MARS and neural networks with applications to nondestructive evaluation

by

Brett Alan Peterson

A Thesis Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE

Major: Computer Science

Iowa State University
Ames, Iowa
1992

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Dozens of interesting problems in science, engineering, and business fall into the category of multidimensional nonlinear function approximation. In other words, a set of input/output examples (called a *training set*) of a (usually unknown) multidimensional nonlinear real-valued function is given. Using only the training set, the task is to incrementally build a mathematical model that will, given an input vector, approximate the (possibly multidimensional) output of the unknown function. (In order to standardize terminology, we will refer to the model building process as a *learning algorithm.*) Not only must the model be able to approximate the training set "successfully," it must also produce "acceptable" output in response to a set of new input vectors (called a *testing set*). This ability to respond to novel inputs is often called *generalization.* Comparing learning algorithms with one another typically entails the use of algorithmic criteria such as run time, parallelizability, and ease of use. However, a learning algorithm is worthless if the resultant model performs poorly. This observation inspires a set of model criteria used to judge algorithms including input data noise tolerance, testing set approximation accuracy, and interpretability.

Nonlinear function approximation has been studied in mathematics and statistics [50] (under the name "regression") for many years. A new method called Multivariate Adaptive Regression Splines (MARS) has been introduced recently by Jerome

Friedman of Stanford University [17, 18, 19]. MARS builds a function approximation model in the form of an expansion in product spline basis functions [11]. Basically, MARS adds certain important features to an earlier regression procedure known as recursive partitioning [6, 17].

The problem of multivariate function approximation has also been attacked using feed-forward neural networks (FFNNs). Justification for this approach has been demonstrated in numerous papers proving FFNNs to be theoretically capable of either exact function replication [24] or approximate function realization [10, 21, 23, 31]. Hecht-Nielsen's concise findings [24] are well known and use results by Kolmogorov [36] and Sprecher [56] to prove the *existence* of a FFNN that implements any continuous function.

This thesis explains MARS and various FFNN techniques in the context of multivariate function approximation. Empirical results for both contrived and actual function approximation problems are reported. Interspersed throughout the explanations of existing FFNN methods are descriptions of both successful and unsuccessful new FFNN techniques we developed. Empirical comparisons between various function approximation methods are given based on the criteria stated above. Specifically, chapter 2 describes and discusses MARS; chapter 3 explains various FFNN approaches to function approximation; chapter 4 is dedicated to an important paradigm of learning for FFNNs called *generative learning*; chapter 5 explains Quantitative Nondestructive Evaluation (QNDE), an important "real-world" application area for function approximation algorithms, and gives empirical results for MARS and FFNNs applied to QNDE; and chapter 6 presents conclusions and discussion of relevant issues.

2. MULTIVARIATE ADAPTIVE REGRESSION SPLINES (MARS)

Multivariate Adaptive Regression Splines (MARS) [17, 18, 19] is a relatively new procedure for building a function approximation model. The method deals with the approximation of many-to-one mappings; thus, functions with n-dimensional output ($n > 1$) require n separate MARS invocations (one for each output). An alternative way of dealing with multidimensional output is suggested in [18]. Problems involving moderate training set sizes (between 50 and 1000 elements) and moderate input dimension (between 3 and 20 inputs) are considered good candidates for the MARS approach.

2.1 Adaptive Computation and Local Approximation

For purposes of this thesis, we define a *local* function as being comprised of several subfunctions each defined over a specific portion of the domain. Similarly, we define a *global* function as being made up of one subfunction having the entire function domain as its domain. In essence, a global function is interchangeable with its subfunction. Function approximation in statistics has traditionally attempted to fit a global function to a training set using a method such as least-squares [50]. MARS espouses the notion of fitting a *local* function to the training data. The resultant MARS model may then be called a *local approximator*.

Traditional function approximation defines an a priori structure to the resulting approximation model. For example, linear least-squares regression assumes a final fixed form model (namely a linear combination of some set of functions of the input variables) that will not change during computation of the ideal model coefficients. *Adaptive* approximation methods loosen the fixed form restriction of traditional methods by dynamically adjusting the form of the model during model computation. Exactly when and how the model is adjusted is a distinguishing characteristic of each adaptive method. Two general strategies have evolved in statistics for implementing adaptive computation: recursive partitioning [6] and projection pursuit [16, 20]. We will concentrate on recursive partitioning since MARS can be explained as a generalization of this method.

2.2 The Recursive Partitioning Algorithm

The goal of this section is to describe the recursive partitioning method with a series of extensions to finally arrive at the MARS algorithm. (This angle of explanation was originally used in [17].)

Recursive partitioning (and hence MARS) is an adaptive method for computing a local approximator to the (usually unknown) function that generated a given training set. First define the step function $H(\nu)$ as

$$H(\nu) = \begin{cases} 1 & \text{if } \nu \geq 0 \\ 0 & \text{otherwise.} \end{cases} \tag{2.1}$$

The final recursive partitioning approximator $\hat{f}$ is a linear expansion of the form

$$\hat{f}(\vec{x}) = \sum_{m=1}^{M} a_m B_m(\vec{x}) \tag{2.2}$$

where each B_m is called a *basis function* and is defined as products of step functions. (The exact form of the basis functions will be given shortly.) The a_m $(m = 1, \ldots, M)$ are coefficients of the linear combination. Recursive partitioning attempts to adjust the values of the coefficients to give the best fit to the training set data *and* dynamically decide on a "good" set of basis functions for the model. The number of basis functions M and the exact form of each basis function B_m are determined by the method, thus demonstrating the adaptive nature of the algorithm.

Figure 2.1 shows the forward stepwise recursive partitioning algorithm. Line 1 of the algorithm initializes the model to respond with a value of 1 for all inputs. The for-loop of line 2 iterates M (the number of basis functions in the model at any given time during computation) from 2 up to the maximum number of basis functions allowed (M_{max} – a parameter of the algorithm). Each iteration through this loop adds one more basis function B_M to the model by splitting an existing basis function B_{m^*} on dimension x_{v^*} at value t^*. The notion of "splitting" one basis function into two basis functions is accomplished in lines 18 and 19 by replacing the existing basis function B_{m^*} by itself times the step function H applied to the argument

$$\left(x_{v^*} - t^*\right). \tag{2.3}$$

Similarly, the new basis function B_M is created by multiplying B_{m^*} by the step function H applied to the negation of argument 2.3. Since the step function H has the value 0 when its argument is negative and 1 when its argument is positive, the effect of the outermost for-loop is to narrow the scope of control of B_{m^*} over the output of the approximator by only allowing it to respond to inputs that make argument 2.3 positive. Inputs that make argument 2.3 negative are now affected by new basis function B_M. In short, the effective domain of the original B_{m^*} has been

1. $B_1(\vec{x}) \leftarrow 1$
2. FOR $M = 2$ TO M_{max} DO
3. $\quad lof^* \leftarrow \infty$
4. $\quad$ FOR $m = 1$ TO $M - 1$ DO
5. $\quad\quad$ FOR $v = 1$ TO n DO
6. $\quad\quad\quad$ FOR $t \in \{x_{vj} | B_m(\vec{x}_j) > 0\}$ DO
7. $\quad\quad\quad\quad g \leftarrow \sum_{i \neq m} a_i B_i(\vec{x}) + a_m B_m(\vec{x}) H(x_v - t) + a_M B_m(\vec{x}) H(t - x_v)$
8. $\quad\quad\quad\quad lof \leftarrow \min_{a_1,\cdots,a_M} LOF(g)$
9. $\quad\quad\quad\quad$ IF $lof < lof^*$ THEN
10. $\quad\quad\quad\quad\quad lof^* \leftarrow lof$
11. $\quad\quad\quad\quad\quad m^* \leftarrow m$
12. $\quad\quad\quad\quad\quad v^* \leftarrow v$
13. $\quad\quad\quad\quad\quad t^* \leftarrow t$
14. $\quad\quad\quad\quad$ ENDIF
15. $\quad\quad\quad$ END FOR t
16. $\quad\quad$ END FOR v
17. $\quad$ END FOR m
18. $\quad B_M(\vec{x}) \leftarrow B_{m^*}(\vec{x}) H(t^* - x_{v^*})$
19. $\quad B_{m^*}(\vec{x}) \leftarrow B_{m^*}(\vec{x}) H(x_{v^*} - t^*)$
20. END FOR M

Figure 2.1: The forward stepwise recursive partitioning algorithm

"split" on dimension x_{v^*} at value t^*. Note that the old B_{m^*} is *replaced* by itself times a step function. This point is important for the development of MARS in the next section.

The obvious question at this point is: how are m^*, v^*, and t^* chosen? In other words we must answer three questions:

1. Which basis function should be split (m^*)?
2. On which dimension of the input should the function be split (v^*)?
3. On what value of the chosen split dimension should the split take place (t^*)?

Looking at Figure 2.1, we see that the for-loop of line 4 iterates over all currently existing basis functions (of which there are $M-1$); the for-loop of line 5 iterates over all possible dimensions of the input (of which there are n); and the for-loop of line 6 iterates over all those data values t that satisfy the following criteria:

1. t is equal to a value of the v^{th} dimension of some input vector j from the training set. The dimension v is set by the surrounding for-loop (line 5) and j ranges from 1 to N where N is the size of the training set.
2. The current basis function under scrutiny for possible splitting (B_m with m set by line 4) must return a positive output when applied to the j^{th} input vector found in the first criterion.

In short, these criteria choose possible split points directly from the training set vectors that fall into the effective input domain of the basis function currently being evaluated. (The effective input domain of basis function B_m are those input values $\vec{x}$ that evoke a positive response from B_m.)

Having seen how candidate values are chosen for m^*, v^*, and t^*, we now look at the heart of the recursive partitioning algorithm in lines 7 through 14 of Figure 2.1. Line 7 builds a new model g with one more basis function than the current model by splitting candidate basis function B_m on dimension v at data point t. Model g is then evaluated in line 8 using the criterion LOF that gives a measure of the lack-of-fit of g to the training set data. More accurately, line 8 performs a linear regression of the model output on the current set of basis functions in g to achieve a minimization of $LOF(g)$ with respect to the coefficients (the a_k's). In general, the LOF function is a modified version of the generalized cross-validation criterion given in [9]. (More explanation of the LOF criterion will be provided during discussion of MARS.) Lines 10 through 14 store the current split point parameters (the m, v, and t) if g has a lack-of-fit score (given in the algorithm by lof) less than the current recorded best score (stored in lof^*). Lines 7 through 14 repeatedly build new models through the splitting process with the best split (as scored by LOF) being added to the set of basis functions in lines 18 and 19. The algorithm finishes with a model consisting of M_{max} basis functions, where each basis function has the form

$$B_m(\vec{x}) = \sum_{k=1}^{K_m} H\left(s_{km} \cdot (x_{v(k,m)} - t_{km})\right). \tag{2.4}$$

As shown by equation 2.4, recursive partitioning produces basis functions that are products of K_m step functions. Since each step function (H) resulted from a "split," the quantity K_m can also be viewed as the number of splits that were required to produce basis function B_m. Each split is parameterized by the arguments of the step function associated with the split. The sign of the argument is given by s_{km} (either positive or negative), and $v(k,m)$ specifies the input dimension on which split k

occurred for basis function m. Thus, $x_{v(k,m)}$ indicates the split dimension while t_{km} represents the split value used (from the training set) for split k of basis function m. One drawback of recursive partitioning is the discontinuity of its final model (which is piecewise constant). This issue is important for the development of MARS in the next section. Figures 2.2, 2.3, and 2.4 give a graphical account of approximating the function $f(x) = x^2$ with recursive partitioning. The approximation starts out with one constant basis function as shown in Figure 2.2. Part way through computation, the approximator may have the form given in Figure 2.3. Finally, the piecewise constant resultant model is shown in Figure 2.4. As mentioned before, all split points on the x-axis are chosen from the training set.

A detailed explanation of the LOF model criteria is given after the explanation of MARS. Also, since recursive partitioning attempts to overfit a model [6], a backwards stepwise procedure is often required to eliminate basis functions (or pairs of basis functions, depending on the algorithm) that do not help the overall fit. We will describe the MARS version of this procedure in the next section.

2.3 The MARS Algorithm

As stated previously, the MARS algorithm can be thought of as an extension or generalization of recursive partitioning. Three concepts form the basis for the transformation of recursive partitioning into MARS. These ideas are best demonstrated by looking at the MARS forward stepwise algorithm in Figure 2.5. The similarity between the recursive partitioning algorithm (Figure 2.1) and the MARS algorithm (Figure 2.5) reflects the similarity between the methods. The MARS algorithm is best explained by looking at the three extensions to recursive partitioning.

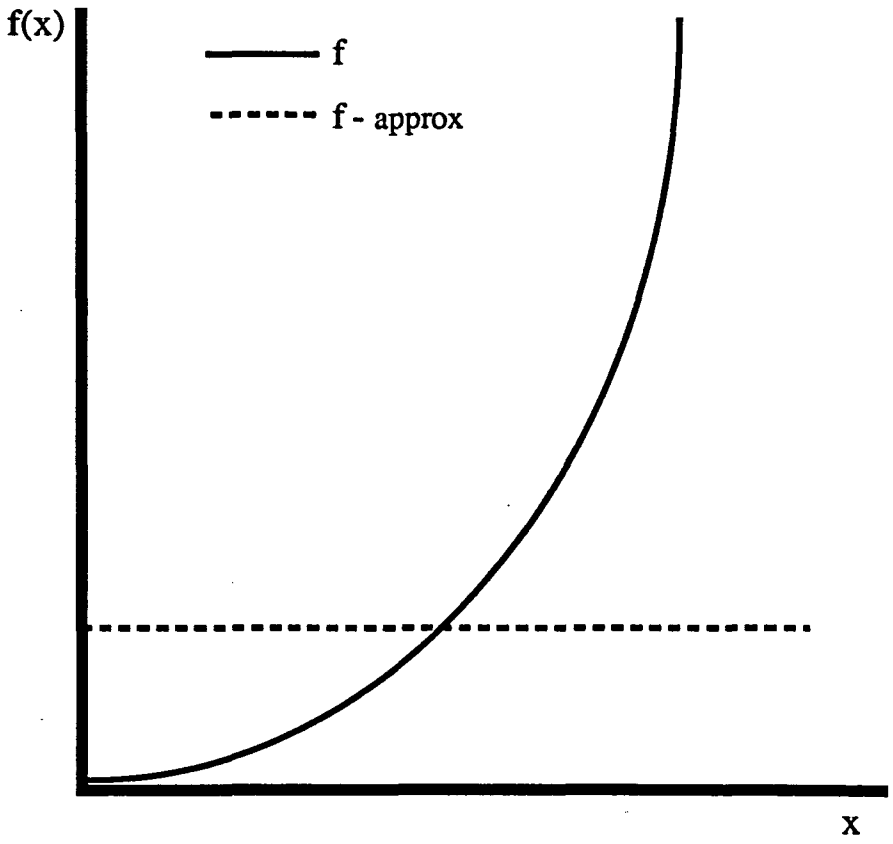

Figure 2.2: Initial recursive partitioning approximation of $f(x) = x^2$

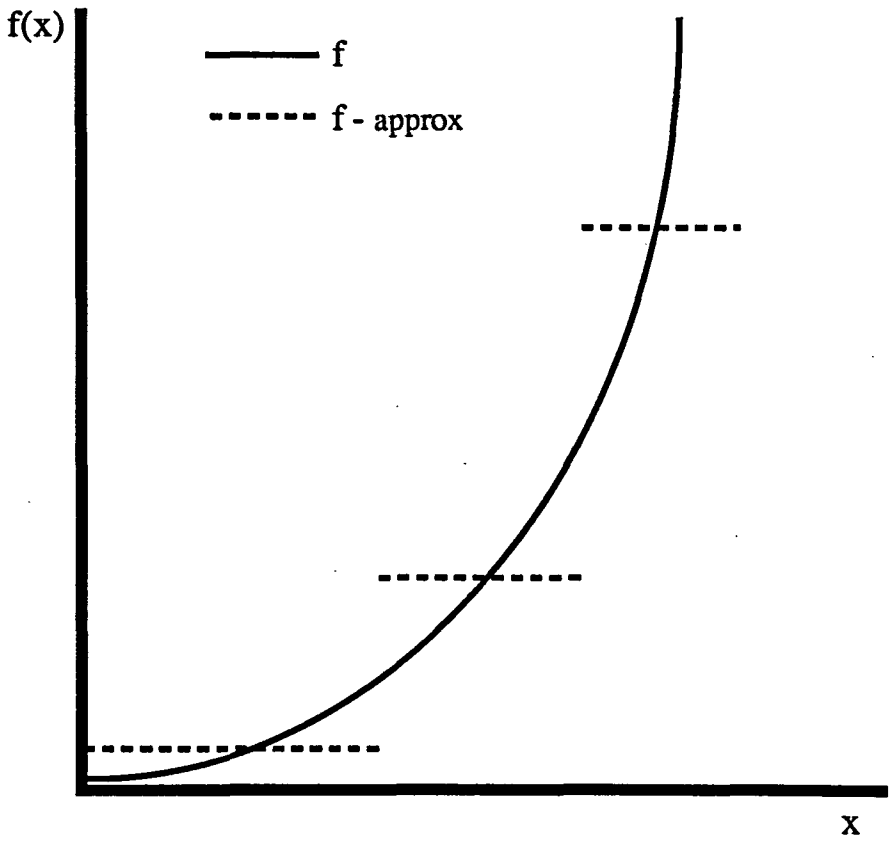

Figure 2.3: Recursive partitioning approximation of $f(x) = x^2$ at some midpoint of computation

