A Systolic Array Implementation of a Dynamic Sequential Neural Network for Pattern Recognition

R.S.Shadafan & M.Niranjan

December 15, 1993

Cambridge University Engineering Department,
Technical Report CUED /F-INFENG/TR.157

Abstract

Recently we have developed a sequential algorithm for designing a multi-layer perceptron classifier [1, 2]. Our approach, called Sequential Input Space Partitioning (SISP) algorithm, results in a one pass algorithm and a growing network. We exploit the fact that class boundary constructed by an MLP classifier is piecewise linear and hence the contribution of each hidden hidden unit to the final decision is essentially local . We have shown that, in a number of benchmark classification problems, the algorithm achieves performances similar to conventional batch methods of training. We have also argued that the sequential design has an indirect computational advantage. This computational advantage comes from the fact that the algorithm sees each data item only once, hence the feasibility of pipelining the training procedures in a true parallel architecture. In this paper, we show how this one pass algorithm can be pipelined and realised by a systolic array implementation.

The idea is to exploit the fact that the locations of boundary segments are determined by solving localised classification problems. Training is achieved by updating local covariances using the Recursive Least Squares (RLS) algorithm. The algorithm is sequential in the sense that training examples are passed only once, and the network will learn and/or expand at the arrival of each example. The major advantage in this sequential scheme is the feasibility of pipelining the training procedures in a true parallel architecture. In this paper, we present a systolic array implementation of the SISP algorithm.

Contents

1 Introduction

There is a considerable interest in the neural networks community in learning algorithms that also adjust the architecture of the network. These algorithms are motivated by the need for avoiding empirical guesses about the network sizes, and produces a more precise network architecture to fit a particular problem, in accordance with its complexity, and are usually based on some heuristic that measure the complexity of the data.

These approaches can be divided into two categories. **Constructive** algorithms start from small network and grow during training. The work of Mezard (*et al.* in the *tiling algorithm* [4], Frean in the *upstart algorithm* [5], Ash [8] and Azimi-Sadjadi *et al.* [9] in *node creation in backpropagation*, Wynne-Jones in *Node Splitting* [3], and Fahlman *et al.* in the *cascade-correlation architecture* [10] all (including our work) involved growing Multi Layer Perceptrons (MLP's) . Whereas Platt's *Resource Allocating Network (RAN)* [7] and Kadirkamanathan *et. al* in their *dynamic network* [6] involved Radial Basis Function (RBF) networks. **Pruning** algorithms start large and reduce during training to more suitable smaller size as in the work of Mozer *et. al.* in their *skeletonization* [11] procedure and Le Cun *et al.* in the *Optimal brain damage* [12]. Alpadyn in his *GAL* [13] algorithm showed a possible strategy for a network performing both growing and pruning.

Our approach to a constructive algorithm is based on the observation that class boundary constructed by MLP classifier is piecewise linear, with every piece is introduced by a single node. In other words, training a hidden node should only effect a local part of the final decision boundary. As every example is passed to network, the network will only train the node which produces the segment of the piecewise boundary local to that example. Failing to associate a node to a certain example, the network will grow, by creating a new node, to encounter for the new arrival and at the same time stays consistent with previously passed examples. The new node will add a new segment boundary close to this example. At the end of the training session, the input space will be partitioned by segments of hyperplanes positioned according to the data items that lie close to them.

Another important feature in our approach is that, unlike iterative algorithms, training and creating process is achieved sequentially, in the sense that adaptation is carried out after every example, and examples are passed only once. These two ideas combined concluded in the development of the Sequential Input Space Partitioning (SISP) algorithm [1, 2].

Sequential, one pass algorithms have an indirect computational advantage. This advantage comes from the fact that each data item is presented to the network only once. However, in order to capture as much information from the presented data we do an increased amount of computation at each presentation. Such algorithms already found their way in the industries. A recent example is the Intel's chip implementing a sequential RBF network similar to the RAN [7]. In this paper, we show a possible hardware implementation for an MLP network based on the SISP algorithm. We will explore the parallelism and the pipelinability of the various SISP computations. This leads to a systolic array architecture that may be implemented in VLSI. We will also present a simulation of the SISP systolic array architecture on a vowel classification problem.

