Proceedings of the 2nd International Symposium on Neuroinformatics and Neurocomputers, pp. 77-84, 1995.

A Provably Convergent Dynamic Training Method for Multi-layer Perceptron Networks

Tim L. Andersen and Tony R. Martinez
Computer Science Department, Brigham Young University, Provo, Utah 84602
e-mail: tim@axon.cs.byu.edu, martinez@cs.byu.edu

ABSTRACT

This paper presents a new method for training multi-layer perceptron networks called DMP1 (Dynamic Multi-layer Perceptron 1). The method is based upon a divide and conquer approach which builds networks in the form of binary trees, dynamically allocating nodes and layers as needed. The individual nodes of the network are trained using a gentetic algorithm. The method is capable of handling real-valued inputs and a proof is given concerning its convergence properties of the basic model. Simulation results show that DMP1 performs favorably in comparison with other learning algorithms.

1 Introduction

One of the first models used in the field of neural networking is the single layer, simple perceptron model (Rosenblatt 1958). The well understood weakness of single layer networks is that they are able to learn only those functions which are linearly separable. Since multi-layer networks are capable of going beyond the limited set of linearly separable problems and solving arbitrarily complex problems, researchers have devoted much effort to devising multi-layer network training algorithms. The unfortunate drawback to many of the current multi-layer training methods is that they require someone to specify the network architecture a-priori (make an *educated guess* as to the appropriate number of layers, number of nodes in each layer, connectivity between nodes, etc.). With a pre-specified network architecture, there is no guarantee that it will be the appropriate network for the problem at hand, and it may not even be capable of converging to a solution.

Thus, several multi-layer training algorithms have been proposed which do not require the user to specify the network architecture a-priori. Some of these include EGP (Romaniuk 1993), Cascade Correlation (Fahlman 1991), Iterative Atrophy (Smotroff 1991), DCN (Romaniuk 1993), ASOCS (Martinez 1990), and DNAL (Bartlett 1993). All of these methods seek to dynamically generate an appropriate network structure for solving the given learning problem during the training phase.

This paper presents a new dynamic method for training multi-layer perceptron networks which we call DMP1 (Dynamic Multi-layer Perceptron 1). This method is similar in spirit to the DCN method given in (Romaniuk 1993) in that:

- The network begins with a single perceptron node and dynamically adds nodes as needed.
- As nodes are added to the network, each node is trained independently from all other nodes.
- Each node is trained on only a portion of the training set, the portion being based upon the current set of misclassified examples - a divide and conquer approach.

However, the DMP1 method differs from DCN in several ways, some of which are:

- The network structure grows backwards from the original output node, rather than forwards.
- The output of each node is connected to at most one other node.
- Each node has as inputs the output of at most two other nodes and the training set inputs.
- A Genetic Algorithm (GA) is used to train the individual nodes of the network.

The basic DMP1 algorithm is discussed in detail in section 2. Two proofs are given, one in section 3 which shows that leaf node convergence implies convergence of the entire network, and one in section 4 which shows that the algorithm is guaranteed to converge to a solution if the training set is consistent. Section 5 discusses the method used to train individual nodes in the network, and simulation results are given in section 6. Conclusion and discussion are given in section 7.

2 The Basic Approach

Although it is relatively straightforward to extend the algorithm to problems which have multiple output classes, in the following discussion we assume that the learning problem has a single, 2 state output. DMP1 begins training with a single output node. If the output node fails to correctly classify some of the positive examples in its training set, then a child node is allocated and trained on a subset of the parent nodes training examples. The subset of training examples given to the new node will be the set of misclassified positive examples combined with the entire set of negative examples from the parent node.

Similarly, if the parent node fails to classify some of the negative examples in its training set, then a child node is allocated and given a training set equal to all of the parent node's positive training examples along with all of the of the parent node's misclassified negative training examples. This process continues until all of the leaf nodes in the network are able to correctly classify their respective training sets.