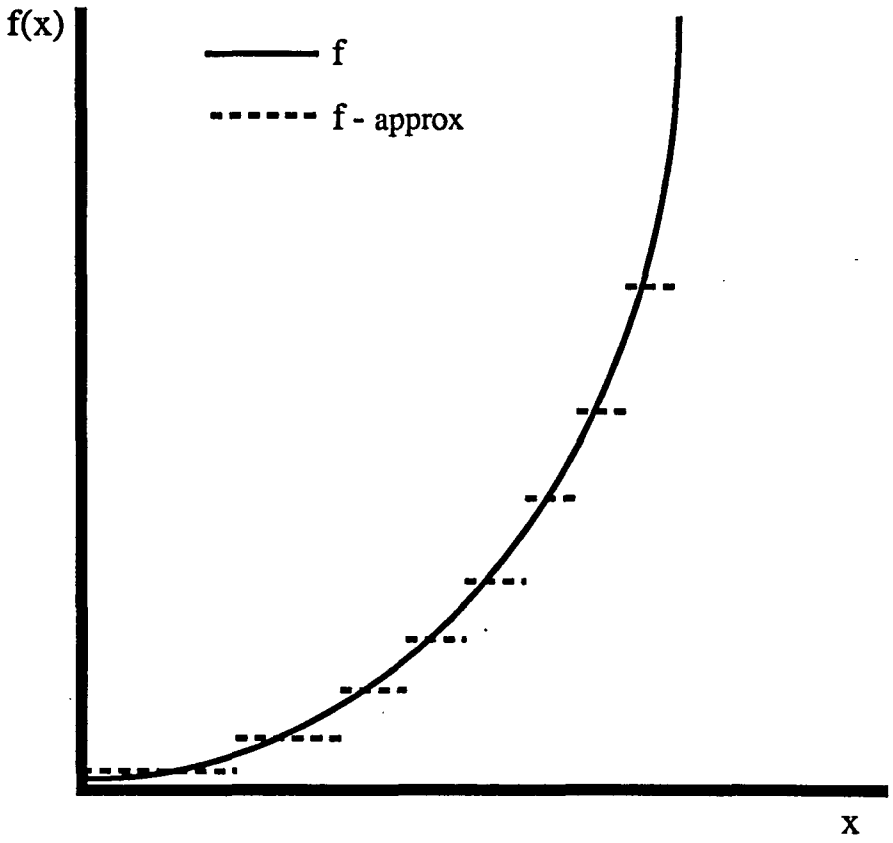

Figure 2.4: Final recursive partitioning approximation of $f(x) = x^2$

1. $B_1(\vec{x}) \leftarrow 1$
2. $M \leftarrow 2$
3. WHILE $M < M_{max}$ DO
4. $\quad lof^* \leftarrow \infty$
5. $\quad$ FOR $m = 1$ TO $M - 1$ DO
6. $\quad\quad$ FOR $v \notin \{v(k,m) | 1 \leq k \leq K_m\}$ DO
7. $\quad\quad\quad$ FOR $t \in \{x_{vj} | B_m(\vec{x}_j) > 0\}$ DO
8. $\quad\quad\quad\quad g \leftarrow \sum_{i=1}^{M-1} a_i B_i(\vec{x}) + a_M B_m(\vec{x})[x_v - t]_+ + a_{M+1} B_m(\vec{x})[t - x_v]_+$
9. $\quad\quad\quad\quad lof \leftarrow \min_{a_1,\cdots,a_{M+1}} LOF(g)$
10. $\quad\quad\quad\quad$ IF $lof < lof^*$ THEN
11. $\quad\quad\quad\quad\quad lof^* \leftarrow lof$
12. $\quad\quad\quad\quad\quad m^* \leftarrow m$
13. $\quad\quad\quad\quad\quad v^* \leftarrow v$
14. $\quad\quad\quad\quad\quad t^* \leftarrow t$
15. $\quad\quad\quad\quad$ ENDIF
16. $\quad\quad\quad$ END FOR t
17. $\quad\quad$ END FOR v
18. $\quad$ END FOR m
19. $\quad B_M(\vec{x}) \leftarrow B_{m^*}(\vec{x})[x_{v^*} - t^*]_+$
20. $\quad B_{M+1}(\vec{x}) \leftarrow B_{m^*}(\vec{x})[t^* - x_{v^*}]_+$
21. $\quad M \leftarrow M + 2$
22. END WHILE

Figure 2.5: The forward stepwise MARS algorithm

The first MARS extension of recursive partitioning forces the resulting model to be continuous (remember that recursive partitioning results in discontinuous piecewise constant models). The discontinuity of recursive partitioning originates with the use of the step function (H) as the fundamental building block of the basis functions. MARS replaces the step function with a continuous function throughout the model to provide model continuity. The fact that step functions are special cases of two-sided truncated power representations of spline basis functions guides the selection of the continuous function. A truncated power basis function is a local function that can be given by the two component functions

$$b(x-t) = [x-t]_+^q \tag{2.5}$$

and

$$b(t-x) = [t-x]_+^q. \tag{2.6}$$

where t is a constant "split point" and q is the spline *order*. The "+" subscript on both equations indicates that only positive arguments are affected by the function. Since Equation 2.6 only takes arguments that are the negation of Equation 2.5, a truncated power basis function effectively splits its domain into two parts with each component function controlling one part. One can now see that the step functions (H) of the recursive partitioning algorithm (Figure 2.1 – lines 7, 18, and 19) are truncated power basis function representations of order zero ($q = 0$) splines. MARS generalizes these functions to first-order ($q = 1$) splines. This fact is demonstrated in lines 8, 19, and 20 of the MARS algorithm (Figure 2.5) in which the step functions of recursive partitioning are replaced by first-order splines. This generalization generates a piecewise-linear continuous model. If one desires continuous derivatives for

the model (which it does not have with $q = 1$ splines), a piecewise-cubic model can be derived from the piecewise-linear output (see [17] for details).

The second MARS extension solves a problem inherent in recursive partitioning. The *interaction order* of a basis function is the number of input dimensions involved in the formation of the basis function. If a split of a basis function occurs on a dimension not already used in the basis function, the recursive partitioning algorithm *eliminates* the basis function and *replaces* it with two basis functions of higher interaction order. The overall effect is an increasingly higher average interaction order of the basis functions as the algorithm progresses. One serious consequence is the inability of recursive partitioning to build a multidimensional additive model. Functions with low-order interactions may be difficult for recursive partitioning to approximate. MARS solves this problem by not removing the parent basis function after a split has taken place. This technique allows multidimensional additive models to be built by always choosing the original basis function B_1 as the function to be split. Since B_1 involves no variables (see line 1 of Figure 2.5), splitting it will always result in basis functions of interaction order one. (The MARS 3.5 implementation provided by Jerome Friedman allows the maximum variable interaction level of the model to be set by the user.) This extension is implemented in lines 8, 19, 20, and 21 of the MARS algorithm (Figure 2.5). The two new basis functions added to the current model in line 8 do not replace any existing basis functions. Similarly, the permanent addition of the best split in lines 19 and 20 does not replace a basis function as is done in recursive partitioning (Figure 2.1 – line 19).

Recursive partitioning allows basis functions to be split on any dimension of the input (training set). Define the *split history* of a basis function to be the set

of dimensions on which a split has occurred in the formation of the basis function at a given time in the execution of the algorithm. Since $q = 0$ splines are used in recursive partitioning, splitting a basis function on a dimension already in the split history of the basis function results in a spline function of the same order (zero). An effect of splitting on a dimension already in the split history using splines of order q ($q > 0$) is to generate basis functions composed of factors that may include individual variables raised to a power greater than q. This phenomenon must not occur if the procedure is to stay within the realm of product spline basis functions (see [11]). Since MARS uses order q spline functions, a mechanism must be supplied to prevent the phenomenon. The third MARS extension to recursive partitioning solves this problem by restricting each basis function to products of distinct input dimensions. The implementation mechanism is found in the FOR loop of line 6 of the forward stepwise MARS algorithm (Figure 2.5). Loop variable v is not allowed to range over the entire set of input dimensions (as in the recursive partitioning algorithm). Instead, v is restricted to those dimensions that are not already in the split history of the current basis function split candidate B_m.

The three extensions to recursive partitioning given above make up the heart of the MARS algorithm. Two issues have yet to be addressed: the LOF function used in both recursive partitioning and MARS, and the backwards stepwise MARS procedure.

The backwards stepwise MARS algorithm given in Figure 2.6 takes the resultant model from the forward stepwise procedure and eliminates one basis function at a time. The effect is to search for the best model and model size using the LOF criterion as a judge of model quality. Specifically, the set of basis functions that should be

1. $J^* = \{1, 2, \cdots, M_{max}\}$
2. $K^* \leftarrow J^*$
3. $lof^* \leftarrow \min_{\{a_j | j \in J^*\}} LOF\left(\sum_{j \in J^*} a_j B_j(\vec{x})\right)$
4. FOR $M = M_{max}$ TO 2 DO
5. $\quad b \leftarrow \infty$
6. $\quad L \leftarrow K^*$
7. $\quad$ FOR $m = 2$ TO M DO
8. $\quad\quad K \leftarrow L - \{m\}$
9. $\quad\quad lof \leftarrow \min_{\{a_k | k \in K\}} LOF\left(\sum_{k \in K} a_k B_k(\vec{x})\right)$
10. $\quad\quad$ IF $lof < b$ THEN
11. $\quad\quad\quad b \leftarrow lof$
12. $\quad\quad\quad K^* \leftarrow K$
13. $\quad\quad$ ENDIF
14. $\quad\quad$ IF $lof < lof^*$ THEN
15. $\quad\quad\quad lof^* \leftarrow lof$
16. $\quad\quad\quad J^* \leftarrow K$
17. $\quad\quad$ ENDIF
18. $\quad$ END FOR m
19. END FOR M

Figure 2.6: The backwards stepwise MARS algorithm

included in the final model is tracked in variable J^*. Thus, line 1 initializes the final model to be the entire basis function set that came out of the MARS forward stepwise procedure. The outer FOR loop of line 4 repeatedly builds the best model with M basis functions where M ranges from M_{max} to 2. The inner for loop builds multiple models by removing one basis function from the current set of basis functions given in L. Each model is compared with all others and the best model of size M is saved in K^* for use by the next iteration of the outer FOR loop. Variable J^* is updated such that the best model found of any size less than or equal to M_{max} is saved. All of this work is done in lines 8 through 17 of Figure 2.6.

The LOF function is a modified version of the generalized cross-validation criterion given in [9]. The exact details of the MARS LOF function are given in [17]. To summarize, the MARS $LOF(g)$ criterion is the average squared-error of the fit of the model g to the training set, multiplied by a penalty function that increases as the number of basis functions in g increases. Associated with the penalty function is a parameter d, which can be regulated by the user, that assesses an increased penalty for large numbers of basis functions. Larger values of d will tend to lead to fewer splits in the final model.

2.4 Discussion

It should be understood that the lack-of-fit function LOF given here is one of many possibilities. The function acts like a heuristic that decides which of a group of models is best. Changing the heuristic may be an interesting area of exploration.

MARS has several parameters which may be set by the user during execution including the maximum number of basis functions M_{max}, the maximum variable

interaction within a basis function, and the d parameter from the LOF function. All experiments reported in this thesis (in upcoming chapters) using MARS 3.5 only altered the setting of M_{max}. All other parameters were left at their default locations. With this simple parameter setting scheme, MARS results were quite good. Among other things, the results demonstrate the parameter-insensitivity of the MARS algorithm.

Many features of MARS are included for computational reasons. The forward stepwise algorithm could conceivably build a piecewise-cubic model; however, the computational advantages inherent in piecewise-linear model building are too great to pass up. One computational bottleneck occurs when executing the least-squares fit (line 9 of Figure 2.5). Recursive partitioning uses characteristics of the step functions to reduce the execution of the least-squares fit to a constant time operation [6]. MARS also uses enhanced least-squares fitting procedures resulting in a final algorithm with a run-time linear in the number of input dimensions, linear in the size of the training set, and approximately cubic in M_{max} (the maximum number of basis functions allowed). Obviously, the key run-time parameter turns out to be the M_{max} parameter. The lower the setting of M_{max}, the faster the algorithm will execute. Results of our experiments (given in upcoming chapters) show that MARS gives good results with reasonably small settings of M_{max} making it a computationally attractive method for function approximation. See [17] for more computational details and considerations.

The potential parallelization of the MARS algorithm is one of its strongest benefits. Friedman [18] has given a formulation of MARS targeted at neural network researchers. We believe that a fruitful area of research is the examination of MARS

from a variety of parallel models of computation. Both the MARS algorithm and the final MARS model may benefit from parallel implementation.

MARS is a promising approach to function approximation worthy of future research. The ideas inherent in the procedure may find value in related fields including feed-forward neural network (and general neural network) research.

3. FEED-FORWARD NEURAL NETWORKS (FFNNs)

As stated in the introduction to this thesis, multivariate nonlinear function approximation is justifiably a major research topic in the field of feed-forward neural networks (FFNNs). This chapter explains the FFNN computational paradigm for *fixed architecture* FFNNs. Fixed architecture FFNNs require the specification of the number of network processors and the processor interconnection pattern before the instantiation of the learning algorithm.

3.1 FFNN Computation

FFNNs are a distributed form of input/output computation. As shown in Figure 3.1, an FFNN is a series of layers of simple processors (called *nodes*). In the most common case of a fully connected FFNN, all nodes in layer m are connected to all nodes in layers $m+1$ and $m-1$. During computation, data only flows from nodes in layer m to nodes in layer $m+1$ where m ranges from 1 to $L-1$ (L being the last or *output* layer). No connections are allowed within a layer or between non-neighboring layers (although these extensions are often interesting). First layer (or *input* layer) nodes receive input from the environment, and output layer nodes send their output signals to the environment. Thus, one can see the inherent input/output nature of FFNNs. Layers between the input and output layers are called *hidden* layers.