2 Sequential Input Space Partitioning (SISP) algorithm

As the name suggests, the SISP algorithm uses hyperplanes, produced by linear perceptrons, to partition the input space of every layer in the system, in a sequential way. We look at each layer as an entity which has its own input space established by previous outputs. It constitutes a group of nodes, each node maps a hyperplane onto the input space, which is positioned according to its parameters. The SISP algorithm sequentially introduces and/or modifies these hyperplanes, to partition the input space of every layer in the system, according to their **local** effects as a constituent segment of the total classification boundary.

The core of the SISP algorithm is based on sequential training where inputs are passed only once through a layer to successfully partition its input space. As each item of data arrives, the algorithm will decide whether to train the existing hyperplanes or to create a new one to accommodate the new example in the right partition of the input space, without causing any inconsistency with previous examples. The decision whether to train or to create a node is based on three heuristic criteria which measure **linear separability, remoteness,** and **locality** [1, 2].

When a current example is not **novel** according to these heuristic criteria, we train a **local** node with which the example is associated. Due to the locality of the problem, it is adequate to have a linear learning rule to update the parameters of the node. To achieve sequential training, nodes are trained using the Recursive Least Squares (RLS) method [14]. As a result, during training local covariances are being accumulated. On the other hand, if the system fails to associate the example to a local node or training causes any form of linear non-separability (denoted as *unrelaxed training*), a new node will be introduced and trained to correctly classify this example within its local neighbourhood. Successful training in this case, requires the node to inherit information about its locality from the nearest local node. Hence, every node is provided with some extra knowledge updated with every training cycle and is ready to share it with any newly created node.

At the beginning, the classifier will start with one layer containing only one node. As examples are passing through this layer, the SISP algorithm will be applied. If after any example, the layer develops into more than one node, a new layer with one node is created. The same will be done to any subsequent layers, and we always add a new layer with a single node if the final layer expanded to more than one node. This scheme will ensure that the machine will always have only one output, and helps to set a sensible criterion when adding more layers to the system.

The machine as a whole is a 2-class classifier. For problems that have $r > 2$ classes, r machines can be built, with each machine classifies only one class against all the other $r-1$ classes. In a 2-class system, it is enough to have one output with two states (0 or 1) to denote the two classes. With an r-class problem, the resultant system will have r outputs, each output denotes one class.

2.1 Recursive Node Training

We cast the problem of single node training as the Recursive Least Square (RLS) solution of the linear perceptron parameters $\mathbf{w}$ for a set of N training data $\mathbf{x}_i,\ t_i,\quad i = 1, ..., N,$

where $\mathbf{x}_i \in \mathcal{R}^{p+1}$ (including a bias element set to 1) and t_i are real values known as targets. The recursive relation is reduced to [1]:

$$\mathbf{w}_i = B_i^{-1} \ G_i, \tag{1}$$

where B_i & $G_i$can be computed recursively as:

$$\begin{aligned} B_i &= B_{i-1} + \mathbf{x}_i \ \mathbf{x}_i^t, & (2)\\ G_i &= G_{i-1} + \mathbf{x}_i \ t_i, & (3) \end{aligned}$$

and using the matrix-inversion lemma [14], B^{-1} can be recursively computed from:

$$\begin{aligned} B_i^{-1} &= B_{i-1}^{-1} - \frac{B_{i-1}^{-1} \ \mathbf{x}_i \ \mathbf{x}_i^t \ B_{i-1}^{-1}}{(1 + \mathbf{x}_i^t \ B_{i-1}^{-1} \ \mathbf{x}_i)}, & (4)\\ \text{or} & \\ B_i^{-1} &= B_{i-1}^{-1} - \mathbf{k}_i \ \mathbf{x}_i^t \ B_{i-1}^{-1}, & (5) \end{aligned}$$

where $\mathbf{k}$ is called the gain vector [14].

