Parallel Neural Network
Multiprocessor Systems

DOCUMENTATION

November 2025

1 Introduction

As part of the *Multiprocessor Systems* course, a Multilayer Perceptron (MLP) was trained and evaluated on the MNIST dataset for the task of Arabic numeral recognition. The matrix operations required for both Feed-Forward and Backpropagation were parallelized, and the effects on training and testing time were measured while varying the number of network layers and processor cores.

This document provides a comprehensive overview of the entire process - from the construction of the neural network, through the parallelization of its matrix computations, to the performance analysis based on the selected variables - culminating with a discussion of the results and final conclusions.

2 Architecture

A Multilayer Perceptron is a net of computational units (neurons) divided into three parts: [input], [hidden layers] and [output].

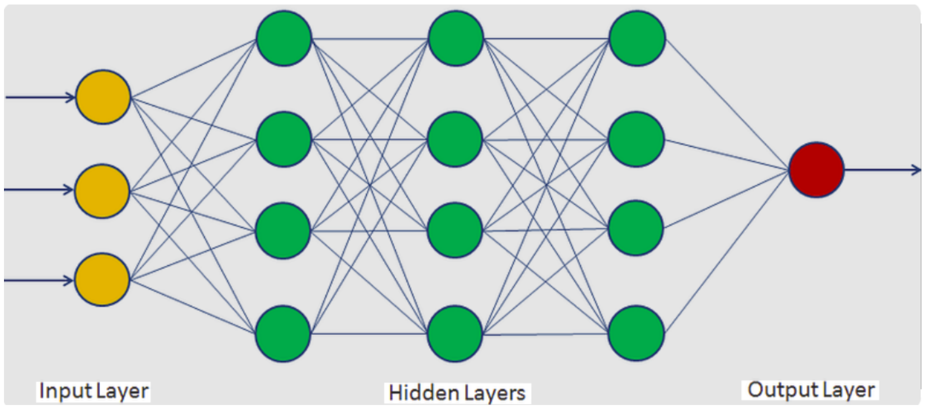

Figure 1: Multilayer Perceptron

At its core, the MLP is trying to approximate a function $f : R^n \to R^m$. Given data (like images $\to$ digit labels), it adjusts internal parameters (weights and biases) and applies a non-linear function (the activation function) so that its outputs match expected outputs. Without the activation function the network would act like one big linear equation, and thus be unable to learn complex patterns.

$$(1) \quad \overline{z}^{(l)} = \overline{W}^{(l)} \times \overline{x}^{(l-1)} + \overline{b}^{(l)} \qquad (2) \quad \overline{a}^{(l)} = \sigma(\overline{z}^{(l)})$$

where σ is the activation function.

In equation (1) above, we observe how the weight $\overline{W}$ and bias $\overline{b}$ matrices determine the "pre-activation" output neuron matrix $\overline{z}$. Equation (2), in turn, illustrates how applying the activation function to $\overline{z}$ produces the activated neurons $\overline{a}$, which are then passed forward as inputs $\overline{x}$ to the next layer.

These two equations are applied layer by layer, starting from the input and continuing forward until we obtain the output of the final layer. This procedure is known as Feed-forward, and it represents one of the two fundamental operations in a Multilayer Perceptron.

The second operation is Backpropagation. Once the Feed-forward pass produces an output, we compare it to the expected (target) value and compute the resulting error (equation (3)). This error is then used to adjust the weights and biases that contributed to the output (equations (6) and (8)), allowing the network to learn from its mistakes. The error is calculated differently for the output and hidden layer. For the hidden layers we calculate the error by using the error of the previous layer (equation (4)).

$$(3) \quad \overline{\delta}^{(L)} = (\overline{a}^{(L)} - \overline{y}) \odot f'(\overline{z}^{(L)}) \qquad (4) \quad \overline{\delta}^{(l)} = (\left(\overline{W}^{(l+1)}\right)^T \times \overline{\delta}^{(l+1)}) \odot f'(\overline{z}^{(l)})$$

$$(5) \quad \frac{\partial E}{\partial W^{(l)}} = \delta^{(l)} \times \left(a^{(l-1)}\right)^T \qquad (6) \quad \Delta W^{(l)} = -\alpha \cdot \frac{\partial E}{\partial W^{(l)}}$$

$$(7) \quad \frac{\partial E}{\partial b^{(l)}} = \delta^{(l)} \qquad (8) \quad \Delta b^{(l)} = -\alpha \cdot \frac{\partial E}{\partial b^{(l)}}$$

where L is the output layer index,
$\overline{y}$ is the target output,
$\overline{\delta}$ is the error matrix,
f' is the derivative of the activation function,
$\odot$ is element-wise multiplication,
α is the learning rate,
E is the error.