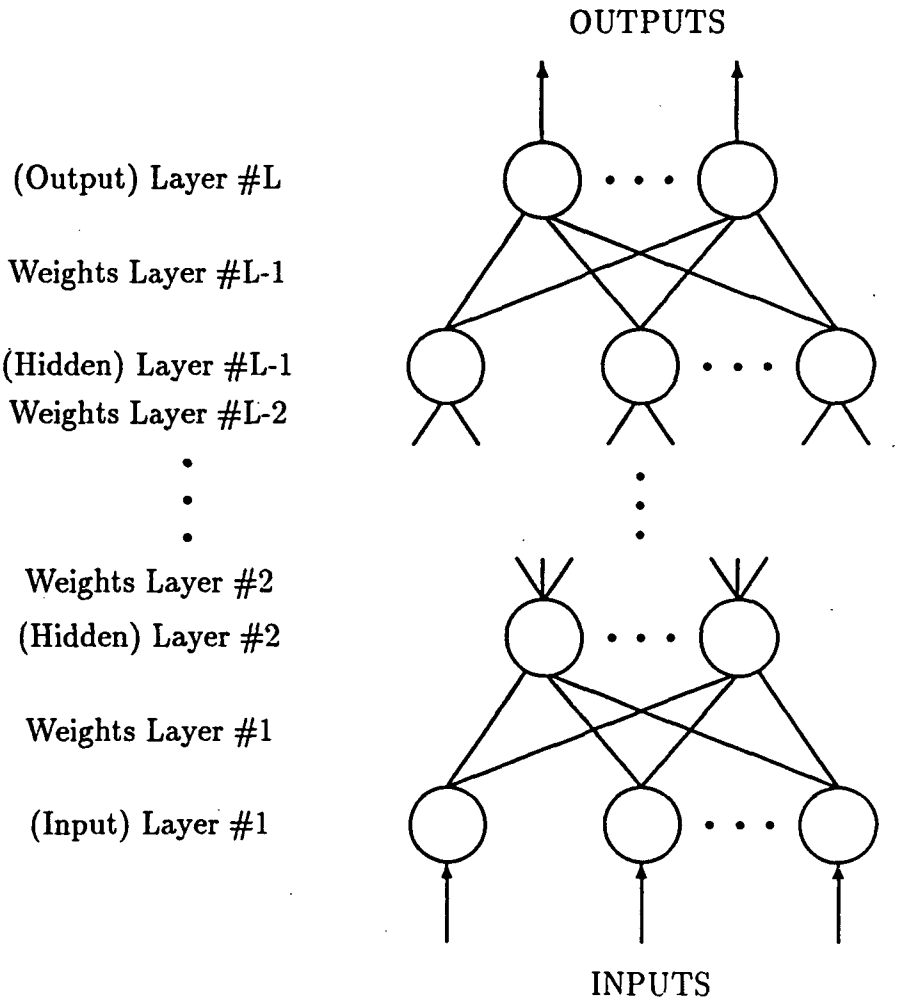

Figure 3.1: A feed-forward neural network (FFNN)

Each node-to-node connection has a scalar *weight* associated with it. The weight associated with the connection from node i in layer $m-1$ to node j in layer m is specified by the variable w_{ji}^{m-1}. Each input signal x_i to node j of non-input layer m is the output signal of node i of layer $m-1$. (The input layer receives unweighted inputs from the environment.) Each signal x_i to node j of layer m is multiplied by the corresponding weight w_{ji}^{m-1}. When an input vector p from the training or testing set is presented to the network input layer nodes, a cascade of computation proceeds through the FFNN. Letting the output of node j at layer m due to input vector p be given by o_{pj}^{m}, the total (or *net*) input to node j in layer m due to input vector p is then given by

$$net_{pj}^{m} = \sum_{i} w_{ji}^{m-1} o_{pi}^{m-1}. \tag{3.1}$$

The weighted sum of equation 3.1 becomes the argument of some nonlinear function f (called the *activation function*) to produce the output o_{pj}^{m} of node j on pattern p. The ability of the network to approximate nonlinear mappings requires the use of a nonlinear activation function. Popular activation functions include the sigmoid, gaussian, and threshold functions.

A learning algorithm in the context of fixed architecture FFNNs must provide a method of setting the weights such that the network "successfully" approximates the desired function. All the FFNN learning algorithms discussed in this thesis belong to the class of *supervised* learning algorithms. This term implies that the algorithm has access to both input and output values in the training set. The training set output values, in conjunction with the supervised learning algorithm, act like a "teacher" guiding the network to an appropriate set of weights. More than one set of appropriate weights may exist.

Historically, FFNNs originated with the contribution of McCulloch and Pitts [38] in 1943. This work combined threshold elements with finite state machines in an effort to describe forms of memory. The field advanced considerably with Rosenblatt's proof of the famous Perceptron Convergence Theorem [46]. Basically, this theorem stated that any two layer (i.e., no hidden layers) FFNN of threshold elements (called a *Perceptron*) can learn to emulate any mapping that it is capable of emulating [39, 46]. The problem is that Perceptrons can only learn mappings which are linearly separable [39]. This discovery slowed research in neural networks for many years. However, it was known that multilayer FFNNs (i.e., FFNNs with one or more hidden layers) could implement some functions which were not linearly separable. The problem now became one of finding an effective way to set the weights of a multilayer FFNN.

Not until 1986 did a method of setting the weights of a multilayer FFNN gain wide acceptance. The backpropagation learning algorithm [47] (which is discussed in a future section) revolutionized the field of FFNNs and set into motion a slew of new research. One should note that Werbos [61] is often credited with the original backpropagation idea; however, the promulgation of the method is due to [47].

3.2 No Hidden Layer Learning Algorithms

Supervised learning algorithms for FFNNs with no hidden layers (i.e., only one layer of weights between the input and output layer) have existed for many years. This class of learning algorithms is often significantly faster computationally than algorithms for training multilayer FFNNs. Since only one layer of weights is necessary, training the weights for each output node of a multidimensional output FFNN can be treated as an independent problem. Thus, it suffices to think of these algorithms

as approximating a single many-to-one function for each output node. As a result, we need not explicitly address multiple output node FFNNs with no hidden layers.

The fixed-increment perceptron learning algorithm [39, 46] was the first supervised FFNN learning method to gain significant acceptance. This technique learns to classify sets of n-dimensional inputs into one of two classes (call the classes 0 and 1). The algorithm is fairly simple:

1. Initialize the weights to random real values.
2. For each training set pattern execute steps 3 and 4.
3. Pass the pattern through the perceptron.
4. If the pattern is misclassified as a 0 when it should be a 1, then add the value of the training pattern vector to the weight vector. If the pattern is misclassified as a 1 when it should be a 0, then add the negated value of the training pattern vector to the weight vector.
5. After each input pattern has been presented, check to see if any patterns were misclassified. If they were, then go to step 2. If all patterns were classified correctly, then exit.

This algorithm has been proven to converge on a set of weights that implement any classification that is linearly separable [39].

Another family of single-layer FFNN classification learning algorithms is based on the *relaxation* method. This method, which was introduced simultaneously in [1] and [40], is used to solve linear systems of inequalities. The input examples from the training set can be viewed as a collection of hyperplanes with "right" and "wrong"

sides. The goal is to find a point on the "right" side of all the hyperplanes. This point satisfies all the constraints imposed by the input examples; thus, it can be used as the weights in the single-layer FFNN. This technique has been used successfully in Hopfield networks [42], bidirectional associative memories [41], and single-layer FFNNs.

The delta rule [47] is a no hidden layer learning algorithm with similarities to the perceptron algorithm. The next section covers the generalized delta rule (i.e., backpropagation) algorithm for multilayer networks. Since the delta rule is simply a restriction of the generalized delta rule, an explanation of the one-layer delta rule method is not given here.

As mentioned above, single-layer learning algorithms are relatively quick compared with multilayer learning algorithms. However, the fact that single layer FFNNs can only learn linearly separable problems limits their usefulness. The *functional-link net* approach [43] attempts to retain the one-layer architecture while simultaneously providing a method of making non-linearly separable problems linearly separable. The idea is to increase the input dimension of the problem. Adding input layer nodes that take functions of original inputs as their inputs may provide the required linear separability in the higher dimension. A common input function simply multiplies the original inputs together as shown in Figure 3.2. Selecting the input functions for the new nodes can be a difficult problem. Deciding how many input dimensions are enough is also unsolved.

We tried one possible solution to the input function selection problem based in Fourier series [34, 59]. A functional-link net can be viewed as a function approximator made up of a linear expansion of basis functions. Suppose each functional-link

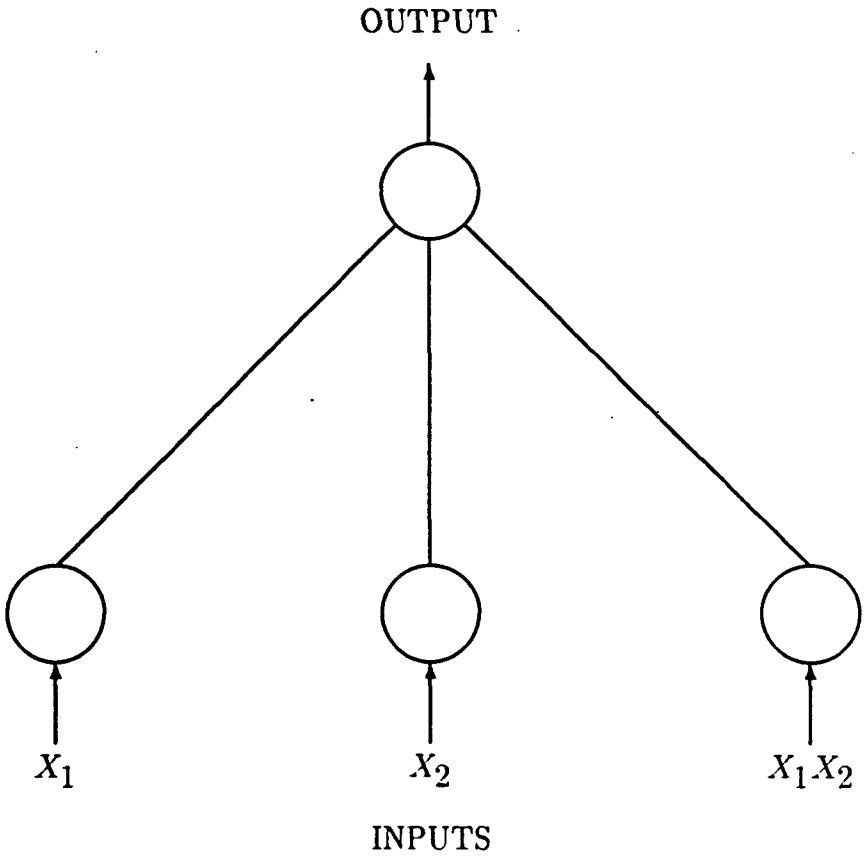

Figure 3.2: A functional-link net FFNN

net node has one trigonometric Fourier series function as its input node function. It has been proven that the mean squared error of the approximation of the training set will be minimal when the coefficients (or in this case, the functional link net weights) are given by the Fourier coefficients [59]. Our idea was to use a one-layer learning algorithm on a functional-link net with input functions given by the trigonometric Fourier series functions. This set of underlying input functions was a rational and theoretically sound choice. Unfortunately, when the input dimension of the problem exceeds three, the necessary formulation of the Fourier series functions becomes almost impossible. Due to this restriction, we abandoned this thread of research. However, if a method is found to list all of the trigonometric Fourier series functions then this idea is certainly worth further investigation.

3.3 The Backpropagation Learning Algorithm

Many supervised learning techniques are based on the concept of error minimization. Specifically, a learning algorithm attempts to minimize the difference between the actual output of the FFNN and the target output given in the training set. Since a training set consists of multiple patterns, a more appropriate and popular error measure is the *sum squared error.* More rigorously, the sum squared error for pattern p over all nodes j in output layer m is given by

$$E_p = \frac{1}{2} \sum_j \left(t_{pj} - o_{pj}^m \right)^2 \tag{3.2}$$

where t_{pj} is the target output for pattern p at node j and o_{pj}^m is the output of node j for pattern p. Of course, other error measures could be used if the situation deemed them appropriate.

The *backpropagation* multilayer FFNN learning algorithm (also known as the generalized delta rule) [47] utilizes approximate gradient descent in the sum squared error as its minimization process. True gradient descent cannot be claimed unless weights are not updated until after all training patterns have been iterated through the FFNN. Usually, backpropagation implementations update the weights after each training set pattern is presented thus only approximating true gradient descent.

The derivation of the backpropagation weight update formulae is based on the first degree Taylor polynomial approximation of the sum squared error function. It is helpful to notice that the error function can be viewed as a function of the FFNN weight vector since all other quantities in equation 3.2 are constant. (The weight vector is buried in the activation function o_{pj}^{m} in equation 3.2, but it is the only variable in the equation.) In order for the sum squared error to always decrease (in pure gradient descent) the change in some $(m-1)$-layer weight w_{ji}^{m-1} due to pattern p must be proportional to the negation of the partial derivative of E_p with respect to w_{ji}^{m-1}:

$$\Delta_p w_{ji}^{m-1} = -\eta_1 \frac{\partial E_p}{\partial w_{ji}^{m-1}} \tag{3.3}$$

where η_1 is a parameter of the algorithm called the *learning rate*. The question now is to determine an expression for calculating the partial derivative of E_p with respect to each weight.

We begin by recalling the expression for a node's net input given in equation 3.1. For this derivation, we assume the popular sigmoid node activation function of node j for pattern p at layer m given by

$$o_{pj}^{m} = \frac{1}{1 + exp(-\beta_j^m \cdot net_{pj}^{m})} \tag{3.4}$$

where β_j^m is a constant regulating the steepness of the function. By the chain rule, we may break up the partial derivative of E_p with respect to weight w_{ji}^{m-1}:

$$\frac{\partial E_p}{\partial w_{ji}^{m-1}} = \frac{\partial E_p}{\partial net_{pj}^m} \frac{\partial net_{pj}^m}{\partial w_{ji}^{m-1}}. \tag{3.5}$$

The goal now becomes to derive a formula for each of the factors of equation 3.5. First, notice that the second factor of equation 3.5 is given by

$$\frac{\partial net_{pj}^m}{\partial w_{ji}^{m-1}} = \frac{\partial}{\partial w_{ji}^{m-1}} \left(\sum_k w_{jk}^{m-1} o_{pk}^{m-1} \right) = o_{pi}^{m-1}. \tag{3.6}$$

Let the negation of the first factor of equation 3.5 be denoted by

$$\delta_{pj}^m = -\frac{\partial E_p}{\partial net_{pj}^m}. \tag{3.7}$$

Combining equations 3.5, 3.6, and 3.7 results in

$$-\frac{\partial E_p}{\partial w_{ji}^{m-1}} = \delta_{pj}^m o_{pi}^{m-1}. \tag{3.8}$$

At this point we see that equations 3.3 and 3.8 specify the weight update formula given by

$$\Delta_p w_{ji}^{m-1} = \eta_1 \delta_{pj}^m o_{pi}^{m-1}. \tag{3.9}$$

The δ_{pj}^m is called the backpropagation *error signal.* This signal takes on different values for the weights entering the output layer nodes than for the weights entering the hidden layer nodes. Applying the chain rule to equation 3.7 gives

$$\delta_{pj}^m = -\frac{\partial E_p}{\partial net_{pj}^m} = -\frac{\partial E_p}{\partial o_{pj}^m} \frac{\partial o_{pj}^m}{\partial net_{pj}^m}. \tag{3.10}$$

The second factor of equation 3.10 is seen to be

$$\frac{\delta o_{pj}^m}{\delta net_{pj}^m} = \beta_j^m o_{pj}^m (1 - o_{pj}^m). \tag{3.11}$$

The first factor is different for output layer weights than for hidden layer weights. For weights entering the output layer, the first factor of equation 3.10 is given by

$$\frac{\partial E_p}{\partial o_{pj}^m} = -(t_{pj} - o_{pj}^m). \tag{3.12}$$

The first factor is derived for weights entering hidden layer nodes as follows:

$$\frac{\partial E_p}{\partial o_{pj}^m} = \sum_k \left(\frac{\partial E_p}{\partial net_{pk}^{m+1}} \frac{\partial net_{pk}^{m+1}}{\partial o_{pj}^m} \right) \tag{3.13}$$

$$= \sum_k \frac{\partial E_p}{\partial net_{pk}^{m+1}} \frac{\partial}{\partial o_{pj}^m} \left(\sum_i w_{ki}^m o_{pi}^m \right) \tag{3.14}$$

$$= \sum_k \frac{\partial E_p}{\partial net_{pk}^{m+1}} \left(w_{kj}^m \right) \tag{3.15}$$

$$= -\sum_k \delta_{pk}^{m+1} w_{kj}^m \tag{3.16}$$

We combine all of the above results to form the final weight update rule for output layer weights given by

$$\Delta_p w_{ji}^{m-1} = \eta_1 (t_{pj} - o_{pj}^m) \beta_j^m o_{pj}^m (1 - o_{pj}^m) o_{pi}^{m-1}. \tag{3.17}$$

Hidden layer weights have a final weight update rule given by

$$\Delta_p w_{ji}^{m-1} = \eta_1 \beta_j^m o_{pj}^m (1 - o_{pj}^m) \left(\sum_k \delta_{pk}^{m+1} w_{kj}^m \right) o_{pi}^{m-1}. \tag{3.18}$$

The backpropagation algorithm repeatedly presents each training pattern to the FFNN. After each pattern has passed through the network, the weights are altered as prescribed by equations 3.17 and 3.18. (One pass through the entire training set is called an *iteration.*) The sum squared error continues to decrease as more and more iterations are run. Eventually, the network will reach an error level that the user can tolerate (within reason).