In order to facilitate the computations for simpler hardware implementation, we multiply equation 5 by equation 3, and compare it with equation 1 to get

$$\begin{aligned} \mathbf{w}_i &= \mathbf{w}_{i-1} + [B_{i-1}^{-1} \ \mathbf{x}_i^t - \mathbf{k}_i \ \mathbf{x}_i^t \ B_{i-1}^{-1} \ \mathbf{x}_i] \ t_i - \mathbf{k}_i \ \mathbf{x}_i^t \ \mathbf{w}_{i-1}, \\ &= \mathbf{w}_{i-1} + \mathbf{k}_i \ t_i - \mathbf{k}_i \ \mathbf{x}_i^t \ \mathbf{w}_{i-1}, & (6) \end{aligned}$$

Let $o_i = \mathbf{x}_i^t \ \mathbf{w}_{i-1}$, and the error $e_i = t_i - o_i$, then equation 6 will reduce to

$$\mathbf{w}_i = \mathbf{w}_{i-1} + \mathbf{k}_i \ e_i. \tag{7}$$

Equation 7 implies that we only need to keep and update $\mathbf{w}$, B^{-1} (to update $\mathbf{k}$), and the error e_i, to achieve the least square error solution in a recursive manner. However, as mentioned earlier, each node should also contain more information about its locality, necessary when creating new nodes. Such information is quantified in $\mathbf{w}_A, B_A^{-1}$ and $\mathbf{w}_B, B_B^{-1}$, where the suffixes A, B correspond to classes A & B. These quantities are still updated by equations 5 & 6, but only applied for the class whose member is currently presented. In addition, each node should also contain the number of examples from each class n_A, n_B associated with this node. At the end of a training cycle, the nodes compute their new outputs Y:

$$y_{i,j} = f(\mathbf{x}_i^t \ \mathbf{w}_{i,j}), \tag{8}$$

where j denotes node j in the layer, and $f(.)$ is a sigmoid function given by:

$$f(\alpha) = \frac{1}{1 + \exp(-\alpha)} \tag{9}$$

2.2 Criteria Computation

As mentioned earlier, the assessment of the degree of associating an example to a node depends on three heuristic criteria which the node need to compute before the SISP algorithm decides whether to train or to create. These criteria measures **linear separability**, bf Remoteness and **locality** [2].

2.2.1 Linear Separability

This criterion provides a quantitative measure of the linear separability of the current example to a node. Since hyperplanes are planed to solve simple and local linearly separable problems, training will only be favoured if this criterion indicates high degree of linear separability. This criterion is based on the correlation between the actual linear outputs and their corresponding linear targets (the quantity $T^t\ X\ \mathbf{w}$ which is equal to $G^t\ B^{-1}\ G$), normalised with the highest correlation possible which is the case when $T = \ X\ \mathbf{w}$. The recursive formulae is given by:

$$Z_i = \frac{z_i}{zn_i} = \frac{G_i^t\ B_i^{-1}\ G_i}{T^t\ T}, \tag{10}$$

substituting with equations 3 & 4 for G_i & B_i^{-1}, we obtain the following recursive formulae for z:

$$z_i = z_{i-1} + 2\ t_i\ o_i + t_i^2\ a_i - \frac{(o_i + t_i\ a_i)^2}{1+a}, \tag{11}$$

where scalar $a_i = \mathbf{x}_i^T\ B_{i-1}^{-1}\ \mathbf{x}_i$. The normalising factor zn is simply updated by

$$zn_i = zn_{i-1} + t_i^2. \tag{12}$$

2.2.2 Remoteness

To avoid the cases where examples are linearly separable but remote from the local boundary, we introduce the remoteness criterion to calculate how far is this example from the hyperplane under inspection.

