Title: High Performance Neural Network from Scratch using C++ and OpenMP

Author: Anshika Gaur
Course: COIS 4350 – High Performance Computing
Instructor: Dr. Brian Srivastava
Date: April 6th, 2025

1. Abstract

This project explores how high-performance computing (HPC) principles can be applied to a basic neural network architecture written entirely in C++. The focus is on optimizing training performance using OpenMP, a widely adopted parallel programming model for shared memory systems. By implementing a feedforward neural network from scratch and applying multithreaded parallelism to the training loop, we observe substantial improvements in runtime efficiency. The model was tested using the Wine Quality dataset, and parallelization yielded a speedup of nearly 2.7x. This project demonstrates how even simple machine learning models can benefit from parallel computing techniques and offers foundational insights for scaling to larger, more complex architectures in the future

2. Dataset & Preprocessing

We used the **Wine Quality (Red)** dataset from the UCI Machine Learning Repository, which contains various chemical properties of red wine samples along with quality ratings.

- **Input Features:** 11 numerical variables (e.g., fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, etc.)
- **Target:** Wine quality (normalized to a range of 0 to 1)

Custom C++ code was written to parse the CSV file. During preprocessing:

- Rows with malformed or missing values were skipped.
- Input features were normalized.
- The target column (quality) was scaled to fall within the range [0, 1], enabling smooth convergence using the sigmoid activation function.

3. Neural Network Design

The neural network built for this project is a basic **feedforward** architecture with one hidden layer. It was intentionally kept small for simplicity and to focus on the impact of parallelism.

- **Input Layer:** 11 nodes
- **Hidden Layer:** 16 nodes with sigmoid activation
- **Output Layer:** 1 node with sigmoid activation (for regression-style output)
- **Loss Function:** Mean Squared Error (MSE)
- **Training Method:** Batch Gradient Descent
- **Epochs:** 100

- **Implementation Language:** C++ (no libraries used for training)

Weights and biases were initialized using small random values. All matrix operations and forward/backward propagation logic were implemented manually using raw arrays.

4. Why C++ and OpenMP?

C++ was chosen for its speed and control over memory management, making it ideal for performance-focused projects. By avoiding external ML libraries, we gained a deeper understanding of how neural networks operate at the algorithmic level.

OpenMP allows developers to parallelize loops using simple compiler directives. Since gradient updates and loss calculations in neural networks often involve operations over large batches, they are highly parallelizable.

Applying OpenMP to the training loop provided a hands-on experience with thread-level parallelism and allowed us to benchmark the performance improvements on a multi-core processor.

5. Parallelization with OpenMP

The most compute-intensive part of training is the loop that processes each training sample during forward and backward propagation. Using OpenMP, we parallelized this section as follows:

```
#pragma omp parallel for reduction(+:loss)
for (int i = 0; i < num_samples; ++i) {
    // Forward pass
    // Backward pass
    // Loss accumulation
}
```

OpenMP automatically distributes the iterations of this loop across multiple CPU cores. The reduction clause ensures that the loss value is safely accumulated without race conditions.

Performance was measured using std::chrono timers.

6. Results & Analysis

Performance Table:

Version	Training Time (ms)	Notes
Without OpenMP	395 ms	Single-threaded
With OpenMP	147 ms	Multithreaded with omp

Speedup Calculation:

Speedup = 395 / 147 ≈ 2.69x

This shows a substantial improvement in training efficiency with parallelism. While the neural network is simple and small, OpenMP still significantly cut down the training time, highlighting how even lightweight models can benefit from HPC techniques.

7. Reflection & Learnings

This project offered me a deeper appreciation for how neural networks function internally, especially when stripped of any high-level frameworks. Implementing every layer manually, from the activation function to the weight update logic, significantly strengthened my understanding of backpropagation.

From an HPC perspective, I learned to identify parts of an algorithm that are amenable to parallelism. Applying OpenMP showed me how multi-threaded performance gains can be achieved with minimal code changes, and how to measure such gains rigorously.

I also gained experience in writing efficient, readable C++ code, managing memory, and debugging performance bottlenecks. This experience has encouraged me to explore further optimizations, such as:

- Adding more hidden layers
- Comparing OpenMP to MPI in a distributed setting
- Deploying the model via WebAssembly or integrating into a front-end app

8. Limitations & Future Work

- The model is shallow and may not generalize well to more complex datasets.
- No learning rate tuning or early stopping was implemented.
- Only one form of parallelism (OpenMP) was explored.

In future iterations, I aim to:

- Implement GPU-based training with CUDA
- Compare training performance with TensorFlow or PyTorch
- Visualize loss convergence and model accuracy over epochs

8. Files & Submission

- neural_net.cpp: Full implementation of the neural network
- wine.csv: Dataset used
- README.md: Instructions and summary
- report.docx: This document (updated)

GitHub Repo: https://github.com/gauranshika29/hpc-neural-network
Portfolio Site: https://hpc-portfolio-site.vercel.app

End of Report