3.4 Backpropagation Improvement Techniques

Backpropagation learning is plagued with excessively long run times. As a result, a major field of investigation in the field of FFNNs is the run-time improvement of the algorithm given in the previous section. The most prominent of the improvement techniques is *momentum* [47]. This method adds the following term to the final update formulas of equations 3.17 and 3.18:

$$\alpha \Delta_{p-1} w_{ji}^{m-1} \tag{3.19}$$

where α is a parameter of the algorithm that determines the effect of previous weight changes (notice the $p-1$ subscript) on the current weight change. Dozens of other backpropagation improvement techniques have been proposed included those in [32, 33, 57]. Interesting use has also been made of higher order (e.g., second derivative) methods [5].

An interesting new approach with limited biological plausibility [7] is neuronal learning [58]. The idea purports the existence of a tuning parameter within the FFNN node that can be manipulated into the learning algorithm. Specifically, the parameter T_j^m for node j in layer m (called the *temperature*) is given by

$$T_j^m = \frac{1}{\beta_j^m} \tag{3.20}$$

where β_j^m is the value shown in equation 3.4. This implementation of the technique is limited to sigmoid activation functions; however, other parameters of other activation functions may work just as well. Approximate gradient descent is used to adjust the temperatures throughout the FFNN in a method similar to the weight update procedure of backpropagation. The same sum squared error and net input functions

used in backpropagation are used in neuronal learning. The derivation of the neuronal temperature update rules is similar to backpropagation.

We first notice that the change in temperature for node j at layer m for pattern p must be proportional to the partial derivative of the error with respect to the temperature:

$$\Delta_p T_j^m = -\eta_2 \frac{\partial E_p}{\partial T_j^m} \tag{3.21}$$

where η_2 is a learning rate parameter of the algorithm analogous to the backpropagation learning rate. Since neuronal learning is an extension of backpropagation, it must be pointed out that the two learning rates are independently set by the user. By using the chain rule we observe that

$$\frac{\partial E_p}{\partial T_j^m} = \frac{\partial E_p}{\partial o_{pj}^m} \frac{\partial o_{pj}^m}{\partial T_j^m}. \tag{3.22}$$

The second factor of equation 3.22 is given by

$$\frac{\partial o_{pj}^m}{\partial T_j^m} = -\frac{net_{pj}^m}{\left(T_j^m\right)^2} \left(o_{pj}^m\right)\left(1 - o_{pj}^m\right). \tag{3.23}$$

Similar to backpropagation, we define the node error term as

$$\hat{\delta}_{pj}^m = -\frac{\partial E_p}{\partial o_{pj}^m}. \tag{3.24}$$

For nodes in the output layer m, the error term is

$$\hat{\delta}_{pj}^m = \left(t_{pj} - o_{pj}^m\right). \tag{3.25}$$

By an argument similar to the backpropagation weight update derivation, the neuronal error term for non-output layer m is

$$\hat{\delta}_{pj}^m = \sum_l \left(\hat{\delta}_{pl}^{m+1}\right) \left(\frac{o_{pl}^{m+1}\left(1 - o_{pl}^{m+1}\right)}{T_l^{m+1}} \right) w_{lj}^m. \tag{3.26}$$

Combining all the equations we come up with the neuronal update rule

$$\Delta_p T_j^m = -\eta_2 \hat{\delta}_{pj}^m \left(\frac{net_{pj}^m}{\left(T_j^m\right)^2} \left(o_{pj}^m\right) \left(1 - o_{pj}^m\right) \right) \tag{3.27}$$

where $\hat{\delta}_{pj}^m$ is given by equations 3.25 and 3.26. This technique is a reasonable and well-founded extension to backpropagation.

3.5 Discussion

As stated throughout this chapter, FFNNs (especially multilayered FFNNs) often require extremely long learning times. This phenomenon is especially true of backpropagation even though empirical studies have been undertaken to investigate the problem [13]. On top of this rather severe run-time problem, it is often difficult to find appropriate backpropagation parameters such as the learning rate and momentum rate. Even with these difficulties, FFNNs do have significant positive points. They are a form of extremely distributed computation, thus FFNNs are tolerant to possible node failure. It is also claimed that FFNNs are quite robust to noise in the input (both training and testing set).

As a mapping approximation tool, FFNNs must compete with conventional techniques such as MARS. As our experiments show (in upcoming chapters), FFNNs (especially backpropagation networks) train many times slower than MARS, have many barriers to easy use, and can perform worse on testing data than MARS. One must not dismiss FFNNs altogether, however. The field is young and maturing.

Other function approximation neural network paradigms do exist other than backpropagation. Examples include counterpropagation [25] and a one-shot learning

method given in [27]. For an excellent introduction to a large number of neural network paradigms, please see [53].

4. GENERATIVE LEARNING IN FFNNs

Generative learning refers to the process of dynamically building the architecture (number of nodes and the pattern of connectivity between them) of a FFNN during the learning process. Two major techniques have emerged in this general area: node pruning and node addition. Node pruning [52] concentrates on removing nodes and weights that are contributing little to the final result. Node addition tries to "grow" new nodes as necessary to help find a satisfactory architecture [2, 3, 15, 28, 29]. (See [2] for a listing and description of current addition and pruning methods.) Using both techniques in one learning algorithm would be an interesting endeavor; however, most generative learning algorithms deal with one method at a time. Most of the work reported in this thesis concentrated on node addition algorithms; thus, we will concentrate solely on dynamic addition learning techniques.

4.1 Dynamic Node Creation (DNC)

The dynamic node creation (DNC) technique of Timur Ash [3] is an attempt to incrementally build a three-layer (i.e., one hidden layer) FFNN by adding hidden nodes one at a time to increase function approximation accuracy. Standard back-propagation is used to train each newly generated FFNN, and nodes are added when the sum squared error curve given by equation 3.2 becomes sufficiently flat over time.

As before, define an *iteration* as one pass through the training set. Let w be the DNC *window size* given in iterations, Δ_T be the *trigger slope*, a_t be the average squared error at time t for a given output node, m_t be the maximum squared error at time t for any output node, C_a be the average squared error cutoff, and C_m be the maximum squared error cutoff. A hidden node is added to the single hidden layer when the following conditions hold:

$$\frac{a_{t-w} - a_t}{a_{t_0}} < \Delta_T \tag{4.1}$$

and

$$t - w \geq t_0 \tag{4.2}$$

where t_0 is the training iteration during which the last hidden node was added. Conceptually, equation 4.2 guarantees that a node is not added until w iterations through the training set have been completed (thus the name "window size"). Equation 4.1 measures the steepness of the error curve and detects when it has become excessively flat over time. Node addition halts when

$$a_t \leq C_a \tag{4.3}$$

and

$$m_t \leq C_m. \tag{4.4}$$

New nodes are given full connection to all input and output nodes. Ash reports slower convergence time than standard backpropagation, but this effect was expected. The benefit of DNC is the purported solution to the problem of finding a good network architecture.

With any generative learning algorithm, a variety of design questions must be addressed including

1. how many nodes should be added,

2. to which layer should the node(s) be added,

3. what pattern of connectivity should be associated with the node(s),

4. when should the node(s) be added, and

5. what activation function(s) should the node(s) use.

DNC chooses relatively simple and arbitrary answers to these questions without any formal justification. In general, no firm mathematical basis is given for the DNC technique. The method relies mainly on heuristically formed conditions parameterized by user input. Indeed, setting the various backprop and DNC parameters is a difficult task. Severe node growth explosions can result if the parameters are not set correctly, especially on nontrivial function approximation problems. (The simple XOR problem is easily solved by DNC, but this benchmark is too simple to judge the quality of any algorithm.)

4.2 Neuronal Dynamic Node Creation

Neuronal Dynamic Node Creation (Neuronal DNC) is a generative technique we invented that simply combines the neuronal learning technique [58] with DNC [3]. Since both neuronal learning and DNC are based on the backpropagation weight update algorithm, placing the methods together into one algorithm was a natural exercise.

The neuronal DNC technique possesses the attractive and intuitive appeal of cutting back the convergence time of standard DNC by allowing more optimiza-

tion parameters (the neuronal temperatures associated with the sigmoid activation function – see equation 3.20) to vary in a mathematically grounded way. Since our underlying purpose is to approximate fairly high dimensional multivariate functions, a reduction in learning time would be a significant advantage because the large dimension of the functions only adds to the slow convergence of backpropagation-based algorithms.

Computer simulation code for neuronal DNC was written in C and executed on a variety of DECstation 5000, DECstation 2100, and Sun Sparcstation platforms. Twenty-eight parameters are input to the simulator from standard input. Twelve of the parameters are "administrative" (input file names, random number seed, program mode, etc...) and the remaining sixteen are algorithmically relevant. Notice that two separate learning rates must be considered: one for neuronal learning and one for standard backpropagation. Both rates remain static throughout learning. The "momentum" parameter is used in a momentum term added to every weight update in order to account for past weight changes.

A relatively limited number of experiments were attempted. The exclusive-or (XOR) benchmark function was tried, and the network quickly learned the function to the desired degree of accuracy regardless of whether neuronal learning was used. Since XOR is a toy problem in which the training set is equivalent to the testing set, a more interesting continuous multivariate function was formed. The mapping from $\Re^4$ to $\Re^2$ is given by

$$z_1 = x_1 x_2 + x_3 \tag{4.5}$$

$$z_2 = x_2 x_4 + \frac{x_2 x_3}{2} + x_1 \tag{4.6}$$

where $0 \leq x_i \leq 2 \quad \forall i$. Testing and training cases were easily generated.

Table 4.1: Neuronal DNC results for the XOR problem

Expt number	momentum	back prop learn rate	neuron learn rate	initial number hidden nodes	window size	result number hidden nodes	number of iterations
1	0.9	0.5	0.1	1	200	4	705
2	0.9	0.5	0.0	1	200	5	2726
3	0.9	0.5	0.1	1	100	3	359
4	0.9	0.5	0.0	1	100	9	2704
5	0.9	0.5	0.1	1	50	3	277
6	0.9	0.5	0.0	1	50	22	2499
7	0.9	0.5	0.1	1	25	4	235
8	0.9	0.5	0.0	1	25	48	2145
9	0.9	0.1	0.1	1	50	3	228
10	0.9	0.1	0.0	1	50	101	9864

As mentioned previously, the addition of neuronal learning did improve the network's performance over standard DNC for XOR. In fact, neuronal DNC required significantly less training time than standard DNC in many cases.

Parameter settings were chosen based upon observed performance, and no rigorous justification can be given for the decisions made. All experiments for the XOR function used the same four element training set and four element testing set. The DNC trigger was set to 0.05. The average error cutoff was set to 0.0001, and the maximum error cutoff was set to 0.001. Unlimited hidden node addition was allowed. All experiments began with one hidden node. Table 4.1 summarizes a representative sample of experiments attempted. Since all experiments approximated the mapping sufficiently, the important results to observe are the number of iterations through the training set required to obtain the approximation and the number of

hidden nodes generated. Experiments 6, 8, and 10 (all not using neuronal learning) generated extreme hidden node explosions, while the corresponding neuronal DNC experiments (5, 7, and 9) performed well. In all cases neuronal DNC significantly outperformed regular DNC both in node growth and in the number of required iterations. Of course, this statement cannot be generally made because all possible parameter settings have not been attempted. Standard backpropagation was run by using experiment number 10 and shutting off node creation. Three hidden nodes were initially provided. Standard backprop took 12698 iterations compared with 228 for experiment number 9 (the neuronal DNC version).

The continuous mapping problem presented above was attempted and promising results were obtained in which neuronal DNC outperformed normal DNC and standard backpropagation. Unfortunately, this success was tempered by the excessively long process of finding appropriate algorithmic parameters. Also, more tolerance for error was accepted for the continuous problem, but when the tolerance was decreased to a low yet acceptable level, no parameter settings could be found that favored neuronal DNC.

The success of the neuronal DNC technique in the experiments should be tempered by the difficulties experienced with the continuous mapping problem. A variety of possible explanations for this difficulty are presented below.

First and foremost, the vast number of parameters (sixteen) used by neuronal DNC possess an enormous number of possible settings. No clean method is available to decide the appropriate values for each parameter. Indeed, this problem afflicts most backpropagation-based algorithms to some degree; however, DNC seems to worsen the affliction. Neuronal DNC may be a viable technique if the parameters

are set correctly, so calling it an outright failure may be too hasty. Parameter values similar to those given in the original papers were used, but the same success was not achieved. A massive "simulation search" will need to be undertaken to solve this problem; however, since convergence is still relatively slow (three hours for one run on a DECstation 5000), this idea may not be possible.

Another problem may lie in the DNC technique itself. As noted earlier, a node growth explosion sporadically occurs. The underlying problem may be the fact that DNC has no method of rigorously controlling the growth of nodes other than the correct setting of parameters. Thus, adding neuronal learning compounds the problem by setting loose more unregulated parameters. Even though neuronal learning is well founded in approximate gradient descent, the changing architecture may disrupt the search space. Moreover, a newly added node possesses no guarantee of doing "useful" work, i.e. the weights and temperature may play little or no role in the final approximation. Seemingly, the only way to minimize neuron growth in this model is to enlarge the window size sufficiently enabling the existing network to be "fine-trained" as necessary; however, a price is paid in greater convergence times.

A large number of other reasons for the experimental results are possible; however, the common threads in all of them are an excessive number of input parameters, extreme parameter sensitivity, and relatively uncontrolled node addition.

Future directions of this work may include large simulation and diagnostic studies, the addition of novel controllers to DNC, and algorithmic optimization such as decaying learning rates. Actually, we see more promise in studying the underlying node basis functions. Neuronal learning is still a valuable technique that we expect will prove fruitful in future work.

4.3 Cascade-Correlation

The cascade-correlation (CC) generative learning algorithm [15] creates a variant of architecturally pure FFNNs (as understood in this thesis). Unlike DNC, CC makes an attempt to control the usefulness of added nodes by correlating their output with the residual error of the output nodes. It also utilizes only single-layer learning techniques by freezing certain weights during learning. This explicit avoidance of multilayer backpropagation of error signals is one of the most desirable properties (from a run-time standpoint) of the CC technique. CC uses a higher order technique called *Quickprop* [14] in an attempt to speed up convergence time.

