APPLICATIONS OF NEURAL NETWORKS IN
IC LITHOGRAPHY

by

MANISHA MAHENDRA, B.S.E.E.

A THESIS

IN

ELECTRICAL ENGINEERING

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

Approved

May, 1995

84615
805
T3
1995
No. 19
C.2

AEO-34
JAC 10/

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to Dr. Thomas Krile for all his guidance and assistance during the course of this work. I would also like to thank the members of my committee, Dr. William Oldham and Dr. Donald Wunsch, as their suggestions and help were invaluable throughout the endeavor.

I would like to thank all my friends and colleagues for their support and friendship as it has made my stay here an enjoyable and learning experience. Finally, I would like to thank my parents, brothers, and sister-in-law, for their never ending love, support and encouragement. Without them, none of this would have been possible.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	ii
LIST OF TABLES	v
LIST OF FIGURES.....	vi
CHAPTER	
I. INTRODUCTION	1
II. THE LITHOGRAPHY MODEL	3
III. THE NEURAL NETWORKS	13
3.1. Introduction.....	13
3.2. The Back-Propagation Neural Network	14
3.2.1 The Learning Rule.....	16
3.2.2 The Basic Delta Rule.....	16
3.2.3 Activation Function.....	19
3.2.4 The Levenberg-Marquardt Algorithm.....	20
3.2.5 Matlab.....	23
3.3 Abductive Modeling Network.....	23
3.3.1 The GMDH Algorithm.....	24
3.3.2 Algorithm Structure	25
3.3.3 The AIM (Abductive Information Modeler) Software.....	27
3.3.4 The Predicted Square Error (PSE).....	28
3.3.5 Development of a Network Model.....	29
3.3.6 Application of AIM	30
IV. EXPERIMENTAL RESULTS AND DISCUSSION	31
4.1 Back-Propagation	32
4.2 Abductive Modeling	57
V. CONCLUSION.....	83
REFERENCES.....	85
APPENDIX	
A. PROGRAM FOR GENERATING PATTERNS	86
B. SAMPLE MATLAB PROGRAM	88

C. SAMPLE AIM PROGRAM 90

D. DETAILED GUIDE TO USING AIM 99

E. STATISTICS OF AIM PROGRAM..... 101

LIST OF TABLES

4.1 Values used in generating patterns for training and testing the Neural
Networks..... 33

4.2 Cases trained for in 2-D sample space 46

E.1 Statistics of Aim Program 102

LIST OF FIGURES

2.1 Experimental Setup for IC Lithography	4
2.2 2-D wafer view showing variations	5
2.3 Some 1-D Mask Patterns	5
2.4 1-D mask patterns without maximum factor	8
2.5 1-D patterns with intensity of 25	9
2.6 Fourier transform of 128 points of pattern.....	10
2.7 Fourier transform of patterns at twice the sampling rate	11
2.8 1-D mask patterns with 2X sampling rate (30 points)	12
3.1 Diagram of Neuron	13
3.2 A two-layered neural network.....	15
3.3 Error surface in delta rule in a 2-D weight space	17
3.4 Activation functions	19
3.5 Error Minimization with Gradient Descent.....	21
3.6 Stopping criterion for GMDH algorithm	27
4.1 Train and test data for double exposure pattern with one output network.....	34
4.2 Train and test data with output d1 in network in 0.38-2 range	36
4.3 Train and test data with output d1 in network in 2-3 range.....	37
4.4 Train data for network with two outputs, where the outputs are periods and one output is constant	38
4.5 Train data for network with two outputs, where the outputs are amplitudes and one output is constant.....	40
4.6 Test data for network with two outputs, where the outputs are amplitudes and one output is constant.....	41
4.7 Train data for network with two outputs, where the outputs are phases and one output is constant	42