A new version of this criterion is developed to suit the systolic array design and to make use of quantities already calculated by the RLS algorithm. The idea is based on the squared Mahalanobis distance [20, 21]

$$d_\Sigma^2 = (\hat{x} - u)^T\ \Sigma^{-1}\ (\hat{x} - u), \tag{13}$$

where $\Sigma = E[\hat{X}\hat{X}^T] - uu^T$ is the covariance matrix, u is the mean of $\hat{X}$ ($\hat{X}^T = [\hat{x}_1, ..., \hat{x}_N]$), and $\hat{x}_i$ is $\mathbf{x}$ excluding the bias term. Notice that equation 13 represents the general form of a hyperellipsoid [23]. In fact, Mahalanobis distances are hyperellipsoids around the mean u and trace the contours of the points which have constant normal distribution density [20]. If we assume normal distribution of the input data, we can use Mahalanobis distances as a measure of how remote any example is to the centre of its cluster. Consider

$$B = \ XX^T \ = N\ \begin{pmatrix} E[\hat{X}\hat{X}^T] & u \\ u^T & 1 \end{pmatrix},$$

Using matrix partitioning to find the inverse of a matrix [22], we can calculate B^{-1} as:

$$B^{-1} = \frac{1}{N}\ \begin{pmatrix} [E[\hat{X}\hat{X}^T] - uu^T]^{-1} & -[E[\hat{X}\hat{X}^T]]^{-1}\ u\ [1 - u^T[E[\hat{X}\hat{X}^T]]^{-1}u]^{-1} \\ -u[E[\hat{X}\hat{X}^T] - uu^T]^{-1} & [1 - u^T[E[\hat{X}\hat{X}^T]]^{-1}u]^{-1} \end{pmatrix},$$

Substituting $E[\hat{X}\hat{X}^T]$ with $(\Sigma + uu^T)$, and $[E[\hat{X}\hat{X}^T]]^{-1} = \Sigma - \frac{\Sigma\ u\ u^T\ \Sigma}{(1+u^T\ \Sigma\ u)}$ by the matrix inversion lemma [14], B^{-1} will reduce to:

$$B^{-1} = \frac{1}{N} \begin{pmatrix} \Sigma^{-1} & -\Sigma^{-1}u \\ -u^T\Sigma^{-1} & 1 + u^T\ \Sigma^{-1}\ u \end{pmatrix},$$

hence, the quantity $\mathbf{x}^T\ B^{-1}\ \mathbf{x}$ becomes:

$$\begin{aligned} \mathbf{x}^T\ B^{-1}\ \mathbf{x} &= \frac{1}{N}[\hat{x}^T\ \Sigma^{-1}\ \hat{x} - u^T\ \Sigma^{-1}\ \hat{x} - \hat{x}^T\ \Sigma^{-1}\ u + u^T\ \Sigma^{-1}\ u + 1], & (14) \\ &= \frac{1}{N}[(\hat{x} - u)^T\ \Sigma^{-1}\ (\hat{x} - u) + 1], & (15) \\ &= \frac{1}{N}[d^2 + 1], & (16) \end{aligned}$$

or

$$d_i^2 = N\ [\mathbf{x}_i^T\ B_i^{-1}\ \mathbf{x}_i] - 1. \qquad (17)$$

2.2.3 Locality

Locality is defined as follows [2]:
The input $\mathbf{x}_i$ is local to a boundary if there is no other boundary between the location of the input and that boundary. The aim here is to find the orthogonal projection point, $\hat{\mathbf{x}}'$, of $\hat{\mathbf{x}}$ onto $\hat{\mathbf{w}}$ (again $(\hat{.})$ denotes vector without the bias term), by solving these two equations:

$$\begin{aligned} \hat{\mathbf{x}}' - \hat{\mathbf{x}} &= q\ \hat{\mathbf{w}}, & (18) \\ \mathbf{x}_i'\ \mathbf{w} &= 0, & (19) \end{aligned}$$

where q is an unknown scalar. Multiplying equation 18 with $\hat{\mathbf{w}}^T$ and substituting with equation 19 and $o = \mathbf{x}^t\ \mathbf{w}$ will lead to the solution:

$$\hat{\mathbf{x}}' = \hat{\mathbf{x}} - \frac{o\ \mathbf{w}^T}{\mathbf{w}\ \mathbf{w}^T}. \qquad (20)$$

