Neural Network Framework from Scratch

Valerio Melissa
Machine Learning 2025
Sapienza University of Rome
melissa.2139153@studenti.uniroma1.it

January 7, 2026

Abstract

This project presents an implementation of a neural network framework built entirely from scratch in CUDA C++ with dual CPU and GPU implementations. Currently the framwork allows the creation of a multi layer perceptron with fixable dimension, activation functions and loss function. To prove the implementation correctness the framework have been tested on the MNIST digit recognition dataset. Code available at: `https://github.com/Valerio3240264/nn-scratch`.

1 Introduction

This project originated from a desire for greater low-level control than Python typically offers. Rather than adopting an existing C++ framework, I chose to build a neural network framework from scratch. The result is not comparable to established frameworks like PyTorch or TensorFlow, but it has been a great learning experience, expecially about CUDA kernels, weights initialization and backpropagation.

The most important characteristic of this project is the dual implementation of the framework, one for the CPU and one for the GPU that can be choosen by the user when declaring the mlp.

2 Related Work

Backpropagation. The project sticks to classic backpropagation.

Xavier/Glorot initialization. Xavier/Glorot initialization [1] is used on both CPU and GPU to keep variance stable across layer where the activation function is linear or an hyperbolic tangent.

He initialization. He initialization [2] is used on both CPU and GPU to keep variance stable across layer where the activation function is a ReLU.

Activations and losses. The MLP supports ReLU, Tanh, Linear and softmax, paired with MSE for regression and cross-entropy for classification.

Stochastic Gradient Descent. Updates use plain SGD (vectorized CUDA update kernel on GPU, straightforward loop on CPU), keeping the focus on correctness of gradients rather than optimizer features.

3 Method

3.1 Computational Graph Architecture

The framework implements a directed acyclic graph (DAG) where nodes represent computations and edges represent data flow.

Here how the MLP graph is represented:

$$\begin{tikzcd}\text{Layer} \arrow[r] & \text{Layer} \arrow[r, dashed] & \text{Layer} \arrow[r] & \text{Loss}\end{tikzcd}$$

Here how the layers graphs are represented:

$$\begin{tikzcd}\text{Input} \arrow[r] & \text{Weights} \arrow[r] & \text{Activation}\end{tikzcd}$$

3.2 Core Components

-

Input. Holds the input of the layer and the partial gradients of the layer.

Weight. Holds the layer weights and biases with different initialization methods (Xavier/Glorot and He).

Performs the forward pass $y = Wx + b$ that will be used to compute the output of the layer from the activation component. Performs the backward pass updating the weights and biases gradients and propagating the partial gradients to the input component.

Activation. This component performs element-wise nonlinearities (ReLU, Tanh, Linear) plus Softmax on the output of the weights component.

Layer. This component links the input, weight and activation components together and performs the forward and backward pass of the layer.

Mlp. This component initializes the layers and the loss component, links them together and performsthe forward and backward pass of the MLP.

Loss. This component computes the loss of the layer and starts the backward pass that will progagate to the first layer.

3.3 Gpu Implementation

In the kernels for the GPU version have been used the following techniques: - Vectorization

- Tiling
- Shared memory
- Reduction
- Coalescing

3.4 How does the network work?

The network receives an input, which is passed to the first layer. This first layer sends it to the weight component inside the layer, where the transformation $y = Wx + b$ is performed. The result is then passed to the activation function component, which applies the activation function. After activation, the output is sent back to the layer component, which transforms it again into an input for the next layer. This process continues through all layers until the last layer.

After the last layer, the output is passed as input to the loss component, which stores it. When the method `mlp.compute_loss(target)` is called, the loss component computes the loss and starts the backward pass that will propagate back to the first layer.

Unfortunately, I wasn't able to implement batch processing, which could be realized similarly to the current implementation but would require additional time and was not the primary goal of the project.

4 Experimental Results

4.1 MNIST Training (CPU vs GPU)

To evaluate the correctness of the implementation, I wrote two different tests: one for the CPU and one for the GPU. Each test is an MLP with 3 layers—256, 128, and 10 neurons—with ReLU activation functions in the first two layers and softmax in the last layer, using the cross-entropy loss function.

Training setup: 5 epochs, batch size 100, 32k training samples, and 10k validation samples.

Running these test files, I obtained the following results:

Device	Epochs	Batch	Val Acc	Time/epoch	Total Time
CPU	5	100	96.90	18.64	93.18
GPU	5	100	96.83	5.48	27.38

Table 1: Measured MNIST results on 5 epochs.

5 Discussion and Conclusions

As expected, the GPU version is much faster than the CPU version, achieving comparable accuracy to the CPU version. These results may vary depending on the hardware and the computer settings. However, they are a good indication of the correctness of the implementation, which consistently achieves around 95% accuracy on the validation set.

My primary goal for this project was to have fun and learn about CUDA kernels, training neural networks, and their inner workings. I hope to continue this project in the future, adding more features and improving the performance.

I hope you enjoyed the project and the work behind it.

References

[1] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.

[2] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” *Proceedings of the ieee international conference on computer vision*, pp. 1026–1034, 2015.