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].

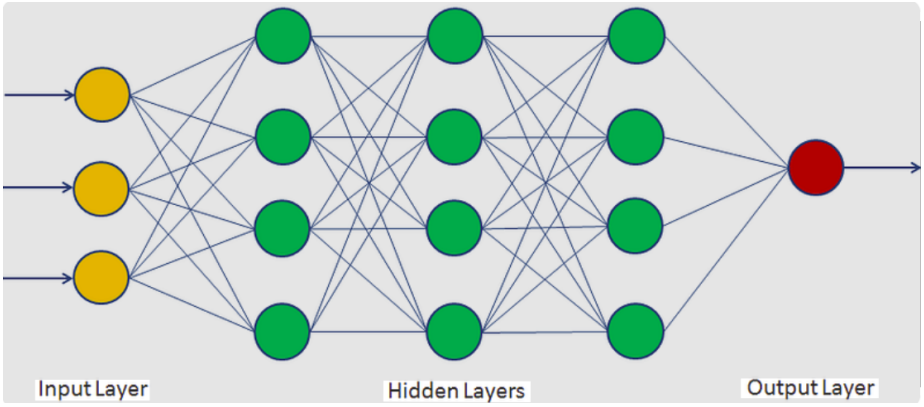

Figure 1: Multilayer Perceptron

At its core, the MLP is trying to approximate a function $f : R^n \to R^m$. Given data (like images → 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.

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

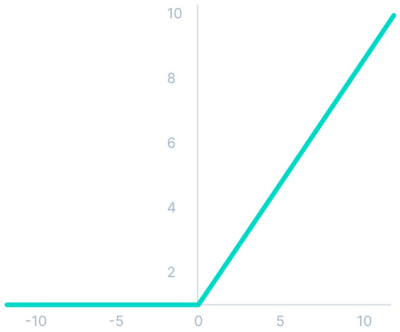

Figure 2: ReLu function

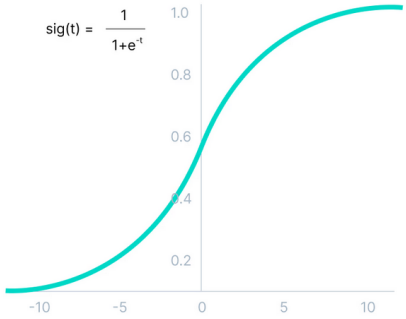

Figure 3: Softmax function

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.

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

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

Feed-forward and backpropagation are executed repeatedly during training, so focusing on parallelizing the operations used inside these phases can lead to significant speed-ups. In the feed-forward phase, the dominant computations are matrix multiplication, addition, and the application of activation functions to individual matrix elements, while in backpropagation similar operations appear along with matrix transposition. All of these operations are typically implemented using two nested for loops that iterate over matrix rows and columns:

```
1 Matrix transpose() {
2     Matrix output(_cols, _rows);
3     for (int x = 0; x < _rows; x++) {
4         for (int y = 0; y < _cols; y++) {
5             output.at(y, x) = at(x, y);
6         }
7     }
8     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

Since there are no loop dependencies, meaning that each iteration operates on independent data, these loops are well suited for parallel execution.

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.

4 Results

//should explain how the num of layers, initialization and num cycles influences the results theoretically //show results we got and compare the two - theory and practise