4.8	Test data for network with two outputs, where the outputs are phases and one output is constant	43
4.9	Plot of 2 parameters' sample space	44
4.10	Training data for network with two varying outputs	45
4.11	Error plot for training network with two outputs, where the outputs are periods	47
4.12	Train data for network with two outputs, where the outputs are periods (case 1).....	48
4.13	Test data for network with two outputs, where the outputs are periods (case 1).....	49
4.14	Test data for network with two outputs (case 2)	50
4.15	Test data for network (case 2) with one output constant	51
4.16	Test data for network with outputs e1 and d3 (case 3)	52
4.17	Test data for network with outputs e1 and θ_1 (case 4)	53
4.18	Test data for network with outputs e1 and e2 (case 5).....	54
4.19	Train data for network with outputs θ_1 and θ_2 (case 6).....	55
4.20	Test data for network with outputs θ_1 and θ_2 (case 6)	56
4.21	Train and test data for double exposure pattern with one output network.....	58
4.22	Network for Fig. 4.21	59
4.23	Network for Fig. 4.24	59
4.24	Train data for network with two outputs, where the outputs are amplitudes and one output is constant	60
4.25	Test data for network with two outputs, where the outputs are amplitudes and one output is constant	61
4.26	Network for Fig. 4.27	62
4.27	Plot of test data = train data - output θ_1 for node 1 where, node 2 output is constant valued θ_3	62
4.28	Plot of test data $\neq$ train data - output θ_1 for node 1 where, node 2 output is constant valued θ_3	63

4.29	Network for case 1	63
4.30	Train data for network with two outputs, where the outputs are periods (case 1)	65
4.31	Test data for network with two outputs, where the outputs are periods (case 1)	66
4.32	Network obtained for case 2	67
4.33	Train data for network with two outputs (case 2)	68
4.34	Test data for network (case 2)	69
4.35	Test data for networks (case 2) with one output constant	70
4.36	Networks for case 3	71
4.37	Test data for networks (case 3) with amplitude-period pattern	72
4.38	Network for case 4	73
4.39	Train data for networks in case 4	74
4.40	Test data for networks with case 4	75
4.41	Test data for networks for case 5	76
4.42	Networks for case 5	77
4.43	Networks for case 6	77
4.44	Train data for networks for case 6	78
4.45	Test data for networks for case 6	79
4.46	Plot of test results for showing how the test results varied from the actual results	81
4.47	Input patterns showing how the actual results varied from the desired	82

CHAPTER I

INTRODUCTION

The continuing trend in the semiconductor industry is towards ever smaller critical dimension integrated circuit (IC) structures on ever larger die sizes. To realize such smaller critical dimensions, there is a need for improvement in the field of lithography.

One method that has shown encouraging results without an extensive development of resources is multiple exposure interferometric lithography [1]. This method uses the interference effects between two coherent laser beams to create grating patterns in photoresist. With the addition of multiple exposures, and variations in amplitude, period, phase, and orientation, highly complex IC mask patterns have been developed. The advantage of this method is that it overcomes the traditional limits to both resolution and depth of field. However, since the photoresist exposure and development process is highly nonlinear, there is no known method for determining the exposure parameters, given the desired mask structure.

This thesis will discuss how neural networks can be used to find the values of the period, phase, and amplitude at each exposure, given an input of desired line profiles or mask patterns. Thus the neural networks can be considered as learning the solution to a multiparameter nonlinear inverse problem.

Two types of neural networks were applied to the problem and both will be discussed and compared to see which one is more effective on this problem. The two types of neural networks that were applied were back propagation using the Levenberg-Marquardt algorithm, and the abductive polynomial modeling network, a close relative of the Group Method for Data Handling (GMDH). A commercial software package known as AIM (Abductive Information Modeler) was used to implement the abductive

polynomial network, while the back-propagation neural network was applied using the neural network toolbox in Matlab.

First, the lithography model and experimental setup will be discussed in Chapter II. Chapter III will discuss the neural networks by first giving an overview of neural networks and then discussing the algorithms for each type of network used here. A brief description of the software used will also be given. Chapter IV will present the various experimental results of the two approaches, and the results will be discussed and compared. The thesis will end with a conclusion and summary of the work done and ideas on further work.

CHAPTER II

THE LITHOGRAPHY MODEL

The cumulative photoresist exposure pattern, E , resulting due to multiple laser beam exposures at different periods, phases, and amplitudes is given as follows:

$$E(\rho) = \sum_i e_i [1 + \cos(q_i \cdot \rho + \theta_i)]. \quad (2.1)$$

Here ρ is the position vector, e_i is the amplitude of the i th exposure, θ_i is the phase, and q_i is the wave vector or spatial frequency for the i th exposure, where for a single exposure pattern,

$$q = \frac{2\pi}{d}, \text{ with } d = \text{period.}$$

The model we used for the photoresist [1] is as follows and incorporates both exposure and development. The normalized photoresist thickness for a positive photoresist is

$$t = \max \left[1 - \left(\frac{1 - \exp[-kE(\rho)]}{1 - \exp(-kE_0)} \right)^n, 0 \right] \quad (2.2)$$

where

k = photoresist sensitivity,

E_0 = minimum exposure to clear photoresist after development, and

n = coordination number that affects the development rate.

