The ITB Journal

Volume 2 | Issue 2 Article 3

2001

A Review of Parallel Mappings for Feed Forward Neural Networks using the Backpropagation Learning Algorithm

Stephen Sheridan

A Review of Parallel Mappings for Feed Forward Neural Networks using the Backpropagation Learning Algorithm

Stephen Sheridan

School of Informatics and Engineering, ITB

1.1 Introduction

The Back-Propagation (BP) Neural Network (NN) is probably the most well known of all neural networks. Many mappings of the BP NN have been implemented for both special purpose and general-purpose computers. This mapping process can fall into one of two categories: *heuristic* mapping and *algorithmic* mapping. Heuristic mapping concentrates on the architecture and behaviour of the network as it trains whereas algorithmic mapping concentrates on the parallelization of the learning algorithm. Mappings in the heuristic category tend to take a trial and error approach based on the understanding of the network and of the target machine. In comparison mappings in the algorithmic category tend to take a more theoretical approach to the parallelization process. A number of heuristic mapping schemes exist for BP networks so it is worthwhile to investigate their strengths and weaknesses.

1.2 Useful terminology

The following is a brief introduction to some useful terminology that can be used in describing the different parallel implementations of BP networks.

Training Set

Consists of a number of training patterns, each given by an input vector and the corresponding output vector.

Network Size

The Network Size for a network of N_i input units, N_h hidden units, and N_o output units, can be written as N_i x N_h x N_o.

Training Iteration

Denotes one complete presentation of the training data.

Weight Update Strategy

Three different approaches can be used:

- *Learning by pattern (lbp)*: updates the weights after each training pattern has been presented.
- *Learning by block (lbb)*: updates the weights after a subset of the training data has been presented.
- *Learning by epoch (lbe)*: updates the weights after each training iteration.

Weight update interval

The number of training patterns that are presented between weight updates is termed μ. For *lbp*, μ = 1, whereas for *lbe* μ = *P*, where *P* is the number of training patterns in the training set.

1.3 Possible parallel implementations for the BP algorithm

Research into the BP network has revealed three possible parallel implementations as described below.

Training set parallelism

This approach splits the training set across the processing elements. Each element has a local copy of the complete weight matrix and accumulates weight changes for the given input patterns. The weights can be updated using *lbe/lbb*.

Pipelining

This approach allows the training patterns to be "pipelined" between the layers of the network. This means that the output and hidden layers are computed on different processors. So while the output layer calculates error values for the present pattern the input and hidden layers processors can process the next pattern. Pipelining requires a delayed weight update scheme, *lbb* or *lbe*.

Node parallelism

Node parallelism computes the neurons within a layer in parallel (Neuron Parallelism). Furthermore the computation within each node can also run in parallel. This method is also referred to as *synapse* or *weight* parallelism, *Nordström* [1].

2.1 Partitioning computation for each type of BP Parallelism

This section describes ways in which each type of BP parallelism can be distributed across a number of processors.

2.2 Training Set Parallelism

Training set parallelism is often referred to as data set parallelism as the training set data is distributed across a number of processors. An example is this type of distribution can be seen in Figure 2.1.

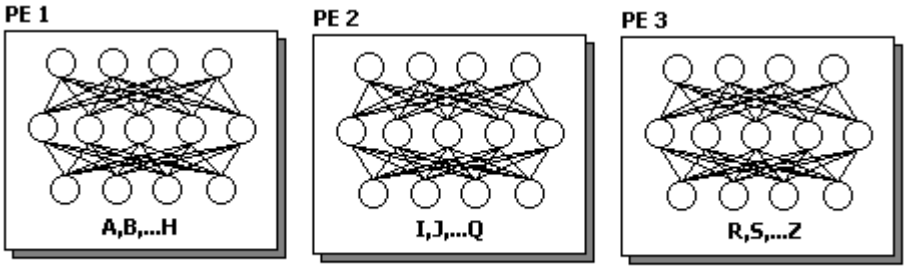

Figure 2.1. Training set parallelism for english alphabet

Each processing element (PE) has a local copy of the complete weight matrix and makes weights changes based on its range of input patterns. A global weight change operation is performed when each copy of the network has processed its current input pattern. This ensures that the weights remain consistent across each copy of the network. Therefore the global weight change operation must employ *lbb* or *lbe.* Each local copy of the weight matrix is updated by summing the weight change values for each PE.

So for example:

PE1 processes the letter A, and computes weight change $PE1\Delta w_i$
PE2 processes the letter I, and computes weight change $PE2\Delta w_i$
PE3 processes the letter R, and computes weight change $PE3\Delta w_i$

where $_i$ is the current training set iteration for that copy of the network.

Therefore each local weight matrix must be updated with a weight change value of

$$\Delta w = \sum_{j=0}^{i=0} PE_{j+1}\Delta_{i+1}$$

where (0>=j<= *Num_Processors*) and (0>=i<=*Training_set_size*).