As for the choice of the activation itself, for the hidden layers the Rectifier Linear Unit (ReLu) function (equation (9)) was used as the activation function while for the output layer the Softmax function (equation (10)) was used.

$$(9) \quad \mathrm{ReLU}(x) = \max(0, x) \qquad (10) \quad \mathrm{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

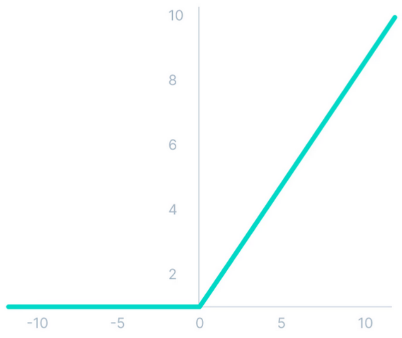

Figure 2: ReLu function

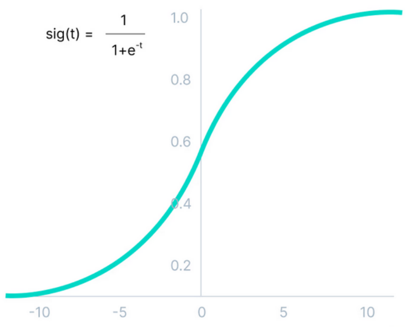

Figure 3: Softmax function

The ReLU activation function (figure 2) introduces nonlinearity while remaining computationally simple, since it has no exponents or divisions. It effectively acts as a filter by allowing only positive values to pass through and setting all negative inputs to zero. This reduces the number of neurons needed to calculate the values of the neurons in the next layer during the feed-forward pass, allowing for compiler optimizations and helping speed up training. By not squashing positive values into a narrow range like most activation functions, ReLU also avoids the vanishing gradient problem during backpropagation, which improves training efficiency.

However, ReLU is not without drawbacks. Neurons can permanently "die" if they receive negative inputs too often, causing their gradients to become zero and preventing further learning. Additionally, because ReLU outputs are unbounded and always non-negative, it can lead to unstable activations or biased gradient updates unless properly managed through initialization or normalization.

The softmax function (figure 3) is used to convert a vector of raw output scores into a probability distribution. It works by exponentiating each score and normalizing the result so that all outputs fall between 0 and 1 and sum to 1. The drawback when using this function is that because it involves exponentiation, it can be sensitive to large input magnitudes, potentially causing numerical instability unless stabilized with techniques like subtracting the maximum input value.

The derivatives of these two functions used during Backpropagation are fairly simple to compute and are shown in equations (11) and (12). For softmax the derivative is a bit more complicated than shown in equation (12), the one shown in the equation is actually the derivative of the softmax function and the cross-entropy loss function combined since it simplifies calculation and matches our use case.

$$(11) \quad \text{ReLU}'(x) = \begin{cases} 0 & \text{if } x < 0, \\ 1 & \text{if } x > 0 \end{cases} \qquad (12) \quad \text{softmax'}(x) = 1$$

To mitigate the potential issue of gradient explosion introduced through the activation functions, we use the uniform ranges of He (equations (13)) and Xavier (equation (14)) for weight initialization. These ranges are carefully chosen to keep the variance of activations roughly constant across layers. For ReLU, He initialization uses a range that will compensate for the fact that ReLU zeroes out negative inputs. For softmax, Xavier initialization uses a range that doesn't leave room for an initialization where the weight values are disparagingly different, leading to no learning because of loss of input data, but also isn't so small as to leave the weight values more or less the same, leading to a loss of influence on the output from the weight values themselves.

$$(13) \quad W \sim U\Big(-\sqrt{\frac{6}{n_{\text{in}}}}, \sqrt{\frac{6}{n_{\text{in}}}}\Big) \qquad (14) \quad W \sim U\Big(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\Big)$$

where n_{in} is the number of input neurons,
n_{out} is the number of output neurons.

3 Parallelization

Training the neural network is done by picking a sample and applying feed-forward and backpropagation so we can compute the weights and biases. This is done many times so that these two characteristics of the network are refined until they can accurately capture the number represented by a 28x28 pixel image input once tested. The key takeaway is that this refinement process links the new values of the weights and biases to the old ones, so there is an inherent dependency between the iterations of the for loop body that executes the code. The situation is similar for feed-forward and backpropagation, calculating value n is done by using value [n-1] and/or [n+1]. This means that executing training or the two phases, feed-forward and backpropagation, in parallel without any modification to the workflow of the code (ie batch training) is not possible.

However, the matrix operations used by feed-forward and backpropagation carry no loop dependencies and are used repeatedly during training, making them the ideal target for parallelization. Operations like matrix addition, multiplication, transposition and element-wise function application are all typically implemented using two nested for loops that iterate over matrix rows and columns:

```
Matrix transpose() {
    Matrix output(_cols, _rows);
    for (int x = 0; x < _rows; x++) {
        for (int y = 0; y < _cols; y++) {
            output.at(y, x) = at(x, y);
        }
    }
    return output;
}
```

Listing 1: matrix transposition

```
Matrix multiply(Matrix&& target) {
    assert(target._rows = = _cols);
    Matrix output(_rows, target._cols);
    for (int x = 0; x < output._rows; x++) {
        for (int y = 0; y < output._cols; y++) {
            T result = T();
            for (int k = 0; k < _cols; k++)
                result += at(x, k) * target.at(k, y);
            output.at(x, y) = result;
        }
    }
    return output;
}
```