For the above model, some values for the terms were taken from [2], and the rest were found by experimenting with the model and comparing results to those shown in [1]. As

can be seen by the equation, there are severe nonlinearities present in this model, and so neural networks are an ideal tool to use in finding the parameters. Before going further, a brief overview of an experimental setup for the IC lithography process will be given. This will enable a better visual idea of the actual process

Fig 2.1 shows an experimental setup for the wafer fabrication process [1]. As the diagram illustrates, the pattern is produced by the interference effects between two coherent laser beams, where one beam is being created by the reflection of the original beam in a mirror in the setup. The resulting grating pattern after one exposure is shown in Fig. 2.2. Complex patterns are generated by varying the intensity (amplitude) of the laser beam hitting the wafer, as well as the frequency and phase during multiple exposures. The frequency (or period, d) is varied by rotation of the mirror as shown in Fig. 2.1. This changes the distances in the grating patterns (Fig. 2.2). The grating direction vector, ρ , is varied by rotating the wafer in the X-Y plane. The phase is varied by moving the XY translation in the setup. As the wafer is moved, a phase shift occurs (see θ in Fig. 2.2). The amplitudes are changed using the amplitude mask, which varies the

Fig. 2.1 Experimental Setup for IC Lithography

Fig. 2.2 2-D wafer view showing variations

intensity of the lasers on the wafer. With multiple exposures of the grating patterns, complex patterns are generated.

Fig. 2.3 illustrates some of the 1-D mask patterns that were generated using the above models for the process (eq. 2.1 and 2.2). As can be seen in the figure, 15 samples in the ρ direction have been taken for the plots.

(a) $d_1 = 0.51 = d_3$ and three exposures

Fig. 2.3 Some 1-D Mask Patterns

(b) $d1 = d3 = 1.5$ and three exposures

(c) $d1=0.51$ and two exposures

Fig. 2.3 Continued

These plots were generated using eq. 2, where $E_0=89$, $n=5.0$, $k=0.015$ and $E(p)$ was found using eq. 2.1. The values used in eq. 2.1 were as follows for the above patterns (Fig. 2.1a and b) $e_1 = e_2 = e_3 = 65$; $\theta_1 = 0$, $\theta_2 = \frac{\pi}{2}$, $\theta_3 = 0$; $d1=0.5-1.5$, $d2=2.0$, $d3=0.5-1.5$. For the double exposure pattern, everything was the same except that the third exposure parameters were not included.

During the course of the experiments, the networks were able to train well for one varying parameter (d1, for example) with 15 samples, especially when there were only double exposure patterns, as they generated richer patterns. When training began on two varying parameters for triple exposure patterns (d1 & d3, for example), lower amplitudes had to be chosen as the data was not varied enough for the networks to train on otherwise. The reason was that most of the pattern is washed out at such higher intensities. Fig. 2.4 shows pictures of the above patterns (Fig. 2.3a and b) where the maximum factor that limits the patterns to between 0 and 1 was removed. They are shown at an intensity of 65 and most of the pattern is seen below 0. Therefore, the intensity was varied and finally lowered to 25. The resulting graphs are shown in Fig. 2.5.

As more complex patterns were created, more problems were encountered and so the sampling rate of the patterns was studied. Fig. 2.6 shows a Fourier transform of a typical sample, where 128 points were taken centered around zero. The transform does not show the spectrum decaying out at the high frequencies, suggesting some aliasing.

Therefore, the sampling rate was doubled and the Fourier transform taken again. Fig. 2.7 shows the transformed pattern, which does seem to decay out towards the corners. Fig. 2.8 shows mask patterns generated with the finer sampling rate. Therefore, for the same distance, 30 points were sampled instead of 15. All this illustrates how important it is to understand the data when training the networks. The next chapter will discuss the neural networks.

(a) $d_1 = d_3 = 0.51$

(b) $d_1 = d_3 = 1.5$

Fig. 2.4 1-D mask patterns without maximum factor.