3 Systolic Array Implementation

Systolic arrays are arrays of small computational units which might contain local memories. These units are pipelined to their nearest neighbours, so they take inputs from previous units, process them and latch the result at their outputs [15, 16, 17]. Systolic arrays have several features such as modularity, synchrony, and high pipelinability which make them distinct from conventional computers [16]. They have been widely utilized to perform special tasks such as digital signal processing and filtering, matrix manipulations including multiplications, inversion and solving linear systems [16, 18, 19].

We now show how vector and matrix computation in the proceeding equations can be implemented in systolic arrays [16]. The quantity $B_i^{-1}\mathbf{x}_i$ represents a matrix-vector multiplication and was realized with the array shown in Fig. 1.**I**. Clearly, this configuration requires $p + 1$ nodes. Three dot product implementations are also needed to compute $\mathbf{x}_i^T B_i^{-1}\mathbf{x}_i$, $\mathbf{x}_i^T\mathbf{w}_i$ and $\mathbf{w}_i^T\mathbf{w}_i$, each of these operations requires only one multiplication

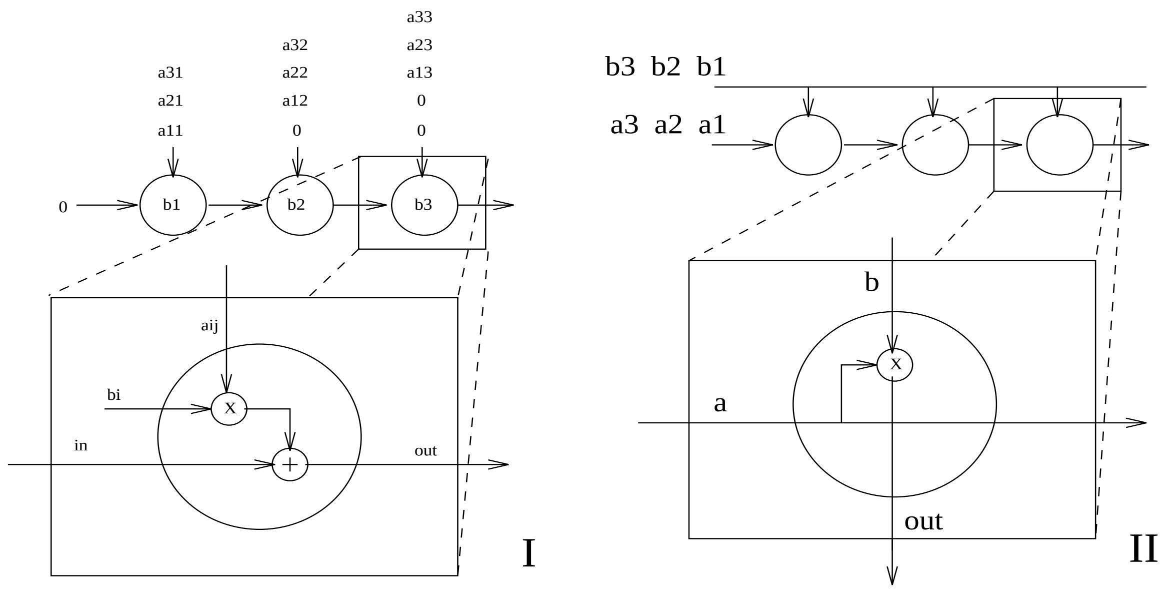

Figure 1: A systolic array diagram for I: matrix-vector multiplication, II: vector-vector cross-product, and their node configuration.

node of the kind shown in Fig. 1.**I** with its output fed back to its input. In addition, a vector cross-product implementation was needed to compute the quantity $\mathbf{kx}_i^T B_i^{-1}$. The corresponding systolic configuration is shown in Fig. 1.**II**. The design takes advantage of the fact that the matrix $\mathbf{kx}_i^T B_i^{-1}$ is symmetric and only one triangular side need be computed. Again, this configuration for cross-product also requires $p+1$ nodes.