2.3 Pipelining

In pipelining the computation for each layer is carried out by a separate PE. Figure 2.2 shows a pipelining example for a network with one hidden layer. NOTE: A separate PE is not required for the input layer as it merely presents patterns and does little computation.

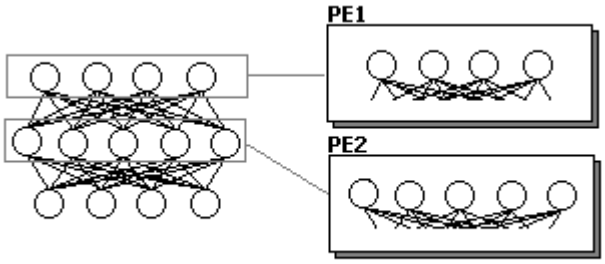

Figure 2.2. Mapping weight matrices for pipelining

Using this approach processing interleaves between the forward and backward passes of the BP learning algorithm. Figure 2.3 shows how this interleaving occurs.

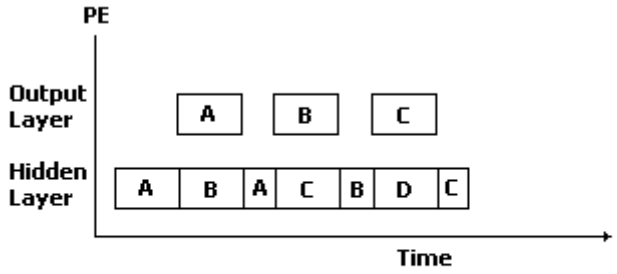

Figure 2.3. Interleaving of forward and backward pass in pipelined BP learning

So for example Figure 2.3 represents the following processing sequence:

1. Hidden layer computes output for letter A
2. Output layer reads values for letter A and computes error values
3. Hidden layer concurrently computes values for the next input pattern, B.
4. Hidden layer reads weight change for letter A and both processors accumulate weight change values for A.
5. Output layer reads values for letter B and computes error values.
6. Hidden layer concurrently computes values for the next input pattern, C.

…

2.4 Node Parallelism

Node parallelism can be split into two subclasses, neuron parallelism, and synapse parallelism.

2.4.1 Neuron Parallelism (Vertical Slicing)

Using this approach all incoming weights to one hidden and one output neuron are mapped to a PE. This vertical slicing corresponds to storing one row of the weight matrix in each PE. A matrix-vector product can be used to calculate the output of the neurons:

$$\begin{bmatrix} y_I,1 \\ y_I,2 \\ y_I,3 \end{bmatrix} = \begin{bmatrix} w_I 1,1 & w_I 1,2 & w_I 1,3 \\ w_I 2,1 & w_I 2,2 & w_I 2,3 \\ w_I 3,1 & w_I 3,2 & w_I 3,3 \end{bmatrix} \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix}$$

Figure 2.4 shows how a network with one hidden layer containing three nodes can be mapped onto three different PE's.

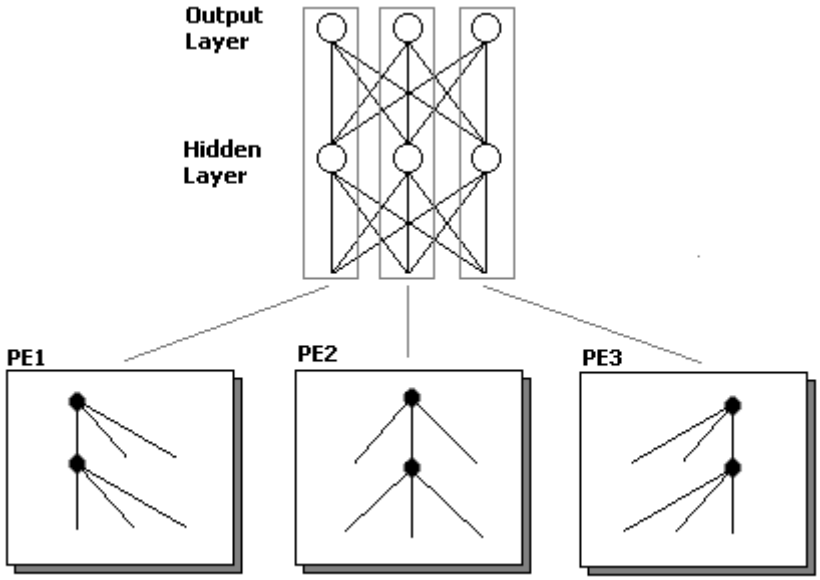

Figure 2.4. Neuron Parallelism (Vertical Slicing)