The intuition behind the DMP1 approach is that if a node incorrectly classifies a certain subset of positive training examples from its training set, then we create a new node (the left child) and pass the parent node's misclassified positive examples to it which, hopefully, will be able to separate the misclassified positive examples from the parent node’s set of negative training examples. Conversely, if the node incorrectly classifies a set of negative training examples we create a new node (the right child) and pass those examples to the new right child. The DMP1 method thus builds a multi-layer network in the form of a binary tree. The algorithm is now discussed in greater detail. We define the following:

T = the original set of examples in the training set.
n_i = node i of the network.
$T_i \subseteq T$ = the set of examples which are used to train node n_i.
t_{ij} = the jth example from training set T_i.
k = the number of input variables.
w_{ij} = the jth weight of n_i, $1 \leq j \leq k+2$ (the 2 extra weights are for the children of n_i).
$A(i,j)$ = the activation of node n_i when presented with training example t_{ij}.
$O(i,t)$ = the output of node n_i when presented with training example t, where

$$O(i,t) = \begin{cases} 0 \text{ if } A(i,t) \leq 0 \\ 1 \text{ if } A(i,t) > 0 \end{cases}.$$

$Z(i,t)$ = the target output of node n_i when presented with training example t.
$PARENT(i) = n_p$ (where $p = \lfloor i/2 \rfloor$) such that the output of node n_i is connected to n_p.
$LEFTCHILD(i) = n_p$ (where $p = 2i$) such that the output of node n_p is connected to n_i through weight w_{ik+1}.
$RIGHTCHILD(i) = n_p$ (where $p = 2i+1$) such that the output of node n_p is connected to n_i through weight w_{ik+2}.

Consider an arbitrarily complex, non-linearly separable, two output classification problem with training set T. Each example in T is defined to be a vector of continuous valued attributes along with an associated target output classification. The network is initially set up with a single node (the output layer). We label this node n_1, and set T_1 =

T. The node is trained with the training set T_1 via the Delta rule training method or some other method until the total sum-squared error (TSSE) settles (the actual method DMP1 uses to train the individual nodes is discussed in section 5). During the training of node n_1, the weights w_{1k+1} and w_{1k+2} are held at zero.

At this point, node n_1 can not make further progress towards a solution. If the network has not converged to a solution (TSSE > ε), then two new nodes, $LEFTCHILD(1) = n_2$, $RIGHTCHILD(1) = n_3$, are created and connected to node n_1 through the weights w_{1k+1} and w_{1k+2} respectively. The left child will be responsible for classifying the positive training examples which were misclassified by the parent node, and the right child will be responsible for classifying the misclassified negative training examples. The values for the weights w_{1k+1} and w_{1k+2} are thus determined by the following, where $SGN(x)$ returns 1 if $x \geq 0$ and -1 otherwise.

$$POS\_SUM(i) = \sum_{j=1}^{k} [(1 + SGN(w_{ij})) \times w_{ij} / 2],$$

$$NEG\_SUM(i) = \sum_{j=1}^{k} [(1 - SGN(w_{ij})) \times w_{ij} / 2],$$

then set

$$w_{ik+1} = \max\{ POS\_SUM(i),\ NEG\_SUM(i)\} + 1$$

$$w_{ik+2} = -w_{ik+1} = -max\{ POS\_SUM(i),\ NEG\_SUM(i)\} - 1$$

Note that for any node n_i, if the output of $LEFTCHILD(i)$ is high, this forces the output of n_i to be high. On the other hand, if the output of $RIGHTCHILD(i)$ is high, this forces the output of n_i to be low. If the outputs of both $LEFTCHILD(i)$ and $RIGHTCHILD(i)$ are high then they cancel each other.

The 2 children of node n_i are then passed a training set which is a subset of the parent node's training set. The training set for each child node is determined as follows.

Let $LEFTCHILD(i) = n_p$. Then we define the training set T_p for n_p as follows:

$T_p = \{t_{ij} \mid t_{ij} \in T_i \wedge [(Z(i,j) = 0) \vee (Z(i,j) = 1 \wedge O(i,j) = 0)]\}$.

Then for all $t_{pm} \in T_p$ (where $1 \leq m \leq |T_p|$), we know that $t_{pm} = t_{ij}$ for some $t_{ij} \in T_i$. For each such $t_{pm} = t_{ij}$ set $Z(p,m) = Z(i,j)$.

Let $RIGHTCHILD(i) = n_q$. Then

$T_q = \{t_{ij} \mid t_{ij} \in T_i \wedge [(Z(i,j) = 1) \vee (Z(i,j) = 0 \wedge O(i,j) = 1)]\}$.

Then for all $t_{qm} \in T_q$ (where $1 \leq m \leq |T_q|$), we know that $t_{qm} = t_{ij}$ for some $t_{ij} \in T_i$. For each such $t_{qm} = t_{ij}$ set $Z(q,m) = 1$ if $Z(i,j) = 0$, else $Z(q,m) = 0$.