The CC algorithm begins with no hidden nodes. Training ensues with the single-layer learning technique until no significant change in error is observed for some pre-set number of training iterations (called the *patience parameter* – set by the user). At this point a new hidden node (called a *candidate node*) is given a connection from every input node and from all other hidden nodes. The new node's output is not yet attached to any other node. A pre-set number of iterations over the training set are executed, and the candidate node's input weights are adjusted in order to maximize the correlation between the candidate node's output and the error at the output nodes. A gradient ascent procedure is used to carry out this task. Notice again that only a single layer of weights are being trained. In addition, one may train a group of candidate nodes (possibly in parallel) and choose the best one. This is possible since the weights of the existing nodes are frozen during node addition.

After a node is added, its input weights are permanently frozen. The entire network is then trained using a single-layer training algorithm; however, only the weights entering the output nodes are allowed to vary. When the change in error

over time is minimal, the process of adding a node begins again. This cycle repeats until the stopping criteria (set by the user) are met.

CC has numerous benefits including no backpropagation of error signals, a well grounded method of ensuring that a node does useful work, and the creation of stationary feature detectors through the freezing of input weights for hidden nodes. The main difficulty with the algorithm is its large number of seemingly sensitive parameters. Empirical studies of CC such as [62] were invaluable in our experimentation with the method. (Our results are presented in upcoming chapters.) As stated in [62], a weight update scheme other than quickprop would be worth trying in conjunction with CC.

4.4 Generative Functional-Link Net

Since one usually wants one-layer FFNN algorithms for run-time efficiency, generative algorithms for setting the FFNN architecture, and the ability to approximate non-linearly separable mappings, we hypothesized that a generative version of a functional-link net [43] would provide a plausible solution. This generative algorithm repeatedly adds nodes to the input layer. The input to a new node is the product of the inputs of some number of original input nodes. The node addition criterion is extremely simple: add a node after a certain number of iterations (specified by the user). We implemented this technique with a variant of the relaxation method [41] and with the single-layer delta rule [47].

We decided to try the technique out on the iris classification benchmark problem from the UC-Irvine machine learning data repository [45]. The technique did not perform as well as expected. After adding over 85 nodes one at a time, the generative

functional-link net did not learn the classification. Either the problem did not become linearly separable in higher dimensions or the generative learning algorithms were insufficient for the task. We did not expect these results and cannot explain them at this time.

4.5 Discussion

Generative learning provides the valuable service of automatically determining a FFNN architecture. The variety of techniques given above all hold promise; however, the node growth control mechanism of cascade-correlation is impressive. The understandable foundation of CC's mechanism is an advantage over the techniques of methods like dynamic node creation. However, setting the parameters of the CC algorithm is a difficult problem (as it is with many other methods).

5. APPLICATIONS OF MARS AND FFNNs TO QNDE

Testing materials for hidden defects including flaws, cracks, or corrosion is important for guaranteeing reliability. Should the defect grow and reach critical size, an unexpected breakdown of the material could result. In certain application areas like aerospace, the results could be disastrous.

Until recently, reliability has been obtained by using off-line destructive testing. Destructive tests subject a material to the equivalent of the stresses which the material will encounter during its lifetime. If the material holds up, the test is successful. However, this approach typically requires large "margins of safety" (at the cost of increased machine weight) to reduce the risk of material breakdown caused by internal flaws [49]. With increasing demands on performance, overly conservative "margins of safety" are becoming less and less acceptable.

Quantitative nondestructive evaluation (QNDE), on the other hand, can help to guarantee reliability by detecting, classifying, and sizing flaws without inflicting stress on the material being examined. QNDE allows an object to emerge from a material integrity test with no change in its chemical or physical properties [49]. The physical techniques used in QNDE include magnetic particles, X-radiation, liquid penetrants, eddy currents, and ultrasonics. We will concentrate on the last two techniques for our experiments.

This chapter presents two flaw sizing problems and two classification problems studied at the Iowa State Center for Nondestructive Evaluation (ISCNDE). Each of the approaches involves the construction of a nonlinear function approximator. Individuals at ISCNDE have already utilized FFNNs to build approximators [8, 37, 54]. We present their results along with new results using FFNNs and MARS.

Ultrasonic and eddy current QNDE techniques involve scanning a material with a probe and collecting signals. The signals are then analyzed and preprocessed into an appropriate form for input to a function approximation tool (e.g., a FFNN or MARS model). In general, the process for sizing and classification requires careful formations of probe scan plans, precise measurement of returning signals, appropriate feature extraction from the resulting data, and the formation of a mathematical tool to map the features to the flaw sizes or classes. Although the first three steps are extremely critical, we will be concerned mainly with the last step (except one experiment in which we undertake the feature extraction). The interested reader is referred to [37] for a discussion of the uniform field eddy current probe data acquisition and feature extraction techniques used to obtain the sizing data used in this thesis. For ultrasonic data acquisition information, please see [54] and [8]. Portions of this chapter are contained in [44].

Choosing a FFNN architecture without using generative learning is not an easy problem. All the architectures for the fixed architecture FFNNs described in this chapter were arrived at by trial and error. Currently, educated trial and error is the only valid way to set network architectures (for fixed architecture learning algorithms) since a complete scan of all reasonable architectures would take a prohibitively long time to accomplish.

5.1 Eddy Current Sizing

The flaw sizing problem using eddy currents requires the function approximator to output the dimensions of of the flaw (length, width, and depth) given a set of features of the flaw as input. An underlying mathematical theory exists (due to Auld [4]) describing the interaction of a uniform electromagnetic field with a three-dimensional flaw. This model agrees with experimental results and allows the generation of "synthetic" input/output data for use by the function approximator.

5.1.1 Comparing MARS to existing FFNN eddy current results

The original data described here were gathered and used initially by Jim Mann at the Center for Nondestructive Evaluation at Iowa State University [37]. Two approaches were taken to test the MARS and FFNN models. The first approach used synthetic training and testing sets generated according to Auld's theory [4]. Two synthetic training sets were produced, one with 1000 elements and one with 100 elements. A single 100 element synthetic test set was used to evaluate the performance of the approximators. The second approach used experimental data obtained by Mann [37] from eight real flaws. Each flaw was scanned twenty times for a total data set of 160 measurements.

5.1.1.1 MARS versus neural nets with synthetic data A standard fixed architecture FFNN using backpropagation was used by Jim Mann [37] on both the 100 and 1000 element training sets. The FFNN used for the 100 element training set consisted of 14 input nodes, one hidden layer of 14 nodes, and 3 output nodes (corresponding to flaw half-length, width, and depth). The network trained on the

Table 5.1: Performance of MARS and FFNNs on synthetic eddy current data

	100 element training set		1000 element training set	
	Mean % Error	Standard Deviation % Error	Mean % Error	Standard Deviation % Error
MARS	9.71	11.01	5.84	4.96
Neural Network	16.81	12.37	3.05	2.89

1000 element training set had an additional hidden layer of 14 nodes. We applied MARS to both training sets varying the maximum number of basis functions over the trials. Only flaw depth was considered in the synthetic cases since it provided the most interesting results and since Mann did not report neural network results on width or half-length.

The best MARS model (as measured by average percent error on the 100 element test set) for the 100 element training set was piecewise cubic and had the maximum number of basis functions parameter (M_{max}) set at 34. The final model had 18 basis functions. The best MARS model for the 1000 element training set was piecewise cubic and had M_{max} set to 100. The final model had 59 basis functions. It should be noted that setting M_{max} to 50 gave almost the same results (with 31 basis functions in the final model). Table 5.1 shows the relevant results reported in [37] for the backpropagation FFNNs and our results with MARS. On the 100 element training set experiment, MARS gave better results and built its model much more quickly. The 1000 element training set was handled better by the FFNN; however, an effective study of the MARS parameter settings was not accomplished due to the long running

Table 5.2: Eddy current flaws

Flaw Number	Flaw Depth (mm)	Flaw Type
1	1.05	Fabricated
2	0.85	Fabricated
3	0.63	Fabricated
4	0.40	Fabricated
5	0.33	Fabricated
6	0.00	Fabricated
7	0.33	Actual Crack
8	0.33	Actual Crack

times of this experiment. For completeness, it should be noted that percent error is not the best measure to use in this case since desired output values are often close to zero. However, to compare with Mann's results, percent error was used. A better measure may be mean absolute error (which is used in experiments in later sections).

5.1.1.2 MARS versus neural nets with experimental data Of the eight available real flaws, six were hand fabricated and two were actual cracks. See Table 5.2 for flaw numbers, depths, and types. Two experiments were run by Mann using backpropagation FFNNs to determine flaw half-length and depth. The first experiment used an 80 element training set consisting of data from fabricated flaws #1, #3, #5, and #6. The test set consisted of the 40 data sets from the two other fabricated flaws (#2 and #4). The FFNN used in this experiment for flaw depth consisted of twenty input nodes, ten nodes in the first hidden layer, three nodes in the second hidden layer, and a single output node corresponding to flaw depth. The best MARS model was piecewise linear with M_{max} set at 10, and the final model

consisted of 5 basis functions. The second experiment used a 120 element training set composed of the data from all six fabricated flaws (#1 through #6). The test set was made up of the 40 elements from the two actual cracks (flaws #7 and #8), and the FFNN architectures used for half-length and depth were unspecified in Mann's report. The best MARS model was piecewise cubic with M_{max} set at 3 (a very small model). The final model consisted of 3 basis functions.

Mean absolute error (average of the absolute values of the differences between the model outputs and the target outputs) will be the measurement tool of choice for this data. Results for flaw depth only are given. Early results for flaw half-length were similar; however, we chose not to report them since time did not allow a thorough and fair search of the MARS parameter space. For the first experiment, the FFNN had a mean absolute error of 0.0254 mm with a standard deviation of 0.0161 mm, and the MARS model had a mean absolute error of 0.0364 mm with a standard deviation of 0.0122 mm. See Figures 5.1 and 5.2 for graphical views of the actual approximations of the FFNN and MARS on the eddy current data for experiment one. (The format of the graphs was borrowed from [37].) In the second experiment, the FFNN had a mean absolute error of 0.0100 mm with a standard deviation of 0.0077 mm, and the MARS model had a mean absolute error of 0.0081 mm with a standard deviation of 0.0065 mm. See Figures 5.3 and 5.4 for graphical views of the actual approximations of the FFNN and MARS on the eddy current data for experiment two. As shown by the data, MARS results were better than the neural network in the second experiment and worse in the first. The greatest difference between MARS and FFNNs on these two experiments was running time. MARS took much less time than the FFNNs to build its model.

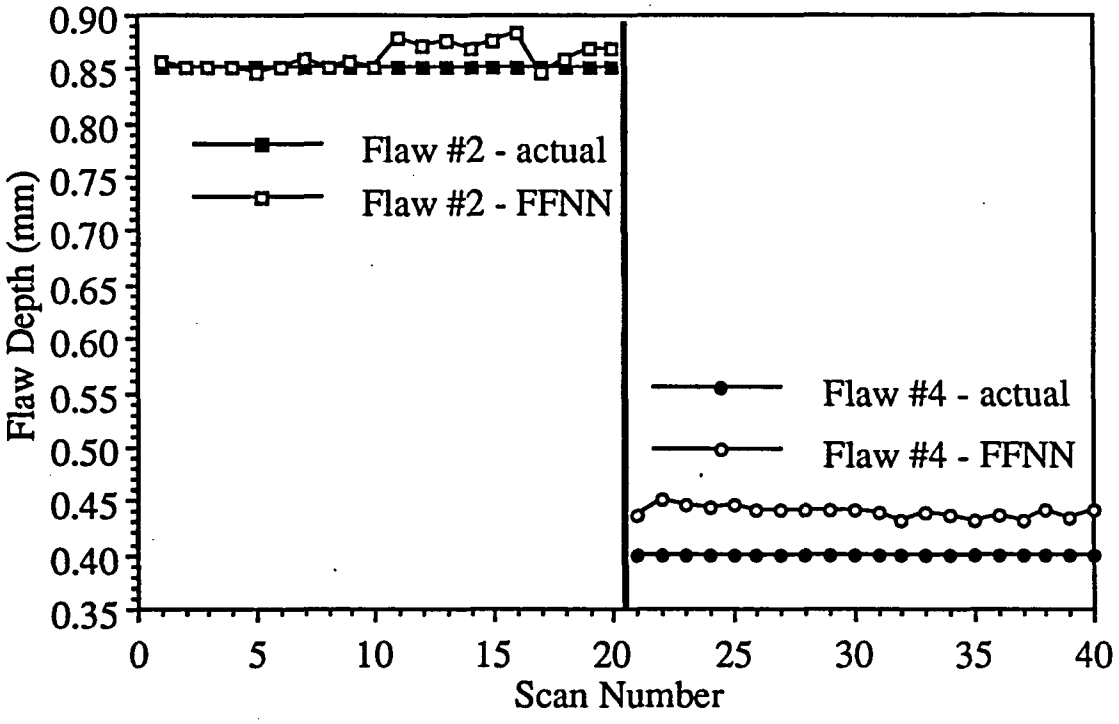

Figure 5.1: Actual values and FFNN estimates of flaw depths for flaws #2 and #4 after training on flaws #1, #3, #5, and #6

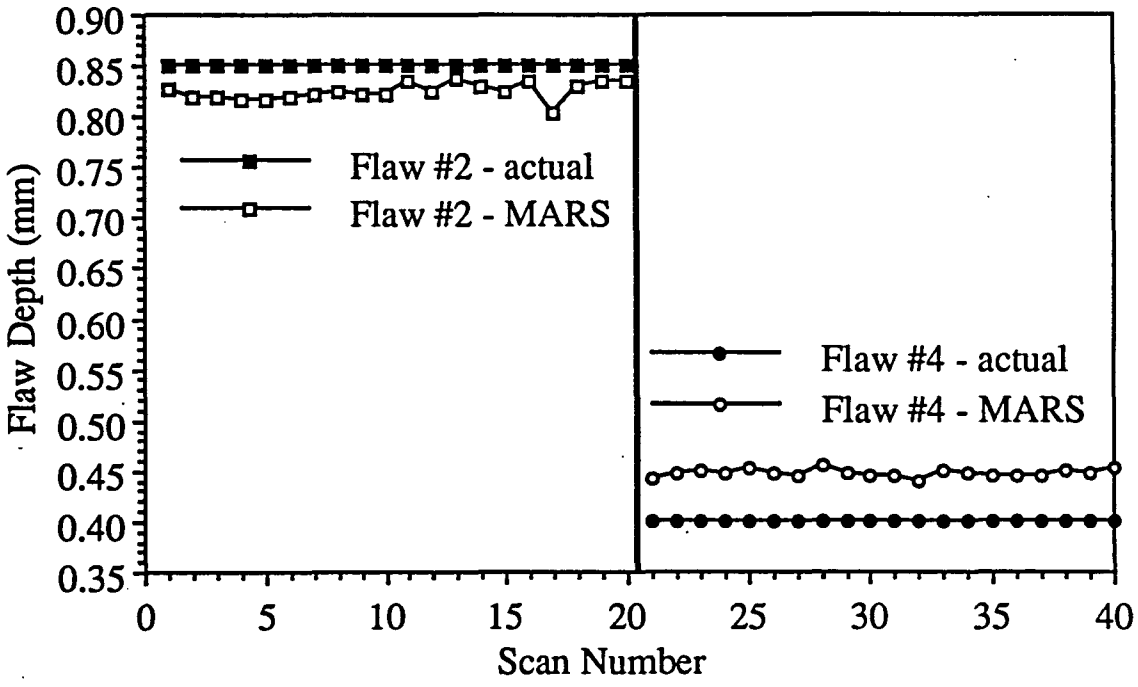

Figure 5.2: Actual values and MARS estimates of flaw depths for flaws #2 and #4 after training on flaws #1, #3, #5, and #6