Each PE computes the value of a hidden node. Then each PE exchanges its value over a communications channel, and continues to calculate a value to be passed to the output layer. Once the output layer has received a complete set of inputs from the hidden layer it can calculate its error weight changes that are used to update the hidden layer weights. The output layer error can be calculated using a vector-matrix product:

$$\begin{bmatrix} \varepsilon' h,1 \middle| \varepsilon' h,2 \middle| \varepsilon' h,3 \end{bmatrix} = \begin{bmatrix} \varepsilon' o,1 \\ \varepsilon' o,2 \\ \varepsilon' o,3 \end{bmatrix} \begin{bmatrix} wo,11 & wo,12 & wo,13 \\ wo,21 & wo,22 & wo,23 \\ wo,31 & wo,32 & wo,33 \end{bmatrix}$$

As the weight matrix is stored in row order the error must be calculated by summing partial products. Some networks actually store a copy of the weight matrix in column order so that the summation is optimised for the backward pass. Weight updates can be duplicated [2,3] or communicated [4] to the other weight matrix. Communication of weight updates has been shown to be faster [4].

2.4.2 Synapse Parallelism

Synapse parallelism takes the opposite approach to *Neuron Parallelism* in that it takes columns of nodes and maps them onto separate PE's. In this case each PE computes a partial sum of a neurons output as shown in Figure 2.5. Synapse parallelism uses a more fined grained approach as each PE must broadcast its partial result to all other PE's before the next layer can compute.

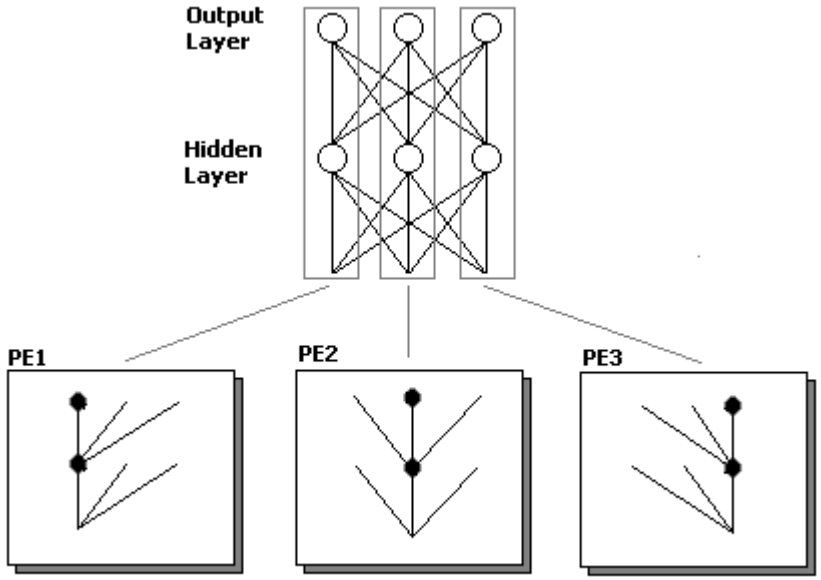

Figure 2.5. Synapse Parallelism

Synapse parallelism does have one advantage over neuron parallelism as the hidden layer error can be computed without communication:

$$\left[\delta' h,1 \middle| \delta' h,2 \middle| \delta' h,3\right] = \begin{bmatrix} \delta' o,1 \\ \delta' o,2 \\ \delta' o,3 \end{bmatrix} \left[\begin{array}{c|c|c} wo,11 & wo,12 & wo,13 \\ wo,21 & wo,22 & wo,23 \\ wo,31 & wo,32 & wo,33 \end{array}\right]$$

2.5 Concluding remarks

Each implementation has inherent limitations given by the following:

- Training set parallelism: The number of patterns in the training set
- Pipelining: The number of weights layers
- Neuron parallelism: The number of hidden units and output units
- Synapse parallelism: The number of input units and hidden units

These limitations indicate the maximum amount of PE's that can be assigned to each implementation. Only a few networks and training sets will run optimally on fixed parallel mappings. What should be considered is this: What degree of parallelism should be included and how many PE's should be assigned to each of them in order to reduce the total training time?

References

[1] T. Nordstrom and B. Svenson, "Using and designing massively parallel computers for artificial neural networks", "Journal of Parallel and Distributed Computing, vol. 14, p260-285, March 1992.

[2] H. Yoon, J. H. Nang, and S. Maeng, "Parallel simulation of multilayered neural networks on distributed-memory multiprocessors", "Microprocessor and Microprgramming, vol. 29, p185-195, 1990.

[3] E. Kerckhoffs, F. Wedman, and E. Frietman, "Speeding up backpropagation training on a hypercube computer", "Neurocomputing, vol. 4, p43-63, 1992.

[4] U. Muller, B. Baumle, P. Kohler, A. Gunzinger, and W. Guggenbuhl, "Achieving supercomputer performance for neural net simulation with an array of digital signal processors", "IEEE Micro, p55-65, October 1992.