In addition, each node is provided with a suitable set of serial-parallel multiplexers, time delays and various non-clocked single operation processors (e.g scalar multiplication, summation,...*etc*). To make the design modular, every node is provided with its own memory to hold the data necessary for the training and creating process mentioned earlier.

The data flow through the various parts is clocked with an internal clock (micro-clock). The longest process in the design is the computation of $\mathbf{kx}_i^T B_i^{-1}$. Since it has to wait for $\mathbf{k}$ to be processed first, it requires $4p-2$ clock cycles to be fully calculated in this particular design. This constraint limits the minimum time for the training process to be completed and receive the next input. This time determines the macro-rate of the data flow through the whole layer. In this particular design the arrays will work on an efficiency of only $p/(4p-2)$.

4 Machine Implementation

On the macro level, the system consists of layers (array of the above nodes). To keep the modularity of the design, we keep an equal number of nodes in every layer. Hence, all layers have the same interconnections, except the input and the output layers. The whole system is supplied with a global clock to control the flow of input data from one layer to another.

Each individual layer is controlled by a Layer Control System (LCS), shown in Fig. 2. As inputs, it receives the values of the three computed criteria $Z, d, \mathbf{x}'$ and the outputs o. The LCS is responsible for deciding to which node(s) this example is local. Then,

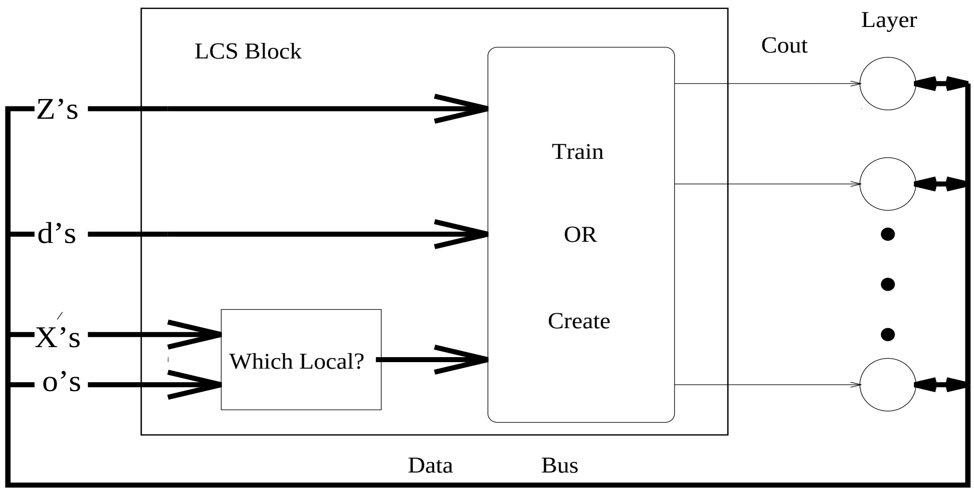

Figure 2: A schematic layout of the Layer Control System (LCS) which control the training/expansion of a layer.

together with the other two criteria and their preset threshold, it decides whether to train the current configuration or create a new node. Although training can involve more than one node in one pass, we only trained one node in our simulations. Specifically, the node with the highest Z value, provided it is local and not remote. Since all the nodes will calculate the new $\mathbf{k}$ and $\mathbf{w}$, training is achieved by clocking these new values into the memory of the chosen node.

On the other hand, if the situation favours the creation of a new node, then the next free node in the layer will be activated, and its memory will be filled with the appropriate values of $B_{i-1}^{-1}, \mathbf{w}_{i-1}$ chosen from a local node with the highest Z value. Data transfer from one node to another is achieved through a data bus connected to all the nodes. Next, the new node will be trained on the current input $\mathbf{x}_i$ in a new clock cycle.