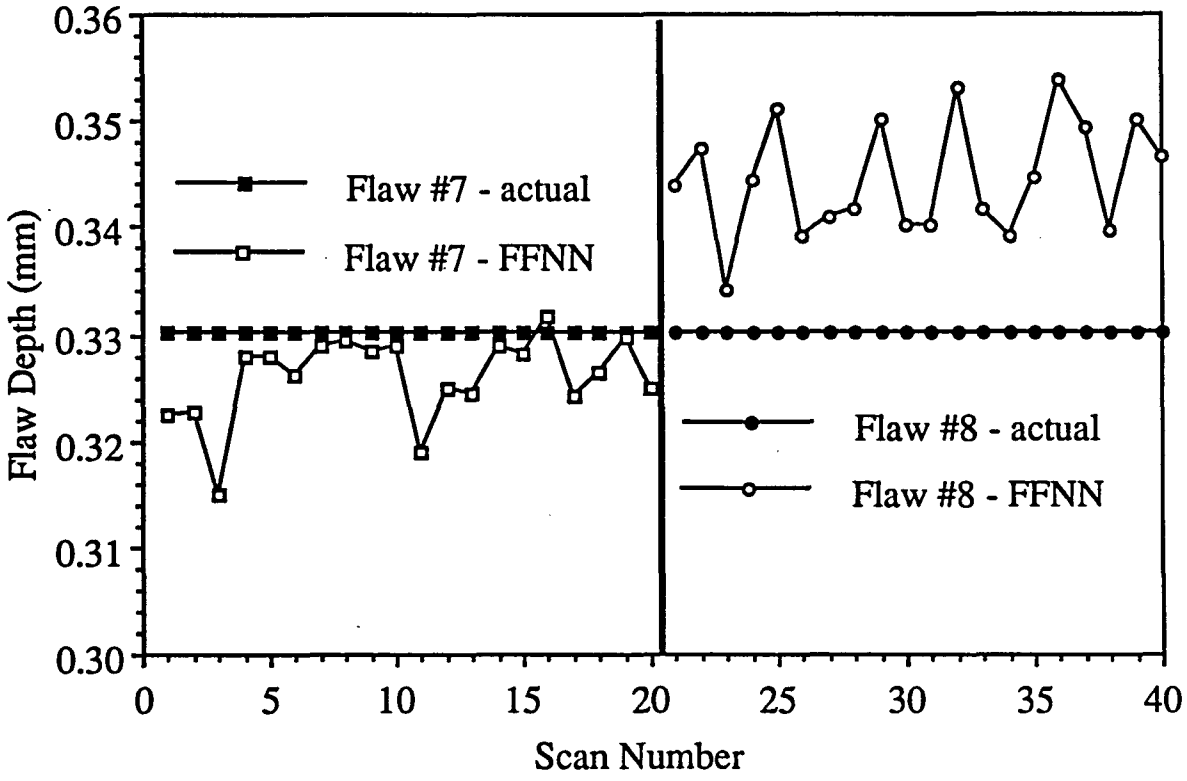

Figure 5.3: Actual values and FFNN estimates of flaw depths for flaws #7 and #8 after training on flaws #1 through #6

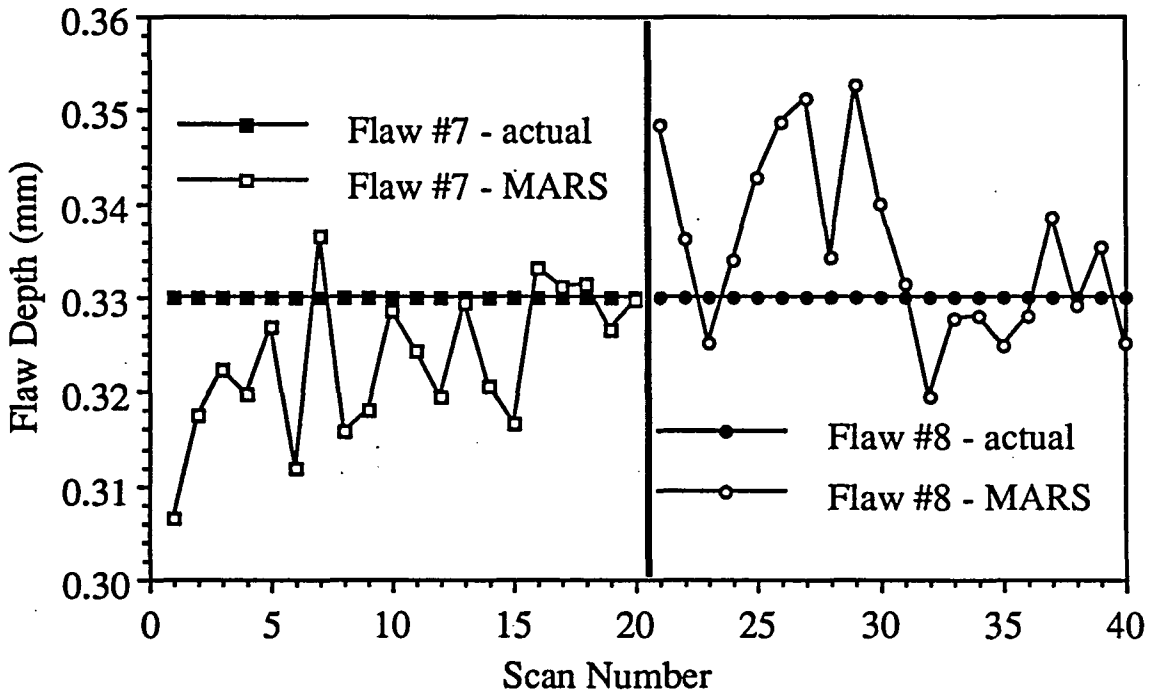

Figure 5.4: Actual values and MARS estimates of flaw depths for flaws #7 and #8 after training on flaws #1 through #6

5.1.2 Comparing MARS, FFNNs, and cascade-correlation

Given the above results, we decided to undertake another experiment using synthetic eddy current data generated according to Auld's theory for flaw depth. Specifically, four training sets were generated of size 100, 200, 500, and 1000. Each training set contained all the elements of the next smallest training set plus additional new patterns. A 4000 element test set was created to examine the generalization capabilities of the techniques used. The two criteria important in this problem are learning time and model performance on the test set. Mean absolute error was used as the test set model performance metric.

Three techniques were applied to this problem: backpropagation, MARS, and cascade-correlation. The only parameter allowed to vary in MARS was M_{max}, which was set from 1 to 100 in increments of 1. Figure 5.5 shows the mean absolute error on the test set as a function of the M_{max} parameter setting for the four training sets. As the graph shows, good results for all training set sizes are obtained with fairly low settings of M_{max}. This observation is important since MARS run-time is approximately cubic in M_{max}.

Setting the parameters in backpropagation and cascade-correlation is a difficult task. Exploratory runs were carried out for each of the two methods to locate "good" parameter settings. (Guidance in the parameter search for cascade-correlation was obtained by referring to the findings of [62].) The patience parameter and maximum growth factor were found to be critical parameters for network convergence and performance in our experiments with cascade-correlation. In fact, one can control the run-time of cascade-correlation significantly through parameter settings; thus, no run-time comparison is given between cascade-correlation and the other methods.

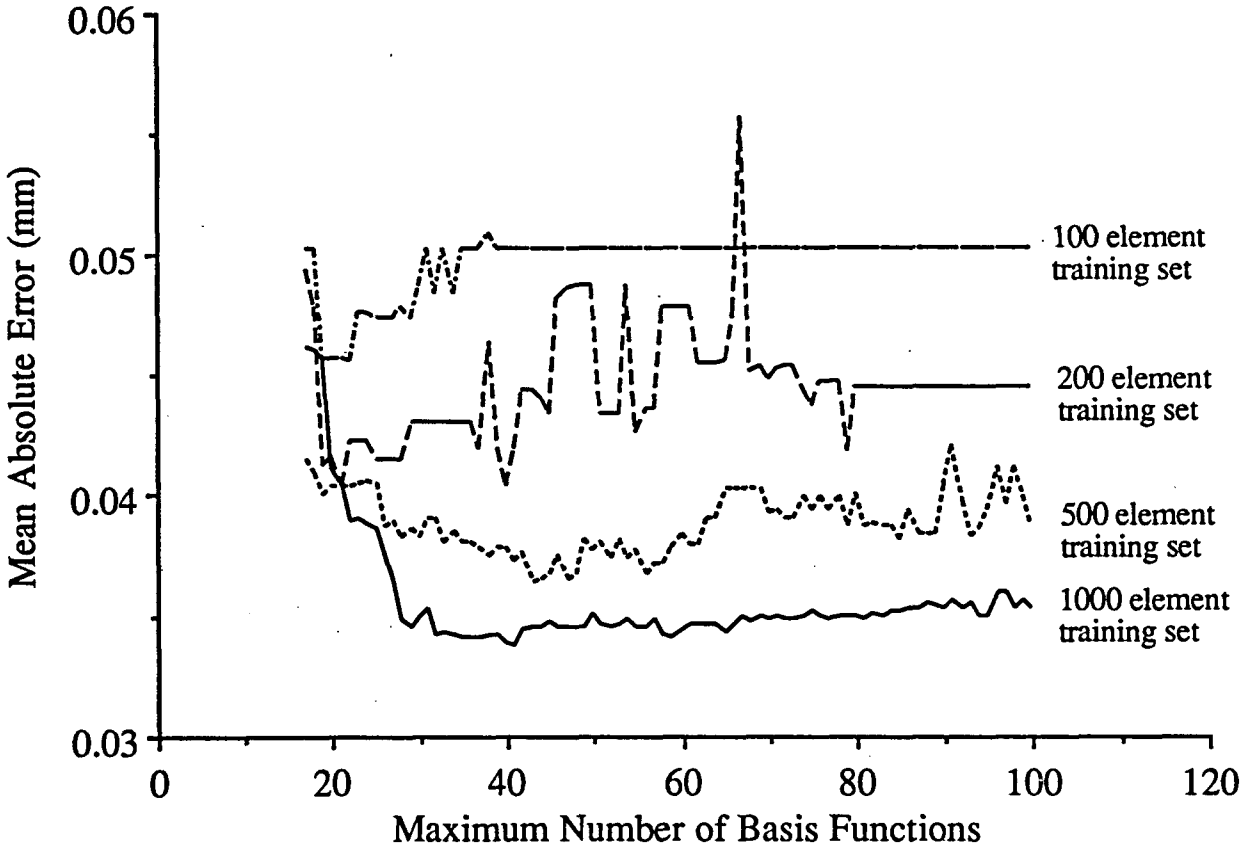

Figure 5.5: MARS performance on the 4000 element test set as a function of the maximum number of basis functions parameter

Table 5.3: Results using backpropagation on the 4000 element test set

	Size of Training Set		
	100	200	500
Number of Hidden Nodes	14	14	28
Mean Absolute Error During Testing (mm)	0.1160	0.0760	0.0593
Standard Deviation of Absolute Error During Testing (mm)	0.1325	0.0757	0.0707
Run Time	8 hours	15.5 hours	71 hours

Suffice it to say, the run-time of backpropagation was much longer than the run-time of cascade-correlation regardless of parameter settings.

Backpropagation entails the additional burden of finding and setting an architecture prior to learning. In the exploratory runs it was found that one hidden layer almost always provided the best results. Adding a momentum term to backpropagation actually hindered network performance both in terms of learning time and performance on the test set; thus, the results given in this section do not utilize momentum. Simulation run-time was used to report learning time results for backpropagation and MARS. All simulations were run on a DECstation 2100 under ULTRIX V4.1.

Results of the best model for each method are shown in Tables 5.3, 5.4, and 5.5. All the backpropagation FFNNs were run for 100,000 iterations and had a learning rate of 0.5. The best networks for the 100 and 200 element training sets had one hidden layer of 14 nodes, while the best network for the 500 element training set had one hidden layer of 28 nodes. No experiments were run on the 1000 element training

Table 5.4: Results using cascade-correlation on the 4000 element test set

	Size of Training Set		
	100	200	500
Number of Hidden Nodes	25	25	40
Type of Hidden Nodes	Gaussian	Sigmoid	Gaussian
Mean Absolute Error During Testing (mm)	0.1451	0.0943	0.0800
Standard Deviation of Absolute Error During Testing (mm)	0.3061	0.0742	0.0697

Table 5.5: Results using MARS on the 4000 element test set

	Size of Training Set			
	100	200	500	1000
Maximum Number of Basis Functions	22	40	43	41
Final Number of Basis Functions	12	17	27	31
Model Type	Piecewise Linear	Piecewise Linear	Piecewise Linear	Piecewise Linear
Mean Absolute Error During Testing (mm)	0.0456	0.0404	0.0364	0.0338
Standard Deviation of Absolute Error During Testing (mm)	0.0428	0.0378	0.0330	0.0293
Run Time	1 min 16 secs	9 min 25 secs	31 min 10 secs	57 min 23 secs

set for backpropagation or cascade-correlation due to the long running times of the procedures.

As the tables show, the performance of cascade-correlation on the test set did not measure up to either backpropagation or MARS. This phenomenon could be the result of inappropriate parameter settings; however, more time was spent fine-tuning the cascade-correlation parameter settings than was spent on the other two methods combined. As stated in [62], a version of cascade-correlation with a simpler learning algorithm may be interesting.

The most significant result is the relatively small run-time of MARS compared with backpropagation. Even if the backpropagation software was made significantly more efficient, MARS would still take much less time to train. The disparity between the run-times of MARS and backpropagation combined with the better performance of MARS on the test set (see Tables 5.3 and 5.5) allow us to conclude that MARS is a better method for this problem. Table 5.6 shows the MARS model which, for each training set, gives the same performance as backpropagation on the testing set. Notice the small run-times due to the small settings of M_{max}. The 1000 element training set column in Table 5.6 is included to show the drastic cut in run-time (without much loss in test set performance) that results from setting M_{max} to 29 instead of the best MARS model setting of 41 (see Table 5.5). Figure 5.5 reinforces graphically that good results are obtained with M_{max} set relatively low. The fact that only one parameter (M_{max}) was allowed to vary is another promising MARS factor. Even better results may be obtained by utilizing some of the other MARS parameters.

Table 5.6: Results using MARS on the 4000 element test set with testing accuracy approximately the same as backpropagation

	Size of Training Set			
	100	200	500	1000
Maximum Number of Basis Functions	3	13	10	29
Final Number of Basis Functions	3	10	10	22
Model Type	Piecewise Linear	Piecewise Cubic	Piecewise Cubic	Piecewise Linear
Mean Absolute Error During Testing (mm)	0.0968	0.0679	0.0585	0.0345
Standard Deviation of Absolute Error During Testing (mm)	0.0745	0.0584	0.0446	0.0302
Run Time	5 secs	53 secs	1 min 4 secs	23 min 6 secs

5.2 Ultrasonic Sizing

The ultrasonic sizing problem discussed here involves finding the dimensions of a best-fit equivalent circular shape for an isolated planar crack. The response wave generated by a crack is characterized by two large peaks called "flashpoints." The time separation between the flashpoints can be related to the radius and orientation of the crack; thus, a measurable quantity exists (time) from which the crack dimensions can be obtained [8]. A mathematical analysis has been carried out [8] which transforms the problem into a three-to-three mapping with the three outputs representing the crack radius and two angular parameters. This output scheme fully describes the crack size and orientation. Often, crack size is the crucial factor in sizing; thus, for the sake of comparison with existing experiments, only crack size will be discussed here. (This implies a three-to-one mapping rather than a three-to-three.)

The data for the ultrasonic flaw sizing problem were originally generated by C.-P. Chiou at the Center for Nondestructive Evaluation at Iowa State University [8]. A 330 element synthetic training set was generated along with a 1920 element synthetic testing set.