Each child node is then trained with its corresponding training set. The input weights on nodes which have children (in this case node n_1) are frozen. If either or both of nodes n_2 and n_3 do not converge to a satisfactory solution on their respective training sets, then the algorithm is repeated recursively with nodes n_2 and/or n_3 as the parent nodes. This process can be repeated until TSSE < ε or the number of positive/negative elements in a node's training set is too small to be statistically significant (i. e. noise). For this paper, the process was simply repeated until no further progress could be made at reducing the TSSE.

After the training phase has been completed the network enters the execution phase. During this phase, incoming examples are presented to every node in the network and the output of node n_1 is then taken to be the predicted output classification for the given example.

3 Leaf Node Convergence

Theorem 1 states that if all leaf nodes in the network correctly classify their respective training sets, then the output of the root node n_1 will be correct for all examples in its training set T_1. This implies that the network output will be correct for all examples in the original training set T, since $T_1 = T$. The formal theorem and proof are given below.

Theorem 1: *If all leaf nodes in the network correctly predict the classification for all of the examples in their respective training sets, then the output of the root node n_1 will be correct for all examples in its training set T_1.*

Proof: The proof is by induction on the maximum network depth. The base case is a tree of depth 1. In this case, there is exactly one leaf node in the tree, which is node n_1. Since the training set of node n_1 is T_1, by definition all of the examples in T_1 are correctly classified if node n_1 (the leaf node) correctly classifies all of its training examples. Furthermore, all examples in the original training set T must be classified correctly, since T_1 is equal to T.

Assume that for a network of depth $\leq h$, if the leaf nodes in the network classify all elements in their respective training sets correctly, then the output of node n_1 will be correct for all examples in its training set T_1. We now show that for a network of depth $h + 1$, node n_1 will classify all of the examples in its training set correctly if the leaf nodes classify their training sets correctly.

Assume that all leaf nodes correctly classify the examples from their training sets. What we must show is that the output of node n_1 is then correct for all examples in its training set. We know that node n_1 has at most two children, a left child (node n_2) and a right child (node n_3), each of which is the root of a tree of depth at most h. Since nodes n_2 and n_3 are each the root of a tree of depth $\leq h$, we have from the inductive assumption that the output of nodes n_2 and n_3 are correct for all of the examples in their respective training sets. There are then two cases for any example $t \in T_1$.

<u>Case 1: $Z(1,t) = 0$</u>

In this case, the output of node n_2 is 0 since $t \in T_2$ from the construction of T_2, and $Z(2,t) = 0 = O(2,t)$ from the inductive assumption.

The output of node n_3 can be either 0 or 1. Assume $O(3,t) = 1$. Then the output of n_1 is forced low by the large negative weight between node n_3 and n_1, and thus $O(1,t) = 0$. Now assume $O(3,t) = 0$. Then we must have $O(1,t) = 0$. This is true since the input weights of node n_1 have not changed since the allocation of node n_3, and if we have $O(1,t) = 1$ then we must have had $t \in T_3$ (from the construction of T_3 since t was misclassified by n_1). Therefore, $O(3,t) = 1$ since node n_3 correctly classifies all elements of its training set, which violates our assumption. Thus for every $t \in T_1$, if $Z(1,t) = 0$, then $O(1,t) = 0$.

<u>Case 2: $Z(1,t) = 1$</u>

This case is similar to case 1. In this case, the output of node n_3 is 0 since $t \in T_3$ from the construction of T_3, and $Z(3,t) = 0 = O(3,t)$ from the inductive assumption.

The output of node n_2 can be either 0 or 1. If $O(2,t) = 1$ then $O(1,t) = 1$. Assume $O(2,t) = 0$. Then we must have $O(1,t) = 1$. Otherwise if we have $O(1,t) = 0$ then we must have had $t \in T_2$ (again from the construction of T_2 since t was misclassified by n_1). Therefore, $O(2,t) = 1$, which again violates our assumption. Thus for every $t \in T_1$, if $Z(1,t) = 1$, then $O(1,t) = 1$.

From case 1 and case 2 we have that for any example $t \in T_1$, $Z(1,t) = O(1,t)$, which concludes the proof. Since the training set T_1 is equal to the original training set T, this implies that the network will have converged to a solution for the original training set T when the leaf nodes of the network have converged to a solution for their respective training sets.

