Simulation of an Artificial Neural Network (MLP) – Parallelization using OpenMP and MPI

Project Title

Improving a C Implementation of a Feedforward Neural Network

1. Project Overview

This project aims to improve the performance, efficiency, and scalability of an existing C implementation of a simple feedforward neural network (multi-layer perceptron). The original version uses full-batch gradient descent, static learning rates, and manual memory management without optimization. Several enhancements are proposed in terms of **memory safety**, **computational performance**, and **parallelization**.

2. Objectives

(a) Memory Management and Debugging

- Use **Valgrind (memcheck)** to detect and fix memory leaks caused by missing `free()` calls or improper allocations.
- Analyze memory usage patterns and ensure every allocation in the model (`z1`, `a1`, `z2`, `probs`, gradients, etc.) has a corresponding deallocation.
- Document all memory allocation and deallocation operations for better code maintainability.

(b) Performance Profiling

- Use **Valgrind Callgrind** to identify computational hotspots in matrix multiplications (`matmul`), backpropagation loops, and bias updates.
- Visualize call graphs with **KCachegrind** to guide optimization efforts.

(c) Training Optimization

- Replace **batch gradient descent** with **mini-batch gradient descent**:
  - Split the dataset into smaller batches (e.g., 32 or 64 samples).
  - Compute forward and backward passes for each batch.
  - Update weights after processing each batch rather than the entire dataset.
- This approach typically improves convergence stability and training speed.

(d) Dynamic Learning Rate (Annealing Schedule)

- Implement a **learning rate decay** mechanism:

$$\eta_t = \frac{\eta_0}{1 + k \cdot t}$$

  where η_0 is the initial learning rate, k is a decay coefficient, and t is the iteration index.

- Explore alternative strategies such as exponential decay or step-based schedules.
- Compare convergence curves and accuracy across different schedules.

(e) Activation Functions

- The hidden layer currently uses the `tanh` activation function.
- Experiment with:
  - **ReLU**: $f(x) = \max(0, x)$
  - **Sigmoid**: $f(x) = \frac{1}{1+e^{-x}}$
  - **Leaky ReLU**: $f(x) = \max(0.01x, x)$
- Modify the corresponding derivative in backpropagation for each activation function.

(f) Parallelization

OpenMP Acceleration:

- Parallelize nested loops in matrix multiplications and gradient calculations.
- Explore the use of **OpenMP tasks** for independent batch computations.
- Measure performance scaling with different thread counts.

MPI Distribution:

- Extend the model to support **data parallelism** across multiple processes.
- Each process handles a subset of the training data (mini-batches).
- Synchronize weight updates using collective operations (`MPI_Allreduce`).
- Benchmark speedup and scaling efficiency.

3. Expected Outcomes

- A **memory-safe**, **faster**, and **scalable** neural network implementation in C.
- Profiling reports identifying computational bottlenecks.
- Comparative analysis of activation functions and learning rate schedules.
- OpenMP and MPI versions demonstrating multi-core and distributed scalability.

3. Code Structure

- **generate_moon.py** — Generates synthetic datasets (`data_X.txt`, `data_y.txt`) for training and testing.
- **main.c** — Entry point; loads data, initializes model, and runs training.
- **model.c / model.h** — Core neural network logic (forward pass, backpropagation, weight updates).
- **utils.c / utils.h** — Helper routines (matrix multiplication, softmax, bias addition, data loading).
- **check_output.py** — Loads trained weights and visualizes decision boundaries or classification results.
- **Makefile** — Automates compilation for sequential, OpenMP, and MPI modes.

Directory Tree Example:

```
mlp_project/
   generate_moon.py
   check_output.py
   main.c
   model.c
   model.h
   utils.c
   utils.h
   Makefile
   data/
    data_X.txt
    data_y.txt
   output/
    W1.txt
    W2.txt
    b1.txt
    b2.txt
```

```
#generate data_X and data_y
python3 generate_moon.py

#compile
make

#run
./mpl

#plot output
python3 check_output.py
```

4. Deliverables

- Refactored and optimized C source code (`model.c`, `utils.c`, `main.c`)
- Valgrind and Callgrind profiling reports

- Mini-batch gradient descent implementation with adjustable hyperparameters
- Increased dataset size in `generate_moon.py`
- OpenMP and MPI parallel versions with comparative benchmarks
- Documentation and performance analysis report