The LCS is a simple combinational logic in which all inputs are binary values, hence it does not need any clocking and only introduces a relatively small delay. The way it chooses which node to train or from which node to create, is handled by its tri-state outputs C_{out}, each connected to one node. At any time, all the outputs C_{out} are set to zero except:

- During training one output is set to 1 to denote the node which was chosen to be trained.
- During creation of a new node one output is set to -1 to denote the node which was chosen to supply its information, and another is set to 1 to denote the newly created node to receive the passed information and to be trained with the current input.

5 Simple Example

This systolic design of the SISP algorithm was implemented by Matlab Simulink and simulated on the Peterson & Barney Vowel Recognition problem [24, 2]. The task is to identify one vowel class out of ten classes. The database is supplied in terms of formant frequencies of these vowels uttered by people of different age and gender. In order to visualize the simulation in the 2-D plan, we chose only the first two formant frequencies

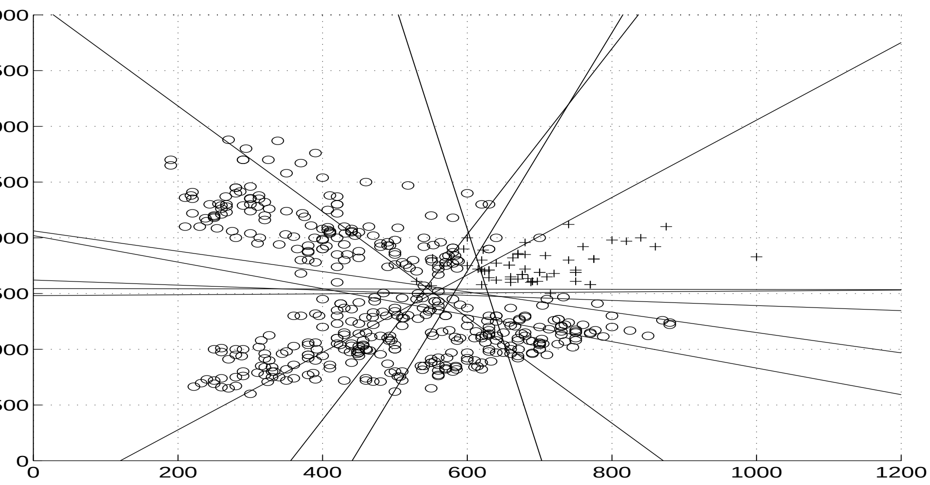

Figure 3: Resultant partitioning of the simulation of the SISP architecture on the 2-D vowel recognition problem. The task is to separate the +'s from the o's. Each boundary line is generated by one node. 500 examples were used.

as our input features. The design contained a layer of 10 nodes, and was provided with instructions to stop creating if the 10 nodes are all occupied. Fig. 3 shows the resultant partitions after applying the Sequential Input Space Partitioning(SISP) simulation on the first 500 examples. The training process required 10 micro-cycles in addition to the small delay in the LCS, so the macro clocking frequency was a convenient $\frac{1}{10}$ division of the micro clock frequency.

6 Conclusions

We have shown that there is an indirect computational advantage in looking at the neural network training problem incrementally. In the past we gave performances on the SISP algorithm. In this work we presented a possible systolic array implementation of the algorithm. Some new modifications on the SISP computations were introduced to make it more suitable for recursive operations. The proposed design is a parallel machine with high modularity, where the data items flow sequentially in one direction through pipelined processes. We also provided a simulation of the architecture on a 2-D vowel recognition problem and showed how it successfully partitioned the input space. We believe that this work emphasizes the importance of sequential learning in designing neural networks by showing , along with other advantages, the feasibility of a simpler hardware implementation of dedicated neural networks.

References

[1] R. S. Shadafan and M. Niranjan, " A Dynamic Neural Network Architecture by Sequential Partitioning of the Input Space," *ICNN'93*, San Francisco, CA. 1993.