Listing 2: matrix multiplication

```
    Matrix applyFunction(function<T(const T&)> func) {
        Matrix output(_rows, _cols);
        for (int x = 0; x < output._rows; x++) {
            for (int y = 0; y < output._cols; y++) {
                output.at(x, y) = func(at(x, y));
            }
        }
        return output;
    }
```

Listing 3: applying activation functions

The core idea of the parallelization strategy is the same for all the mentioned operations: the iteration space of the nested loops is divided among multiple threads, where each thread processes a distinct subset of rows, allowing multiple matrix elements to be computed simultaneously:

```
Matrix transpose() {
    Matrix output(_cols, _rows);
    #pragma omp parallel for num_threads(omp_threads)
       collapse(2)
    for (int x = 0; x < _rows; x++) {
        for (int y = 0; y < _cols; y++) {
            output.at(y, x) = at(x, y);
        }
    }
    return output;
}
```

Listing 4: example of parallelizing matrix transposition

Parallelization is implemented using threads via the OpenMP library. The choice of OpenMP over MPI, which achieves parallelism through separate processes rather than threads, was largely arbitrary. If a justification is needed, it lies in the fact that one of the parallelized operations involves summing matrix elements, a task that is more naturally expressed using shared-memory threading. However, this consideration is not decisive, as that operation could just as easily have been executed sequentially without affecting the overall design.

As discussed earlier, most functions are implemented using nested loops. Although parallel programs are often expected to outperform their sequential counterparts, in practice the use of threads introduces overhead, such as thread creation, scheduling, and synchronization. When the amount of data being processed is small, this overhead can dominate the execution time and reduce or even negate any performance gains from parallelization. For larger data sizes, the relative cost of this overhead becomes negligible. Therefore, to obtain meaningful performance results for our relatively small neural network (input size = 786), we apply loop collapsing to combine nested loops into a single larger iteration space and parallelize that combined loop, rather than parallelizing only the outer loop. This effectively increases the amount of work available for parallel execution by parallelizing over an N×M space instead of just N and is represented in listing 4 by the *collapse(2)* clause in the pragma.

4 Results

The inputs to the program include the neural network topology and dimensions, the paths to the training and test datasets, and the number of processes or threads used for execution. In our experiments, we vary the network size and the number of execution threads to examine their effect on training time. The primary objectives are to demonstrate that parallel implementations achieve measurable speedup when the neural network is ”sufficiently large”, and that this speedup decreases as the network size becomes smaller. We will also show that the change in network size doesn't have a decisive impact on network precision.

The results will be depicted using tables, where each table corresponds to a different network topology listed in the caption using the following notation: **[input : layer 1 dim : . . . : layer N dim : output]**

Table 1: Training time and speed-up for neural network [786:512:256:128:10]

Num of threads	Training time [s]	Speed-up	Precision [%]
1	50.071	1.00	89.23
2	32.2821	1.55	
4	19.3453	2.59	
8	16.0741	3.11	

Table 2: Training time and speed-up for neural network [786:256:128:64:10]

Num of threads	Training time [s]	Speed-up	Precision [%]
1	16.4427	1.00	86.74
2	9.38719	1.75	
4	7.05949	2.33	
8	7.1395	2.30*	

Table 3: Training time and speed-up for neural network [786:256:64:10]

Num of threads	Training time [s]	Speed-up	Precision [%]
1	12.4463	1.00	88.26
2	11.6225	1.07	
4	8.14092	1.53	
8	6.13998	2.03	

Table 4: Training time and speed-up for neural network [786:128:64:10]

Num of threads	Training time [s]	Speed-up	Precision [%]
1	6.8231	1.00	88.31
2	5.41727	1.26	
4	4.56017	1.50	
8	4.88325	1.39*	

The speed-up results marked in the table with an asterisk for the 8 thread version of the program are smaller than the results we have for the 4 thread version. The explanation for that could be the hardware limitation. The program is run on an intel i7 10th generation processor which possesses 8 threads in total. Using all of them could be influencing the result as this program isn't the only process running on the processor. This idea could be supported by the results we get when using 6 or 7 threads, making sure to use layer dimensions that are divisible by one of the two numbers as well as by four and eight so that the load is balanced (see table 5 bellow).

Table 5: Training time and speed-up for neural network [786:120:60:10]

Num of threads	Training time [s]	Speed-up	Precision [%]
1	8.99365	1	87.68
2	5.91038	1.52	
4	5.19014	1.73	
6	5.00726	1.80	
8	5.28303	1.70	

Another notable thing is the difference in speed-up when looking at the 4 thread version used for each of the different neural networks. It is getting smaller as the network topology and dimensions get smaller. This is the influence of the overhead we mentioned.

We can thus conclude that the network parallelization successfully sped-up the training process without hindering network functionality.