(a) $d1 = d3 = 0.51$ with $e1 = e2 = e3 = 25$

(b) $d1 = d3 = 1.5$ with $e1 = e2 = e3 = 25$

Fig. 2.5 1-D patterns with intensity of 25

Fig. 2.6 Fourier transform of 128 points of pattern

Fig. 2.7 Fourier transform of patterns at twice the sampling rate

(a) $d_1 = d_3 = 0.51$

(b) $d_1 = d_3 = 1.5$

Fig. 2.8 1-D mask patterns with 2X sampling rate (30 points)

CHAPTER III

THE NEURAL NETWORKS

3.1. Introduction

Neural networks are information processing systems that have been inspired by biological neural systems such as the human brain. The artificial neural network mathematical models of the biological networks are based on the following assumptions. First, information processing occurs at many simple elements called neurons. Secondly, signals are passed between neurons over connection links and each connection link has an associated weight which, in a typical neural network, multiplies the signal transmitted. Finally, each neuron applies an activation function to its net input (sum of weighted input signals) to determine its output signal. Fig. 3.1 shows a simple diagram of a neuron with inputs $x_1, \dots, x_n$, weights $W_1, \dots, W_n$ and an activation function, F .

Fig. 3.1 Diagram of Neuron

The strength of artificial neural networks lies in their ability to learn and generalize, from experience, the solution to a problem which would not have been possible using conventional computation methods. Learning means that the neuron changes its input/output behavior in response to the environment. The network learns by changing the weights applied to the inputs [3].

Learning in neural networks can be supervised or unsupervised. Supervised learning means the network is directed by a set of correct answers as to what path to follow to get the best solution. Basically, a set of training patterns is the input and a set of desired targets is the output from which the network learns. In unsupervised learning, there is no such guide and the network does not exactly know what the correct response should be. Both types of networks used in this thesis were supervised learning networks.

A neural network can also be either single-layered or multi-layered. Single-layered networks are usually used with linearly separable inputs, while multi-layered networks are implemented for linearly nonseparable inputs [4]. Multi-layered neural networks can also be implemented in nonlinear parameter estimation (function fitting). Since the problem addressed in this thesis is nonlinear parameter estimation, multi-layered networks were used.

Now that a brief overview of neural networks has been given, we will move on with an explanation of both the Back-propagation neural networks and the abductive modeling networks.

3.2. The Back-Propagation Neural Network

The back-propagation neural network is one of the most studied neural networks as it is relatively simple and has the capability to learn sequentially from training patterns. It is a supervised learning multi-layered network that uses the generalized delta learning rule as its training rule.

The training of a network by back-propagation involves three stages: the feedforward of the input training pattern, the calculation and back-propagation of the associated error, and the adjustments of the weights. After training, application involves only the computations of the feedforward phase [4].

Fig. 3.2 presents the interconnection architecture of a back-propagation network. It consists of the inputs $Z_1, \dots, Z_I$, the hidden layer weights $V_{11}, \dots, V_{JI}$, the hidden layer outputs $y_1, \dots, y_J$, the output layer weights $W_{11}, \dots, W_{KJ}$, and the output layer with outputs $o_1, \dots, o_K$. Therefore, it is a two-layered neural network. As previously mentioned, the power of having a hidden layer is that nonlinearly separable problems, and nonlinear function-fitting problems, can be solved using it along with the right set of connections and the right number of hidden units.

Fig. 3.2 A two-layered neural network

3.2.1 The Learning Rule

The basic back-propagation implements the delta learning rule with the gradient (or steepest) descent procedure for error minimization.

In the delta learning rule, if a pattern is submitted for classification and its output is determined to be erroneous, the synaptic weights as well as the thresholds are adjusted so that the current least mean square error is reduced. The input/output mapping, comparison of target and actual values, and adjustment, if needed, continue until all mapping examples from the training set are learned within an acceptable overall error. Mapping is usually cumulative and computed over the full training set. First, the basic delta learning rule will be presented, followed by the generalized delta rule [3].

3.2.2 The Basic Delta Rule

The basic rule for changing weights is given by

$$\Delta_p w_{ij} = \eta \delta_{ok} x_j = -\eta \frac{\partial E_p}{\partial w_{ij}} \text{ for } j = 1, 2, \dots, n, \quad (3.1)$$

where $\delta_{ok} = (d_{pi} - o_{pi}) f'(net_i),$

d_{pi} is the desired value for the i th component of the output for input pattern p , and o_{pi} is the actual output for pattern p . The term $f'(net)$ is the derivative with respect to net of the activation function $f(net)$ computed for $net_i = w_i x$, and x_j is the j th input for pattern p . $\Delta_p w_{ij}$ is the change to be made to the weight from the j th input to the i th node for pattern p .

The delta learning rule implements the gradient descent in error space as follows. Let $E = \sum E_p$ be the total error, and

$$E_p = \frac{1}{2} \sum_i (d_{pi} - o_{pi})^2 \quad (3.2)$$

be the error for pattern p . The derivative of the error with respect to the weight is $\frac{\partial E_p}{\partial w_{ij}}$, which is proportional to $\Delta_p w_{ij}$, as prescribed by the delta rule [4].