4 Proof of Convergence

Here we prove that the network is guaranteed to converge to a solution. There are two assumptions which we must make in order to guarantee convergence. The first is that the training set must be consistent. That is, we must not have two or more training examples which have equivalent input values and different output classifications. However, DMP1 is capable of generating good solutions when presented with inconsistent training sets. Secondly, we assume that each node added to the network is guaranteed to classify at least one positive and one negative example from its training set. This assumption may seem restrictive, but it is always possible to force this condition (although it may not be desirable to do so). For example, one can force this condition simply by arbitrarily choosing a single positive example and a single negative example, and then using the hyperplane which is the perpendicular bisector to the line joining these two points as the decision surface and set the input weights accordingly. Alternatively, one could use the two points chosen to implement a nearest neighbor classifier for the given node.

Theorem 2: *If the training set is consistent and if each additional node added to the network is guaranteed to correctly classify at least one positive and one negative example from its training set during training, then the network is guaranteed to converge to a solution where all examples in the original training set T are correctly classified.*

Proof: The proof is obvious. Each node in the network is guaranteed to have a training set which is at least one smaller in the number of examples than its parent's training set. The maximum depth of the network is then $|T_1|$, the size of the initial nodes training set. At this depth, the leaf nodes will have at most one example in their training sets, which they are guaranteed to correctly classify from the initial assumption. Thus, the leaf nodes will correctly classify all elements in their training sets, and from theorem 1 the network will have converged to a solution.

The decision function used and the way it is implemented can affect the guaranteed convergence properties of the network. With DMP1 it was decided to relax the constraint on forcing correct classification of at least one positive and one negative example. The GA training method used in DMP1 utilizes a fitness function that strongly favors decision boundaries which have at least one positive and one negative example classified correctly, but the very nature of the GA approach makes it impossible to guarantee such a condition. While GA training method does not guarantee convergence of the network, it does make convergence highly likely for consistent training sets, and it was felt that for most cases the GA training method would be more conducive to good classification accuracy than other methods.

5 Training Method

The method used to train each node in the network is based upon genetic algorithms. The basic method is given in figure 1. Each individual in the population consists of a vector of real values, where each individual is considered to be a potential weight vector for the node being currently trained.

```
Initialize a population of individuals.
While not done
        Evaluate the fitness of each individual.
        Select the top x individuals.
        Create new individuals from those selected above via the genetic operations
                of recombination and mutation.
        The x most fit individuals are selected from the parents and children and allowe
                survive.
```

Figure 1. Genetic training algorithm.

The reason for selecting the genetic algorithm method for training each node was so the network would be able to deal with the situation shown in figure 2. In this example, a small set of positive examples are surrounded by a large set of negative examples.

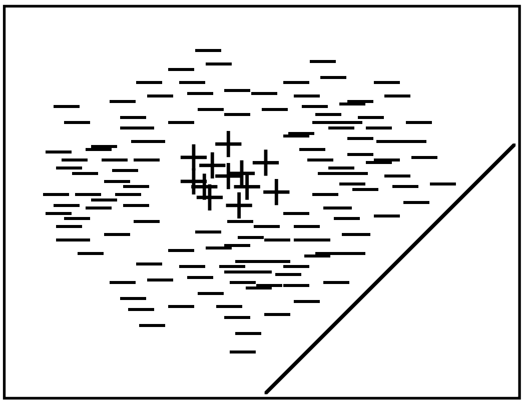
Figure 2. Example pattern distribution.

With this example, the standard perceptron training algorithm is likely to generate a decision surface which has all of the patterns on one side, since that would minimize misclassification error for a single hyperplane. However, assuming that we wanted to separate out the positive examples in figure 2, a decision surface that had all examples on one side would be useless to the DMP1 training method, since it would fail to distinguish at least one positive example from at least one negative example. For example, the decision surface in figure 3a could be more desirable for the purposes of DMP1, since it could eventually lead to the set of decision surfaces in figure 3b, which completely separate the set of positive examples from the set of negative examples.