A fixed architecture FFNN using an adaptive variant of backpropagation was used by C.-P. Chiou [8] on the 330 element training set. The adaptive aspect of the learning algorithm mainly affected the speed of the algorithm. The FFNN used had three input nodes, twelve nodes in the first hidden layer, twelve nodes in the second hidden layer, twelve nodes in the third hidden layer and one output node corresponding to crack size.

We applied MARS to the training set varying the maximum number of basis functions over the trials. The best MARS model was piecewise linear with M_{max} set

Table 5.7: Performance of MARS and FFNNs on ultrasonic synthetic sizing data

	Number Between 0 and 10% Error	Number Between 10 and 20% Error	Number Between 20 and 30% Error	Number Between 30 and 40% Error	Number Between 40 and 50% Error
MARS	1829	0	0	87	4
Adaptive Neural Network	1827	2	0	65	26

to 50. The final model had 38 basis functions. The results in this case were measured in classes of error as shown in Table 5.7. MARS and the FFNN performed about the same for this experiment; however, the issue of run time is once again important. MARS took less time than the FFNN to build its approximator. It should be noted that all of the 30–40% and 40–50% errors occurred when the target value was close to zero. As with the eddy current data, percent error may not be the best measurement tool. For completeness: MARS had a 2.62% mean error with a 8.10% standard deviation.

5.3 Ultrasonic Classification

The flaw classification problem is important in cases where knowledge of flaw type is more important than exact flaw size. Determining the type of a flaw is often sufficient to make crucial decisions about the material.

5.3.1 Ultrasonic classification problem one

The data for the ultrasonic flaw classification problem presented here were originally used by S.-J. Song of the Center for Nondestructive Evaluation at Iowa State University and were generated from samples provided by Westinghouse Corporation [54]. Basically, the goal is to separate known welding defects into three distinct classes: crack, porosity, or slag. A total of 239 input/output examples were selected, 120 for the training set and 119 for the testing set. 104 were known cracks; 53 were porosity; and 82 were slag [54].

Song [54] applied a probabilistic neural network (PNN) to this problem [55]. This type of FFNN has its architecture determined by the number of output classes and the choice of training samples. Specifically, a PNN is a four layer (two hidden layers) FFNN. Nodes in the first hidden layer employ a gaussian activation function f given by

$$f(net) = exp\left((net - 1)/\sigma^2\right) \tag{5.1}$$

where σ is a parameter of the algorithm. One node exists in this layer for each training pattern, and the input layer and first hidden layer are fully connected. The weight w_{ji} from node i in the input layer to node j in the first hidden layer is set to x_{ji}, where x_{ji} is the i-th dimension of the j-th training set pattern. One node exists in the second hidden layer for each output class, and it simply outputs its net sum. A connection w_{ji} exists from first hidden layer node i to second hidden layer node j if training example i belongs to class j. All such weights are set to 1. One output node is required for each output class, and a connection c_i exists from second hidden layer node i to output node i. The c_i are set by the user as parameters of the system and are multiplied by the output of the second hidden layer to give the output

of the network. In general, a higher relative value of c_i will increase the chances of classifying a pattern as class i. The output node with the largest output value is chosen as the class of the input pattern.

Given that there are t_i training patterns from class i, the first hidden layer produces t_i gaussian distributions with centers at the training patterns. A small setting of the σ parameter generates narrow gaussian distributions while a large setting implies wide distributions. Node j of the second hidden layer sums together the gaussian distributions of class j. This technique approximates the probability density function for each class. Please see [54] and [55] for a more detailed description of the underlying PNN theory.

MARS was applied to the training set varying the maximum number of basis functions over the trials. Since MARS has a continuous output value, each of the three categories was assigned a range of output values. We had to choose how large each range would be with respect to the other ranges. This new MARS parameter was found to be critical in the performance of MARS on all classification problems attempted. The best MARS model was piecewise cubic with M_{max} set to 5. The final model consisted of 4 basis functions. The second category (porosity) in the best model had a smaller relative range of output values than cracks or slag.

The model performance evaluation criteria used by Song [54] requires explanation. The correct accept rate CA_i for class i is defined as

$$CA_i = \frac{m_i}{n_i} \tag{5.2}$$

where m_i is the number of test set patterns from class i classified correctly and n_i is the number of test set patterns belonging to class i. The false reject rate FR_i for

Table 5.8: MARS versus FFNN on ultrasonic classification problem one

	MARS	FFNN
Correct Accept Crack (%)	69	75
Correct Accept Porosity (%)	35	42
Correct Accept Slag (%)	49	54
False Reject Crack (%)	34	31
False Reject Porosity (%)	18	10
False Reject Slag (%)	18	22

class i is given as

$$FR_i = \frac{\sum_j m_{ji}}{\sum_j n_j} \tag{5.3}$$

where m_{ji} is the number of test set patterns from class j classified by the model as class i and n_j is the number of test set patterns belonging to class j. In general, these quantities measure fractions of samples classified correctly or incorrectly. To compare our MARS results with Songs neural network results, we'll use the correct accept/false reject notion.

Looking at Table 5.8 we see that the probabilistic FFNN performed better than MARS; however, the fact that MARS is relatively competitive is worth noting. The MARS procedure is not meant to be a classification scheme, so its performance is admirable. One interesting point was the ease with which we were able to build MARS models to favor one class over another by changing the relative sizes of the class output ranges. Thus, it seems MARS could be easily tailored to specific classification applications as necessary.

5.3.2 Ultrasonic classification problem two

The second ultrasonic classification problem is based on data taken at the David W. Taylor Naval Ship Research and Development Center. The flaws were located in the weld zone of 36 steel plates with an approximate thickness of two inches. Half the plates were from a decommissioned submarine, and half were manufactured with intentional flaws placed in the material. Each defect was classified using a consensus approach from a variety of different inspection methods. Two plates were destructively evaluated to verify the accuracy of the classification. It is estimated that 85% of the data we received was classified accurately. This figure may be inaccurate; however, we assume its validity. The data came to us in the form of sampled voltage versus time waveforms coupled with the classes of the defects generating the waves. Four flaw classes are present in the data: crack, lack of fusion, porosity, and slag. Of the 736 samples, 132 were cracks, 260 were lack of fusion, 130 were porosity, and 214 were slag.

One of the most difficult (and ad hoc) steps in the processing of NDE data is feature extraction. Eleven features were extracted from the waveforms to be used as input to a classifier. No particular justification can be given for these features, and we make no claim that they are in any way optimal. Other features may drastically improve the results of the experiments. We extracted features from the time domain (the raw signals) as well as from the frequency domain (via the fast fourier transform (FFT) [12]). We generated the phase graph from the FFT output by taking the inverse tangent of the imaginary part of the FFT divided by the real part. The magnitude spectrum was obtained by taking the square root of the sum of the squares of the imaginary part and the real part of the FFT output. We also translated the

magnitude spectrum graph onto a log scale to allow the smaller maxima of the graph to show up. Finally, the energy of a wave is given by the sum of the squares of each sampled point. The eleven features were:

1. maximum peak of the time domain waveform
2. maximum peak of the magnitude spectrum
3. number of maxima from the time domain waveform
4. number of maxima from the magnitude spectrum
5. number of maxima from the phase graph
6. number of "tall" maxima from the time domain waveform where "tall" is defined with a threshold of 25.0
7. number of "tall" maxima from the magnitude spectrum where "tall" is defined with a threshold of 250.0
8. maximum peak of the log scale of the magnitude spectrum
9. minimum peak of the log scale of the magnitude spectrum
10. the signal duration of the time domain waveform
11. the energy of the time domain waveform

All of the experiments reported in this section compare the performance of the probabilistic neural network (PNN) and MARS. Results are reported using the correct accept/false reject notion introduced in the previous section. Deciding which results

to report is difficult using this metric since "good" results often depend on the needs of the user. For example, a high correct accept rate for a particular class may be more important than an overall high correct accept rate. In fact, a high correct accept for a particular class almost invariably results in a high false reject rate for that class. This tradeoff is important when deciding which results to report. Unfortunately, it is impossible to report results from all of the runs. We report only those results considered most important and interesting with the understanding that many other possibly valuable outcomes are not included here.

For experiments 1A, 1B, and 1C (below) we randomly chose half of the flaws from each of the four classes for the training set and half for the test set. Thus, each set contained 368 samples of which 66 were cracks, 130 were lack of fusion, 65 were porosity, and 107 were slag. For experiments 2A, 2B, and 2C we randomly chose 100 flaws from each class for the 400 element training set. The test set contained the remaining 336 defects.

5.3.2.1 Experiment 1A This experiment weighted all of the classes equally. For MARS, this means that each class was assigned an equal range of output values. For the PNN, each output weight c_i was set to 1. The "best" MARS model was piecewise cubic and had M_{max} set to 28. The final model consisted of 13 basis functions.

Looking at Table 5.9 we see that the PNN performed slightly better for this experiment. The data in the table were from a PNN with σ set to 0.17. For completeness, the PNN with the highest overall probability of detection had σ set to 0.04.

Table 5.9: MARS versus the PNN FFNN for ultrasonic classification experiment 1A

	MARS	PNN FFNN
Correct Accept Crack (%)	14	20
Correct Accept Lack of Fusion (%)	34	32
Correct Accept Porosity (%)	72	72
Correct Accept Slag (%)	11	27
False Reject Crack (%)	02	06
False Reject Lack of Fusion (%)	31	14
False Reject Porosity (%)	55	50
False Reject Slag (%)	03	14

5.3.2.2 Experiment 1B This experiment weighted the crack class more heavily than the other three classes. For MARS, this means that the range of output values for the crack class was greater than the other classes. For the PNN, the output weight c_i was set to 3 for the crack class and 1 for the other classes. The "best" MARS model was piecewise cubic and had M_{max} set to 16. The final model consisted of 10 basis functions.

Table 5.10 shows that the PNN performed significantly better for this experiment. The data in the table were from a PNN with σ set to 0.07. The PNN with the highest overall probability of detection had σ set to 0.02.

5.3.2.3 Experiment 1C This experiment put all non-crack flaws into one class and all cracks into another class. Parameter settings for both MARS and the PNN were set to favor the crack class. For the PNN, the c_i output weight was set to 2 for the crack class and 1 for the non-crack class. The highest overall probability of detection was achieved with σ set to 0.02. Table 5.11 compares the PNN with σ set

Table 5.10: MARS versus the PNN FFNN for ultrasonic classification experiment 1B

	MARS	PNN FFNN
Correct Accept Crack (%)	94	94
Correct Accept Lack of Fusion (%)	21	42
Correct Accept Porosity (%)	11	17
Correct Accept Slag (%)	09	31
False Reject Crack (%)	54	47
False Reject Lack of Fusion (%)	22	10
False Reject Porosity (%)	11	06
False Reject Slag (%)	06	10

Table 5.11: MARS versus the PNN FFNN for ultrasonic classification experiment 1C

	MARS	PNN FFNN
Correct Accept Crack (%)	70	79
Correct Accept Non-crack (%)	70	79
False Reject Crack (%)	30	21
False Reject Non-crack (%)	30	21

at 0.03 with the "best" MARS model (piecewise linear, $M_{max} = 27$, and a final model consisting of 16 basis functions). The PNN showed a significant advantage in test set performance.

5.3.2.4 Experiment 2A This experiment weighted all of the classes equally. For the PNN, each output weight c_i was set to 1. The "best" MARS model was piecewise linear and had M_{max} set to 40. The final model consisted of 21 basis functions.

Table 5.12: MARS versus the PNN FFNN for ultrasonic classification experiment 2A

	MARS	PNN FFNN
Correct Accept Crack (%)	16	56
Correct Accept Lack of Fusion (%)	45	55
Correct Accept Porosity (%)	60	70
Correct Accept Slag (%)	17	43
False Reject Crack (%)	03	13
False Reject Lack of Fusion (%)	26	07
False Reject Porosity (%)	50	22
False Reject Slag (%)	06	18

Looking at Table 5.12 we see that the PNN performed much better for this experiment. The data in the table were from a PNN with σ set to 0.05 (which was the PNN with the highest overall probability of detection).

5.3.2.5 Experiment 2B This experiment weighted the crack class more heavily than the other three classes. For the PNN, the output weight c_i was set to 4 for the crack class and 1 for the other classes. The "best" MARS model was piecewise linear and had M_{max} set to 19. The final model consisted of 14 basis functions.

Table 5.13 shows that the PNN performed better for this experiment. The data in the table were from a PNN with σ set to 0.03. The PNN with the highest overall probability of detection had σ set to 0.02.

5.3.2.6 Experiment 2C This experiment put all non-crack flaws into one class and all cracks into another class. Parameter settings for both MARS and the

Table 5.13: MARS versus the PNN FFNN for ultrasonic classification experiment 2B

	MARS	PNN FFNN
Correct Accept Crack (%)	84	84
Correct Accept Lack of Fusion (%)	32	50
Correct Accept Porosity (%)	33	47
Correct Accept Slag (%)	16	33
False Reject Crack (%)	34	25
False Reject Lack of Fusion (%)	27	09
False Reject Porosity (%)	19	14
False Reject Slag (%)	10	18

Table 5.14: MARS versus the PNN FFNN for ultrasonic classification experiment 2C

	MARS	PNN FFNN
Correct Accept Crack (%)	88	75
Correct Accept Non-crack (%)	79	78
False Reject Crack (%)	21	22
False Reject Non-crack (%)	12	25

PNN were set to favor the crack class. For the PNN, the c_i output weight was set to 3 for the crack class and 1 for the non-crack class. The highest overall probability of detection was achieved with σ set to 0.02. Table 5.14 compares this PNN ($\sigma = 0.02$) with the "best" MARS model (piecewise linear, $M_{max} = 21$, and a final model consisting of 15 basis functions). Unlike all other experiments, MARS showed a significant advantage in test set performance.

5.3.2.7 Comments Again, it must be stressed that the results reported above were chosen for comparison purposes between MARS and PNNs. Quite often, the overall probability of detection was higher for different results, especially for the PNN. Assigning a range of output values to each class for MARS is a difficult problem. Experiment 2C shows that MARS can perform better than the PNN with appropriate settings for the output class ranges; however, in most cases the PNN was easier to use and gave superior results.

5.4 Discussion

For the most part, MARS and FFNNs were able to approximate the above mappings to about the same level of accuracy (as measured by performance on the test sets). However, the greatest advantage of MARS over FFNNs is its fast training time. For example, we were able to run a full series of MARS tests (a scan of the M_{max} parameter) in the time it took to train one neural network for the 100 element synthetic eddy current training set.

One must also recognize the inherent parallelism in FFNN computing. Although most neural nets are simulated on sequential computers today, the future of VLSI neural network chips promises real parallelism. Thus, neural nets may regain a speed advantage over MARS and other sequential algorithms. But with the currently available neural network tools still operating sequentially, MARS is often a faster technique. Of course, one cannot rule out the future possibility of a parallel MARS implementation providing better run-times.

The strong mathematical foundation of MARS gives it another edge over neural networks. Final MARS models may be analyzed for relative variable importance

along with a variety of other interesting information. Neural network models are much more difficult to analyze and comprehend.

MARS also has the advantage of a fixed and predictable run-time. FFNNs (except the PNN) iterate through the training set until a certain condition is met. When this condition will actually occur is not predictable a priori.

For experimental applications like QNDE, one must recognize the challenge and difficulty inherent in selecting features to use as input to the network. This concern was of paramount importance in both ultrasonic classification problems. If the wrong features are selected, then one feature may dominate the entire training process. With MARS this is easily identified, but with neural networks, one may not be able to see this as easily.

MARS could be used as a general purpose classifier as shown above; however, we believe sizing is a more appropriate use for the technique. Forcing MARS into the role of classifier is sometimes clumsy, but the results are often competitive. Assigning a range of output values to each class is not easy, especially as the number of classes grows large.