A neural network without a hidden layer has an error surface shaped like a "bowl," in n -dimensional space, as shown in Fig. 3.3 for the $n = 2$ case, with only one minimum. The delta rule moves a set of weights from somewhere on the surface of the bowl towards the bottom. It does this by moving along the negative gradient of the bowl. The constant η is a measure of the speed of convergence towards the minimum error. A large value of η causes faster movement to the bottom but also causes the weights to oscillate around the minimum. A small value of η does not cause weights to oscillate, but results in a long time for the network to converge.

Fig 3.3. Error Surface in Delta rule in a 2-D weight space

For multi-layered neural networks, the delta rule is modified and the generalized delta rule is developed. The following equations summarize these modifications.

For the hidden layer (see Fig. 3.2),

$$\Delta v_{ji} = \eta \delta_{yj} z_i = -\eta \frac{\partial E}{\partial v_{ji}} \text{ for } j=1,2,\dots,J, \text{ and } i=1,2,\dots,I, \quad (3.3)$$

where

$$\delta_{yj} = f'_j(\text{net}_j) \sum_{k=1}^K \delta_{ok} w_{kj}, \text{ for } j=1,2,\dots,J.$$

From this, the modified weights of the hidden layer are now

$$v_{ji}' = v_{ji} + \eta \delta_{yj} z_i \quad \text{for } j=1,2,\dots,J,$$

which becomes : (3.4)

$$v_{ji}' = v_{ji} + \eta f'_j(\text{net}_j) z_i \sum_{k=1}^K \delta_{ok} w_{kj} \text{ and } i=1,2,\dots,I,$$

where v_{ji} is a weight to the hidden node j from the input pattern component z_i , and w_{kj} is a weight to the output node k from the hidden node.

The modified weights for the output layer are

$$\begin{aligned} w_{kj}' &= w_{kj} + \eta \delta_{ok} y_j \\ &= w_{kj} + \eta (d_k - o_k) f'_k(\text{net}_k) y_j, \end{aligned} \quad (3.5)$$

where y_j , and o_k are the output components of hidden and output layers, respectively.

$f(\text{net})$ and $f'(\text{net})$ are the activation function and its derivative [4].

3.2.3 Activation Function

The activation functions that can be used with the generalized delta rule are continuous functions. The activation functions used in the neural networks here were the nonlinear unipolar sigmoidal function, shown in Fig. 3.4(a), which goes from 0 to 1 and the linear transfer function, shown in Fig. 3.4(b) [5]. The function in Fig. 3.4(a) is written as

$$f(net) = \frac{1}{1 - \exp(-\lambda net)} \quad (3.6)$$

(a)

(b)

Fig. 3.4 Activation functions

where λ is the gain determining the slope at the origin, while the function in Fig. 3.4(b) is

$$f(net) = net. \quad (3.7)$$

Given the sigmoidal activation function, the error term δ_{ok} becomes:

$\delta_{ok} = (d_{pi} - o_{pi})(1 - o_{pi})o_{pi}$, while the error term δ_{yj} becomes

$$\delta_{yj} = (1 - y_j)y_j \sum_{k=1}^K \delta_{ok}w_{kj}. \quad (3.8)$$

For the linear transfer function, $f(\text{net})=1$. Therefore, $\delta_{ok} = (d_{pi} - o_{pi})$, while the error term δ_{yj} becomes

$$\delta_{yj} = \sum_{k=1}^K \delta_{ok}w_{kj}. \quad (3.9)$$

As can be seen, the sigmoidal function squashes the inputs to a 0-1 range, while the linear activation function takes on any value. The sigmoidal function was implemented in the hidden layer, while the linear function was implemented in the output layer. This was done to enable the best mapping from the inputs to the outputs for our parameter estimation problem.

3.2.4 The Levenberg-Marquardt Algorithm

Although the back-propagation algorithm using the generalized delta rule with gradient descent is a good method to train a neural network, it is not very time efficient. The convergence rate can be very slow, especially with a complex problem. In order to improve the speed and general performance of the back-propagation network, optimization techniques such as the Levenberg-Marquardt algorithm can be implemented. The Levenberg-Marquardt technique is a more sophisticated method than gradient descent as it incorporates both gradient descent and a technique called the Gauss-Newton (inverse Hessian) method in error minimization. The Levenberg-Marquardt algorithm

varies between the Gauss-Newton method and the gradient descent method. The latter method is used far from the minimum, switching continuously to the former as the minimum is approached [6].