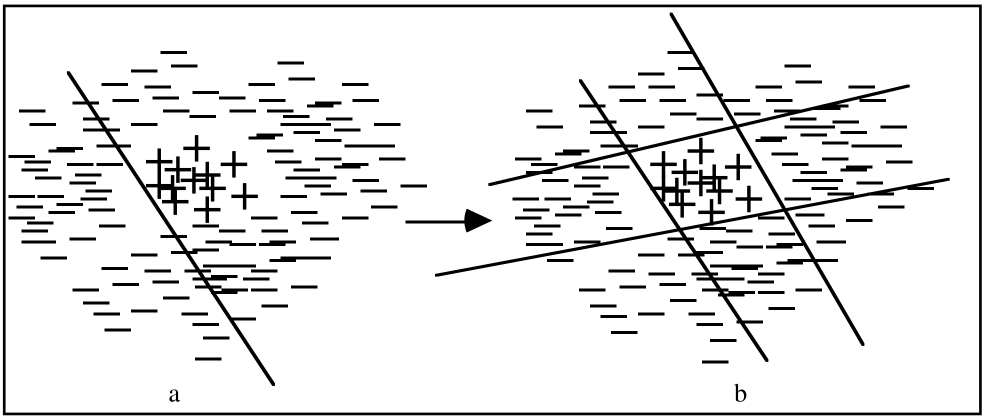

Figure 3. Decision surfaces.

In order to generate these types of decision surfaces with a genetic algorithm, it is necessary to use a fitness function which favors them. For DMP1, the following simple fitness function was selected:

Given a vector of real values $\boldsymbol{i}$ of length k, and
a node n_j of the network, let

$$fitness(i) = sin\left(\frac{\{e|e \in T_j \wedge Z(j,e) = 1 \wedge O(j,e) = 1\}}{\{e|e \in T_j \wedge Z(j,e) = 1\}} \times \pi/2\right) +$$

$$sin\left(\frac{\{e|e \in T_j \wedge Z(j,e) = 0 \wedge O(j,e) = 0\}}{\{e|e \in T_j \wedge Z(j,e) = 0\}} \times \pi/2\right)$$

where the weights of node n_j have been set to $\boldsymbol{i}$.

This function ranges between 0 and 2, and tends to favor decision boundaries which correctly classify at least one example from both output classes. For example, the fitness for a boundary with all of the positive and negative examples on one side will be 1, while the fitness for the boundary in figure 3a would be approximately 1.5. Thus, the genetic algorithm favors the boundary in figure 3a over the boundary shown in figure 2 which minimizes classification error (has all examples on one side). The misclassified examples in figure 3a would then be passed to the child nodes, with the assumption that these nodes will eventually generate a set of decision surfaces similar to those in 3b.

6 Results

The DMP1 training algorithm was tested on several different data sets which were obtained from the UCI Machine Learning Database. The real-world data sets used are Pima Indian (diabetes), Bupa (liver disease), Monks 1-3, Mushroom, Tic-tac-toe, and Sonar. The Monks, Mushroom, and Tic-tac-toe data sets are translated into a binary format where each input had two possible states. For example, a 6 state single input in the original file would be translated into 6 two state inputs, only one of which is set to 1 depending on the original input value. The other data sets which have real-valued attributes are left in their original formats.

15 runs for each data set were conducted. For each run, the data is randomly split into two equal parts, one part being used for training the network and the second part for testing generalization accuracy. Table 1 reports the average accuracy on the test set, and also lists generalization results obtained using two other learning methods for comparison purposes. The first column lists the data sets tested, the second column gives the accuracy on the test set for the DMP1 training method, and the last three columns are the results of the other methods. The numbers in parenthesis are the standard deviations for the reported results. The other methods shown are c4.5 (Quinlan 1986), and a multi-layer back propogation network. These results are taken from (Zarndt 1995). The last row of the table gives the average accuracy across all of the data sets for each of the methods. The average accuracy on the training set across the 15 runs is reported in table 1.

Data Set	DMP1	c4.5 rule	Back Prop
bupa	61.4 (3.2)	59.1	69.8
monk1	94.6 (4.3)	100	77.6
monk2	94.2 (4.2)	65.4	92.2
monk3	98.1 (1.4)	98.9	90
mushroom	99.7 (.002)	100	99.6
pima	62.2 (3.2)	67	76.2
sonar	93.1 (5.4)	70.1	76.4
tic-tac-toe	80.4 (2.4)	65.4	74.4
Average	85.5	78.2375	82.025

Table 1. Average test set accuracy.