As shown by ultrasonic classification problem two, the PNN FFNN is definitely a useful tool for classification. The results obtained for the submarine data were not of industrial quality; however, given the chosen features and 85% data classification accuracy, no method seems to perform significantly better. Improved feature selection techniques are necessary in order to more effectively analyze the data.

Taking into consideration all the factors discussed above including run-time, test set approximation accuracy, ease of use, and "interpretability," we recommend MARS as a generally better method for the given mapping approximation problems.

6. SUMMARY AND DISCUSSION

This thesis presented the MARS technique along with many feed-forward neural network paradigms. Experiments utilizing all the methods were presented and discussed. The purpose of this work was to analyze the applicability of these techniques for function approximation problems, especially in the application area of quantitative nondestructive evaluation.

In the introduction, various measurement criteria were proposed for comparing methods of building function approximators. In the area of run time, MARS almost invariably performed better than FFNNs. Nowhere was this more evident than in the eddy current synthetic data experiment that used a 4000 element test set. FFNNs are by definition highly parallel; thus, the sequential simulations used today are not a fair measure of the speed of the paradigm. However, MARS seems to lend itself to parallel implementation as well. With the extreme disparity in run time between MARS and FFNNs shown in this thesis, it seems reasonable to predict that a parallel version of MARS (if not the sequential version) will still run faster than most backpropagation-based FFNNs. This fact is the impetus for research into faster multilayer FFNN learning algorithms. With respect to ease of use, all methods seem to be about the same; however, MARS is more parameter insensitive than backpropagation and thus slightly easier to use. We found MARS and FFNNs to both be tolerant to noisy input

data; however, FFNNs always performed slightly better than MARS in the presence of noise. This statement is based mainly on the results of the ultrasonic classification problems in which a probabilistic FFNN is used. With respect to interpretability, MARS is unbeatable. The variety of analysis data that is inherent in the MARS approach gives it a firm mathematical basis for interpretation. FFNNs, on the other hand, still remain largely a black box when trying to understand their resulting models. Finally, most of our sizing experiments preferred MARS over FFNNs for test set approximation accuracy. This observation is most obvious in the 4000 element test set problem in the eddy current experiments.

Overall, MARS performed "better" than FFNNs on most of the QNDE applications. However, this statement cannot be generalized to all applications. A necessary area of research entails characterizing the features of functions that may make them suitable for FFNNs or MARS. We also see many possibilities in attempting to join the best features of MARS with the best features of FFNNs. Exactly how this might be done is an interesting problem.

Backpropagation-based FFNNs are one portion of the vast field of neural networking. A huge amount of work is being done on other neural network issues and paradigms not discussed in this thesis. This work includes other neural network paradigms [22, 26, 30, 35, 48], computational learning theory [60], the computability issues of neural networks [51], and dozens of other areas.

BIBLIOGRAPHY

[1] Agmon, S. "The Relaxation Method for Linear Inequalities." *Canadian Journal of Mathematics.* Vol. 6, No. 3 (1954): 382–392.

[2] Alpaydin, E. "GAL: Networks that grow when they learn and shrink when they forget." Technical Report No. 91-032. International Computer Science Institute, 1991.

[3] Ash, T. "Dynamic Node Creation in Backpropagation Networks." *Connection Science – Journal of Neural Computing, Artificial Intelligence and Cognitive Research.* Vol. 1, No. 4 (1989): 365–375.

[4] Auld, B. A., S. R. Jefferies, J. C. Moulder, and J. C. Gerlitz. "Semi-elliptical Surface Flaw EC Interaction and Inversion: Theory." *Review of Progress in Quantitative NDE.* ed. by D. O. Thompson and D. E. Chementi, Vol. 5, 383–393. New York: Plenum Press, 1985.

[5] Becker, S. and Y. le Cun. "Improving the Convergence of Back-Propagation Learning with Second Order Methods." *Proceedings of the 1988 Connectionist Models Summer School.* ed. by D. Touretzky, G. Hinton, and T. Sejnowski, 29–37. San Mateo, CA: Morgan Kaufmann, 1988.

[6] Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Belmont, CA: Wadsworth, 1984.

[7] Carley, L. R. "Presynaptic Neural Information Processing." *Neural Information Processing Systems.* ed. by D. Z. Anderson, 154–163. New York: American Institute of Physics, 1987.

[8] Chiou, C. -P. "Model-based ultrasonic flaw classification and sizing." PhD Dissertation. Iowa State University, 1990.

[9] Craven, P. and G. Wahba. "Smoothing noisy data with spline functions: Estimating the correct degree of smoothing by the method of generalized cross-validation." *Numerische Mathematik.* Vol. 31 (1979): 317–403.

[10] Cybenko, G. "Approximation by Superpositions of a Sigmoidal Function." *Mathematics of Control, Signals, and Systems.* Vol. 2, No. 4 (1989): 303–314.

[11] deBoor, C. *A Practical Guide to Splines.* New York: Springer-Verlag, 1978.

[12] Elliot, D. F. and K. R. Rao. *Fast Transforms: Algorithms, Analyses, Applications.* New York: Academic Press, 1982.

[13] Fahlman, S. E. "An Empirical Study of Learning Speed in Back-Propagation Networks." Technical Report No. CMU-CS-88-162. Carnegie Mellon University, 1988.

[14] Fahlman, S. E. "Faster-Learning Variations on Back-Propagation: An Empirical Study." *Proceedings of the 1988 Connectionist Models Summer School.* ed. by D. Touretzky, G. Hinton, and T. Sejnowski, 38–51. San Mateo, CA: Morgan Kaufmann, 1988.

[15] Fahlman, S. E. and C. Lebiere. "The Cascade-Correlation Learning Architecture." Technical Report No. CMU-CS-90-100. Carnegie Mellon University, 1990.

[16] Friedman, J. H. "Classification and Multiple Response Regression Through Projection Pursuit." Technical Report No. LCS012. Laboratory for Computational Statistics, Stanford University, 1985.

[17] Friedman, J. H. "Multivariate Adaptive Regression Splines." Technical Report No. 102. Laboratory for Computational Statistics, Stanford University, 1988.

[18] Friedman, J. H. "Adaptive Spline Networks." *Advances in Neural Information Processing Systems.* ed. by R. P. Lippmann, J. E. Moody, and D. S. Touretzky, Vol. 3, 675–683. San Mateo, CA: Morgan Kaufmann, 1991.

[19] Friedman, J. H. "Multivariate Adaptive Regression Splines." *The Annals of Statistics.* Vol. 19, No. 1 (1991): 1–67.

[20] Friedman, J. H. and W. Stuetzle. "Projection Pursuit Regression." *Journal of the American Statistics Association.* Vol. 76 (1981): 817–823.

[21] Funahashi, K. -I. "On the Approximate Realization of Continuous Mappings by Neural Networks." *Neural Networks.* Vol. 2, No. 3 (1989): 183–192.

[22] Grossberg, S. "Competitive learning: From interactive activation to adaptive resonance." *Cognitive Science.* Vol. 11 (1987): 23–63.

[23] Hartman, E. J., J. D. Keeler, and J. M. Kowalski. "Layered Neural Networks with Gaussian Hidden Units as Universal Approximations." *Neural Computations.* Vol. 2, No. 2 (1990): 210–215.

[24] Hecht-Nielsen, R. "Kolmogorov's Mapping Neural Network Existence Theorem." *Proceedings of the IEEE First International Conference on Neural Networks.* ed. by M. Caudill and C. Butler, Vol. III, 11-13. San Diego, CA: SOS Printing, 1987.

[25] Hecht-Nielsen, R. *Neurocomputing.* New York: Addison-Wesley, 1990.

[26] Hinton, G., D. Ackley, and T. Sejnowski. "Boltzmann machines: Constraint satisfaction networks that learn." Technical Report No. CMU-CS-84-119. Department of Computer Science, Carnegie Mellon University, 1984.

[27] Hoffgen, K. -U. and H. P. Siemon. "Approximation of Functions with Feedforward Nets." Report No. 346. Department of Computer Science, University of Dortmund, 1990.

[28] Honavar, V. and L. Uhr. "Experimental Results Indicate that Generation, Local Receptive Fields and Global Convergence Improve Perceptual Learning in Connectionist Networks." Technical Report No. 805. Computer Sciences Department, University of Wisconsin – Madison, 1988.

[29] Honavar, V. and L. Uhr. "A Network of Neuron-like Units That Learns to Perceive by Generation as Well as Reweighting of its Links." *Proceedings of the 1988 Connectionist Models Summer School.* ed. by D. Touretzky, G. Hinton, and T. Sejnowski, 472–484. San Mateo, CA: Morgan Kaufmann, 1988.

[30] Hopfield, J. J. "Neurons with graded response have collective computational properties like those of two-state neurons." *The National Academy of Sciences USA.* Vol. 81 (1984): 3088–3092.

[31] Hornik, K., M. Stinchcombe, and H. White. "Multilayer Feedforward Networks are Universal Approximators." *Neural Networks.* Vol. 2, No. 5 (1989): 359–366.

[32] Hush, D. R. and J. M. Salas. "Improving the Learning Rate of Backpropagation with the Gradient Reuse Algorithm." *Proceedings of the IEEE International Conference on Neural Networks.* Vol. 1, 441–448. San Diego, CA: IEEE, 1988.

[33] Jacobs, R. A. "Increased Rates of Convergence Through Learning Rate Adaption." *Neural Networks.* Vol. 1, No. 4 (1988): 295–307.

[34] Knopp, K. *Theory and Application of Infinite Series.* New York: Dover Publications, 1990.

[35] Kohonen, T. *Self-Organization and Associative Memory.* Berlin: Springer-Verlag, 1984.

[36] Kolmogorov, A. N. "On the Representation of Continuous Functions of Many Variables by Superposition of Continuous Functions of One Variable and Addition." *Dokl. Akad. Nauk USSR.* Vol. 114 (1957): 953–956.

[37] Mann, J. M. "Neural networks: Genetic-based learning, network architecture, and applications to nondestructive evaluation." Master's Thesis. Iowa State University, 1990.

[38] McCulloch, W. W. and W. Pitts. "A Logical Calculus of the Ideas immanent in nervous activity." *Bulletin of Mathematical Biophysics.* Vol. 5 (1943): 115–133.

[39] Minsky, M. L. and S. A. Papert. *Perceptrons - Expanded Edition.* Cambridge, MA: MIT Press, 1988.

[40] Motzkin, T. S. and I. J. Schoenberg. "The Relaxation Method for Linear Inequalities." *Canadian Journal of Mathematics.* Vol. 6, No. 3 (1954): 393–404.

[41] Oh, H. and S. C. Kothari. "Adaptation of the Relaxation Method for Learning in Bidirectional Associative Memory." Technical Report No. 91–25. Department of Computer Science, Iowa State University, 1991.

[42] Oh, H. and S. C. Kothari. "A New Learning Approach to Enhance the Storage Capacity of the Hopfield Model." *IEEE International Joint Conference on Neural Networks.* ed. by INNS, 2056–2062. New York: IEEE, 1991.

[43] Pao, Y. -H. *Adaptive Pattern Recognition and Neural Networks.* New York: Addison–Wesley, 1989.

[44] Peterson, B. A., S. Kothari, and L. W. Schmerr. "Conventional and Neural Network Techniques for Flaw Sizing and Classification in Quantitative Nondestructive Evaluation." *Proceedings of the First Iowa Space Conference.* ed. by C. -J. Chen, 133–142. Iowa City, IA: University of Iowa, 1992.

[45] Reinke, R. "Knowledge Acquisition and Refinement Tools for the ADVISE Meta-Expert System." Master's Thesis. University of Illinois at Urbana-Champaign, 1984.

[46] Rosenblatt, F. *Principles of Neurodynamics.* New York: Spartan Books, 1962.

[47] Rumelhart, D. E. and J. L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* Cambridge, MA: MIT Press, 1986.

[48] Sanger, T. "Basis-Function Trees as a Generalization of Local Variable Selection Methods for Function Approximation." *Advances in Neural Information Processing Systems.* ed. by R. P. Lippmann, J. E. Moody, and D. S. Touretzky, Vol. 3, 700–706. San Mateo, CA: Morgan Kaufmann, 1991.

[49] Schall, W. E. *Non-Destructive Testing.* London: Machinery Publishing Co., 1968.

[50] Sen, A. and M. Srivastava. *Regression Analysis: Theory, Methods, and Applications.* New York: Springer-Verlag, 1990.

[51] Siegelman, H. and E. D. Sontag. "Neural Nets are Universal Computing Devices." Technical Report No. SYCON-91-08. Rutgers Center for Systems and Control, 1991.

[52] Sietsma, J. and R. J. F. Dow. "Neural Net Pruning – Why and How." *Proceedings of the IEEE International Conference on Neural Networks.* Vol. 1, 325–333. San Diego, CA: IEEE, 1988.

[53] Simpson, P. K. *Artificial Neural Systems: Foundations, Paradigms, Applications, and Implementations.* New York: Pergamon Press, 1990.

[54] Song, S. -J. "Ultrasonic flaw classification and sizing." PhD Dissertation. Iowa State University, 1991.

[55] Specht, D. F. "Probabilistic neural networks." *Neural Networks.* Vol. 3 (1990): 109–118.

[56] Sprecher, D. A. "On the Structure of Continuous Functions of Several Variables." *Trans. Amer. Math. Soc.* Vol. 115 (1965): 340–355.

[57] Stornetta, W. S. and B. A. Huberman. "An improved three-layer, backpropagation algorithm." *Proceedings of the IEEE First International Conference on Neural Networks.* ed. by M. Caudill and C. Butler, Vol. II, 637–644. San Diego, CA: SOS Printing, 1987.

[58] Tawel, R. "Does the Neuron 'Learn' Like the Synapse?" *Advances in Neural Information Processing Systems.* ed. by D. S. Touretzky, Vol. 1, 169–176. San Mateo, CA: Morgan Kaufmann, 1989.

[59] Tolstov, G. P. *Fourier Series.* New York: Dover Publications, 1962.

[60] Valiant, L. G. "A Theory of the Learnable." *Communications of the ACM.* Vol. 27 (1984): 1134–1142.

[61] Werbos, P. J. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* Master's Thesis, Harvard University, 1974.

[62] Yang, J. and V. Honavar. "Experiments with the Cascade-Correlation Algorithm." Technical Report No. 91-16. Department of Computer Science, Iowa State University, 1991.

ACKNOWLEDGEMENTS

First, I would like to thank the members of my committee, Dr. Suresh Kothari, Dr. Vasant Honavar, and Dr. Les Schmerr, for helping and encouraging my research every step of the way. I especially thank Dr. Kothari for the many years of friendship and academic support he has given me, especially during my graduate years. I appreciate the help and cooperation of many individuals at the Iowa State Center for Nondestructive Evaluation including Bob Forouraghi, Jim Mann, Chien-Ping Chiou, and Sung-Jin Song. Discussions with Jeff Clary, Heekuck Oh, and Jim Lathrop were invaluable in formulating my research.

I greatly appreciate the funding provided by the NASA Space Grant Graduate Fellowship Program/Iowa Space Grant College Consortium for this research. The administration of the program was absolutely superb.

The following individuals have my thanks for providing valuable materials: Vasant Honavar for the backpropagation simulation code, Jerome Friedman for the MARS 3.5 code and sample driver, Jim Mann for the eddy current data, and Sung-Jin Song and Chien-Ping Chiou for the ultrasonic data and PNN code. This work was accomplished using resources from the Iowa State University Computer Science Department, the Center for Nondestructive Evaluation at Iowa State University, and Project Vincent of Iowa State University.

Of course, I must thank my parents, Ruth and Ted Peterson, for their support and encouragement in all areas of my life.

Last but by no means least, I would like to thank my forever patient wife Jennifer. Her understanding, love, and support were limitless throughout the many nights and weekends I spent holed up in my office. I can never express adequately my deep thanks for her commitment to me.