As already shown, gradient descent is the technique where parameters such as weights and biases are moved in the opposite direction to the error gradient. Each step down the gradient results in smaller errors, till the error minimum is reached. It is one of the most fundamental procedures for minimizing a differentiable function of several variables. This method usually works quite well during the early stages of the optimization process, depending on the point of initialization. However as a stationary point is approached, the method usually behaves poorly, taking small nearly orthogonal steps. This zigzagging phenomenon is illustrated in Fig. 3.5, where zigzagging occurs along the valley shown by the dotted lines [7]. This poor convergence in the later stages is due to the fact that gradient descent employs a linear approximation to the function.

Fig. 3.5 Error Minimization with Gradient Descent

On the other hand, the method of Gauss-Newton (inverse Hessian method) is a procedure that deflects the gradient descent direction by premultiplying it by the inverse of the Hessian matrix [7]. This operation is motivated by finding a suitable direction for the quadratic approximation to the function rather than by finding a linear approximation to the function, as in the gradient descent search.

The Gauss-Newton method can be interpreted as a gradient descent algorithm with affine scaling. Also, it can be stated that the Gauss-Newton method works well when initialized close to the minimum only.

Since both methods work well in minimizing the error at different times, the Levenberg-Marquardt algorithm varies between them. The update rule for the weights can be given as

$$\Delta w = (J^T J + \mu I)^{-1} J^T e \quad (3.10)$$

where J is the Jacobian matrix of the derivatives of each error to each weight, μ is a scalar and e is the error vector. If μ is very large, the above expression approximates gradient descent, while if μ is very small, it approximates the Gauss-Newton method. The parameter μ automatically varies, starting with a large value when the algorithm starts, and becoming smaller as the algorithm gets closer to the minimum.

Implementing the above method in back-propagation with the given activation functions does give good results faster and makes it a powerful application tool in neural networks. This section will now be concluded by discussing the software used to implement the back-propagation neural network.

3.2.5 Matlab

Matlab is a very powerful commercially available software package that has the ability to simulate a number of programs in a variety of areas of work, such as image processing, control systems, digital signal processing and neural networks. The neural network toolbox [5] implements learning rules for a variety of training functions. The following list will give a brief description of some of the commands that were implemented:

initff- used to initialize the weights and biases to some random values.

trainlm- used to train the network using the Levenberg-Marquardt algorithm.

simuff- used to simulate the feed-forward network after training and getting the weights.

logsig- the sigmoidal activation function.

purelin- the linear transfer function.

Using commands such as these, a Matlab program can be written to simulate a back-propagation neural network, given the data to train and test on.

3.3 Abductive Modeling Network

The abductive modeling network is based on the theory of GMDH (group method of data handling)[8], abduction, and neural networks. It should be noted that abductive networks are not the typical kind of neural networks discussed in the introduction of this chapter. The way they vary will be illustrated here.

The network is composed of a layered feed-forward network that computes polynomial functions of a set of parallel inputs in order to generate an output. It is similar to back-propagation neural networks only in this aspect. Also, the nodes used here are fewer and more powerful. The main theory behind abductive networks is the fact that

"mathematical functions are powerful representations of knowledge, and the functions are best modeled using a network structure " ([9] p.2-1).

This kind of network was first developed because of the difficulty of modeling complex systems when only the data is known and other things are not known by the modeler. A Russian mathematician and cyberneticist, A. G. Ivakhnenko, introduced a method, based on the perceptron, that allows the modeler to build a complex model of a system without any a priori knowledge, except the data. This method was the GMDH algorithm.

3.3.1 The GMDH Algorithm

The GMDH algorithm is based on the theme of evolution and survival of the fittest, where one starts with a basic set of equations that are then separated and used in order to develop a better model, according to the usefulness of each equation or function. The models are built up by looking at the data and desired results.

For each pair of input variables x_i and x_j , and the desired output y , we have a model that predicts y . After getting these equations, we have to decide which ones to save by looking at which ones best predict y . From the quadratic equations obtained, new independent observations are generated. They are then combined to get other equations. The best estimates are selected and the process is continued until the regression equations begin to have a poorer predictability power. This will happen as the equations are tested with a new set of observations and it will be seen that the model is becoming over-specialized. We pick the best estimate of the quadratic polynomials in that generation. We now have an estimate of y as a quadratic function of two variables, which are each also quadratic estimates of other previous layer variables and so on, resulting in a multinomial estimate of y in terms of the original values [8].