[2] R. S. Shadafan and M. Niranjan, " A Dynamic Neural Network Architecture by Sequential Partitioning of the Input Space," Technical Report (CUED/F-

INFENG/TR.127), Cambridge University Engineering Department, May, 1993. (Accepted for publication in Neural Computation).

[3] Mike Wynne-Jones, "Node Splitting: A Constructive Algorithm for Feed-Forward Neural Networks", in J. E. Moody, S. J. Hanson and R. P. Lipmann, (eds.) *Advances in Neural Information Processing Systems 4*, San Mateo, CA, 1992. Morgan Kaufman Publishers.

[4] M. Mezard and J. Nadal, "Learning in Feedforward Layered Networks: the Tiling Algorithm," *J. Physics A: Math. Gen.,* 22, 12:2191-2203, 1989.

[5] M. Frean, "The upstart algorithm: A method for constructing and training feedforward neural networks," Neural Computation, 2, 198-209, 1990.

[6] V. Kadirkamanathan and M. Niranjan, "Application of an architecturally dynamic network for speech pattern classification," *Proc. of the Inst. of Acoustics*, vol 14, part 6, pp 343-350, 1992.

[7] J. C. Platt, " A resource allocating network for function interpolation," *Neural Computation*, 3(2), 213-225, 1991.

[8] T. Ash, "Dynamic node creation in backpropagation networks," *Connection Science*, 1, 365-375, 1989.

[9] M. R. Azimi-Sadjadi and S. Sheedvash, "Recursive node creation in back-propagation neural networks using orthogonal projection method," *ICASSP 91*, Toronto.

[10] S. E. Fahlman and C. Lebiere, "The cascade-correlation architecture," in D. S. Touretzky (ed.) *Advances in neural information processing systems 2,* 524-532, San Mateo, Ca, Morgan Kaufman Publishers, 1990.

[11] M. C. Mozer and P. Smolensky, "Skeletonization: A technique for trimming the fat from a network via relevance assessment," in D. S. Touretzky (ed.) *Advances in neural information processing systems 1,* 107-115, San Mateo, Ca, Morgan Kaufman Publishers, 1989.

[12] Y. Le Cun, J. S. Denker and S. A. Solla, in D. S. Touretzky (ed.) *Advances in neural information processing systems 2,* 598-605, San Mateo, Ca, Morgan Kaufman Publishers, 1990.

[13] Ethem Alpadyn, "GAL: Networks that grow when they learn and shrink when they forget", International Computer Science Institute, technical report TR 91-032, May 1991.

[14] Simon Haykin, *Introduction To Adaptive Filters.* Macmillan Publishing Company, 1984.

[15] J. V. McCanny and J. C. White, *VLSI Technology and Design*, Academic Press, 1987.

[16] S. Y. Kung, *VLSI Array Processors*, Prentice Hall, 1988.

[17] G. M. Megson *An Introduction to Systolic Algorithm Design*, Oxford Science Publications, Clarendon Press, Oxford, 1992.

[18] S. Y. Kung, H. J. Whitehouse and T. Kailath (editors), *VLSI and Modern Signal Processing*, Prentice-Hall, Inc., NJ, 1985.

[19] Stuart Lawson and Ahmed Mirzai, *Wave Digital Filters*, Ellis Horwood Ltd, England, 1990.

[20] Richard O. Duda and Peter E. Hart, *Pattern Classification and Scene Analysis*, John Wiley & Sons, 1974.

[21] K. V. Mardia, J. T. Kent and J. M. Bibby, *Multivariate Analysis*, Academic Press, London, 1979.

[22] G. G. Hall, *Matrices and Tensors*, Pergamon Press, 1963.

[23] Stephen Barnett, *Matrix Methods for Engineers and Scientists*, McGraw-Hill, UK, 1979

[24] G. E. Peterson and H. L. Barney, "Control methods used in a study of the vowels," *JASA*, vol 24, pp. 175-184, 1952.