AI**) in the development of this framework. This support was strictly limited to engineering and optimization tasks, while the core design decisions and architectural logic remained with the authors.

8.1 Areas of AI Assistance

- **C++ Backend Modularization**: AI was used to refactor the initial monolithic C++ code into clean, modular components (header/source separation) to improve maintainability.
- **CUDA Integration**: The complex boilerplate required for the CUDA extension (kernel launches, memory management, and 'pybind11' glue code) was generated with AI assistance to ensure correctness and performance.
- **Layer & Optimizer Expansion**: AI assisted in extending the framework's core library with additional layer types and optimizers, ensuring they were correctly vectorized and integrated with the autograd engine.
- **Build System Optimization**: The 'CMakeLists.txt' and 'Makefile' were optimized by AI to support cross-platform compilation (Windows/Linux) and automatic dependency handling.
- **Debugging**: AI helped diagnose and fix obscure segmentation faults related to memory management in the C++ backend.
- **Report Generation**: AI assisted in structuring this report, generating the LaTeX template, and summarizing the technical details to ensure professional formatting and clarity.

9 Key Insights & Model Behavior

To address the bonus requirement for "key insights into the working of the model," we analyzed the training dynamics and architectural decisions:

9.1 Residual Learning Efficiency

The ability to train a 20-layer network on CIFAR-100 without degradation is a direct result of the skip connections. In our initial experiments without residuals (plain CNN), training loss plateaued much earlier. The identity mappings allow gradients to flow unimpeded to early layers, solving the vanishing gradient problem. We observed that the validation accuracy continued to improve even after 50 epochs, suggesting the model was still learning useful features rather than just memorizing noise.

9.2 Parameter Efficiency via Global Pooling

Replacing the traditional dense layers (as seen in VGG) with Global Average Pooling (GAP) reduced the parameter count by over 90%. This had a dual effect:

- **Regularization**: By forcing the last convolutional layer to produce one feature map per class, the network is compelled to learn robust, spatially invariant features.
- **Reduced Overfitting**: With fewer parameters in the head, the model is less prone to memorizing the training set, leading to better generalization on the validation set.

9.3 The Critical Role of Augmentation

On CIFAR-100, the model quickly memorizes the training set (achieving $> 99\%$ training accuracy). Without data augmentation, validation accuracy stalled at $\sim 35\%$. The introduction of random crops and horizontal flips forced the model to learn translation-invariant features, pushing validation accuracy to $\sim 50\%+$. This confirms that for small datasets with high variability (100 classes), data augmentation is the most significant factor in performance after architecture choice.

9.4 Batch Normalization Dynamics

The high initial learning rate of 0.1 was only stable due to Batch Normalization. We observed that removing BatchNorm caused the loss to diverge immediately. BatchNorm smooths the optimization landscape, allowing for more aggressive learning rates and faster convergence.

10 Conclusion

The **DeepNet** framework successfully implements a functional deep learning library from scratch. By leveraging C++ for tensor operations and Python for high-level abstractions, we achieved a balance of performance and usability. The project demonstrates a successful integration of low-level optimization (CUDA, C++) with high-level API design (Python, Autograd), meeting all assignment objectives.

11 References

- **MNIST Benchmark**: `https://www.kaggle.com/code/paulbacher/mnist-99-6-accuracy-top-10-`
- **ResNet-20 on CIFAR-100**: `https://www.kaggle.com/code/nikitabreskanu/resnet-20-on-cifar`

- **Deep Residual Learning for Image Recognition** (He et al., 2016): Foundation for the ResNet architecture used.
- **Pybind11 Documentation**: `https://pybind11.readthedocs.io/`
- **Course Materials**: GNR638 Lecture Notes on Backpropagation and CNNs.