Eq. 3.11 shows the complicated polynomial that can be generated using this procedure, called the Ivakhnenko polynomial, which relates m input variables, $x_1, x_2, \dots, x_m$ to a single output variable y .

$$y = a + \sum_{i=1}^m b_i x_i + \sum_{i=1}^m \sum_{j=1}^m c_{ij} x_i x_j + \sum_{i=1}^m \sum_{j=1}^m \sum_{k=1}^m d_{ijk} x_i x_j x_k + \dots \quad (3.11)$$

Using the GMDH technique, this polynomial will not have the large number of coefficients and combinations that standard regression techniques would have produced, as it has discarded most of them. The GMDH algorithm uses a multi-layered perceptron-type network structure to develop the polynomial, hence its resemblance to other neural networks.

3.3.2 Algorithm Structure

The basic GMDH algorithm consists of four major steps which are shown as follows [8].

1. Subdivide data into training and testing sets:

The data is subdivided so that the training data can be used to estimate the parameters for the polynomials while the testing data is used to select the best equations in each layer. The reason for this division of the data points into two sets is as follows. If one wished to find the least-squares polynomial $y = a_0 + a_1 x + a_2 x^2 + \dots + a_m x^m$ of some degree m that best fits a set of data points $(x_1, y_1), \dots, (x_n, y_n)$, one can make the polynomials pass exactly through each point by picking $m = n-1$. However, that does not mean the polynomial has modeled the data well since if we were to compute the sum of squares of errors from the polynomial and a new set of checking data, we could get a large sum of squares error (e.g., due to noisy data). What we want is a polynomial created from

the training data set and checked against the testing data set. This is called the polynomial of optimal complexity.

2. Evaluate polynomial at all n data points:

Here, all the independent input variables $x_1, x_2, \dots, x_m$ are taken two at a time to generate a second order equation (Eq. 3.12) that best fits the dependent observations y_i ($i= 1, \dots, N_{tr}$) in the training set. These equations of $\hat{y}_{il}$ become the set of functions for the next layer variables (these new variables should predict y better than the previous values). Then, the polynomials are evaluated for N_t test data sets and stored in a new array Z , where $z_{ij} = \hat{y}_{il}$ and $i=1, \dots, N_t$ and $j=1$.

$$\hat{y}_{il} = A + Bx_{ij} + Cx_{ik} + Dx_{ij}^2 + Ex_{ik}^2 + Fx_{ij}x_{ik} \quad (3.12)$$

where

$$l = 1, \dots, \frac{m(m-1)}{2} \text{ and } i = 1, \dots, N_{tr} \text{ and } j, k = 1, \dots, m \text{ with } j \neq k \text{ and } j < k.$$

3. Screen out least effective variable:

From all the equations (Z), those that best predict y are selected. The selection criterion implemented is the normalized root mean square error (or regularity criterion) r_j where for N_t testing points

$$r_j^2 = \frac{\sum_{i=1}^{N_t} (y_i - z_{ij})^2}{\sum_{i=1}^{N_t} y_i^2} \quad \text{where } j = 1, \dots, \frac{m(m-1)}{2}. \quad (3.13)$$

The equations in Z whose regularity criterion, r_j , satisfies $r_j < R$ (where R is some threshold value chosen in advance) are chosen to pass to the next layer.

4. The stopping rule:

The minimum value for r_j from step 3 is taken and called Rmin. It is then plotted on a graph as shown in Fig. 3.6. If the current value of Rmin is less than the previous value, we go back to step 2, else if it is greater, we assume that the final value of Rmin has been passed and we stop the algorithm and use the results from the preceding generation.

Fig. 3.6 Stopping criterion for GMDH algorithm

Now that an overview of abductive modeling and its basic structure have been given, the actual software used will be presented, along with how it is more complex than the GMDH algorithm.

3.3.3 The AIM (Abductive Information Modeler) Software

AIM is a sophisticated modeling software based on the GMDH algorithm and the predicted square error (PSE) [9]. It uses the functions in a layered feed forward network structure to model the inputs, where the functions are at each node of a layer. A

combination of these inputs is computed and then the best functions are retained and passed on to the next layer. Which elements are kept is determined by the PSE. The network is halted when the model ceases to improve with the addition of new elements. The model with the least PSE is the final product.