Data Set	Accuracy	Std Deviation
bupa	100	0
monk1	100	0
monk2	100	0
monk3	100	0
mushroom	100	0
pima	100	0
sonar	98.4	0.5
tic-tac-toe	100	0

Table 2. Average training set accuracy for DMP1.

In spite of the fact that overlearning can be a problem with the DMP1 training method, it does appear from the results shown in table 1 that its generalization accuracy is at least comparable with the other learning methods. Indeed, the results on the sonar data set show that the DMP1 method is capable of strongly outperforming the other learning methods on at least one real-world problem of interest. In addition, the average accuracy for DMP1 across all of the data sets is also better than that reported for the other learning algorithms.

Table 2 shows that the DMP1 training algorithm is generally able to come up with a solution which correctly classifies each example in the training set. The only exception to this came with the sonar data set, where a few examples from the training set were misclassified. While the convergence characteristics of DMP1 are interesting, the concern is that could lead to memorization of noise and other idiosyncrasies in the training set data which would cause degradation of generalization performance.

In fact, it does appear that this is the case. Table 3 shows the average number of nodes the network contained after convergence for each of the data sets for the DMP1 method. From this table it can be seen that data sets which require large networks tend to be strongly correlated with poor generalization performance. For example, the pima data set required on average 145 nodes to converge to a solution. This was by far the largest network required by any of the data sets, and as expected the generalization results reported for this data set also faired the worst in comparison with the other learning methods. The results on the tic-tac-toe data set are similarly disappointing, where the DMP1 method had a generalization accuracy of 80.4 compared to a 97 percent accuracy reported for the simple, single layer perceptron.

Data Set	Nodes	Std Deviation
bupa	60.5	5.4
monk1	6.8	2.6
monk2	6	3.2
monk3	2.5	0.7
Mushroom	8	2.2
pima	144.8	7
sonar	5.4	1.2
tic-tac-toe	38.4	3.8

Table 3. Average number of nodes in network.

7 Conclusion and Future Work

The overall empirical performance of DMP1 as measured by average generalization accuracy was better than any of the other learning methods. However, it is expected that the generalization capabilities of DMP1 could be improved upon if some effort was made to reduce its sensitivity to noisy inputs. For example, statistical significance tests could be used to prune child nodes, or part of the training examples could be withheld and used as a node pruning set, or the fitness function could be further refined to help prevent memorization of noise and further limit the size of the network.

Another area for improvement is in the network structure and how nodes are added to the network. It is highly likely that the basic structure of the networks generated by DMP1 (binary trees) will not be optimal for solving many problems of interest. For example, in some cases a better solution might be generated if the output node had three children instead of two. The algorithm could be modified so that it would be capable of dynamically adding more than two children to a node deemed beneficial.

With this in mind, future work is concentrating on the following areas:

- Training of individual nodes - fitness function refinement, GA refinement, and other training methods.
- Pruning algorithms - using a pruning set, statistical tests.
- Modifications to the basic algorithm which allow nodes to have more than two children.

8 Bibliography

Azimi-Sadjadi, Mahmood (1993). Recursive Dynamic Node Creation in Multilayer Neural Networks. *IEEE Transactions on Neural Networks*, Vol 4, No 2, pp 242-256.

Bartlett, Eric (1994). Dynamic Node Architecture Learning: An Information Theoretic Approach. *Neural Networks*, Vol 7, No 1, pp 129-140.

Fahlman, S. and C. Lebiere (1991). The Cascade Correlation Learning Architecture.

Martinez, Tony and Jacques Vidal (1988). Adaptive Parallel Logic Networks. Journal of Parallel and Distributed Computing. Vol 5, pp 26-58.

Martinez, T. R. and Campbell, D. M. (1991). "A Self-Adjusting Dynamic Logic Module", Journal of Parallel and Distributed Computing, v11, no. 4, pp. 303-13.

Quinlan, J. R. (1986). Induction of Decision Trees. *Machine Learning*, 1:81-106.

Romaniuk, Steve and Lawrence Hall (1993). Divide and Conquer Neural Networks. *Neural Networks*, Vol 6, pp 1105-1116.

Rosenblatt, F. (1958). The Perceptron: A probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, Vol. 65, No. 6.

Smotroff, Ira, David Friedman and Dennis Connolly (1991). Self Organizing Modular Neural Networks. *International Joint Conference on Neural Networks*, II, 187-192.

Zarndt, Frederick (1995). Masters Thesis, in preparation.