3.3.4 The Predicted Square Error (PSE)

The PSE criterion was derived by A. R. Barron and is a heuristic measure of the expected network squared error for independent data not in the training database. It can be written as

$$PSE = FSE + KP \quad (3.14)$$

where FSE = fitting squared error of the model on the training data and

KP = complexity penalty.

The KP is determined by the equation

$$KP = CPM * 2 \left(\frac{K}{N} \right) s_p^2 \quad (3.15)$$

where K, N and s_p^2 are determined by the database of examples used to synthesize the network and CPM. Here K is the total number of coefficients, N is the number of training patterns, and s_p^2 is an a priori estimate of the true unknown model error variance.

As N goes up or s_p^2 goes down, AIM can fit the data with more confidence and more complexity. The CPM is the complexity penalty multiplier which helps find the best value for KP. A higher value of CPM increases the impact of the penalty term, thus

resulting in a less complex network, whereas a lower value of CPM reduces the penalty and allows for a more complex network to be built [8].

As it implements the PSE criterion to select the model, the input data does not have to be split and used separately for training and testing the selection process. All the data can be used in the selection process for training and so it works well with less data than GMDH.

3.3.5 Development of a Network Model

AIM develops the layered networks using various elements or nodes which are described below. The algebraic form of each node will be shown, where w_n are the coefficients determined by AIM and x_n are the input variables. There are seven basic nodes which are shown as follows:

1. Singles- (1 input variable) = $w_0 + (w_1x_1) + (w_2x_1^2) + (w_3x_1^3)$.

2. Doubles- (2 input variables)

$$= w_0 + (w_1x_1) + (w_2x_2) + (w_3x_1^2) + (w_4x_2^2) + (w_5x_1x_2) + (w_6x_1^3) + (w_7x_2^3).$$

3. Triples- (3 input variables)

$$= w_0 + (w_1x_1) + (w_2x_2) + (w_3x_3) + (w_4x_1^2) + (w_5x_2^2) + (w_6x_3^2) + (w_7x_1x_2) + (w_8x_1x_3) + (w_9x_2x_3) + (w_{10}x_1x_2x_3) + (w_{11}x_1^3) + (w_{12}x_2^3) + (w_{13}x_3^3).$$

4. Linear Elements-(take weighted sum of all outputs of previous layer) as shown

$$= w_1x_1 + w_2x_2 + w_3x_3 + \dots + w_nx_n.$$

5. Normalizers- normalize input variables using mean-sigma normalization.

6. Unitizers- Convert range of network outputs to range with mean and variance of desired output values.

7. Wire Elements- they are used in networks consisting of only normalizers and unitizers.

The singles, doubles and triples are all third-order polynomials and all their terms do not have to be used [9]. The triples are also 3-input variables unlike GMDH which has only up to 2 variables. Also, the output of any given node can be fed into subsequent layers together with the original input variables, whereas the GMDH algorithm only allows variables from the previous layer. Next, we will discuss how AIM is used to implement a problem.

3.3.6 Application of AIM

AIM is applied to any problem using the following steps:

1. Create a database-- Given a database of examples, AIM builds models. Therefore, data must be imported into the application and stored so that AIM can interpret it correctly.
2. Build an AIM model-- AIM synthesizes the data to build a model from it. The user can change the CPM, and other variables, to adjust the network, but it is mostly automated.
3. Evaluate Model Performance-- After the synthesis of the model, the user can evaluate its performance using AIM's evaluation package which performs a statistical analysis of the results, given the inputs.
4. Apply the model-- Once the model has been developed, it can be implemented by either converting it into standard 'C' code for use in applications or by using the Query function in AIM to give an answer to a certain number of inputs.

Using both Abductive Modeling and Back-propagation, various results were produced. The next chapter will discuss them, followed by the conclusion.

CHAPTER IV

EXPERIMENTAL RESULTS AND DISCUSSION

This chapter will present the experimental results found using both abductive modeling and back-propagation. As will be shown, there were many parameters which could be manipulated in order to get different results. The various outputs that were desired were the periods (frequencies) d , phases θ , and the amplitudes e of laser beam exposures. The inputs were the 1-D photoresist mask patterns wanted.

Various decisions were made while getting the results, like limiting the number of exposures to three, deciding the sampling frequency of the input patterns, deciding a good amplitude range and period range over which to train. Decisions also had to be made on the size of the neural networks. The networks were limited to giving a maximum of two outputs given an input mask pattern. What the output parameters would be was decided when generating the mask patterns. For example, if d_1 and θ_1 were the parameters varied, while the other values were kept constant, they would be the outputs the network would be trained for. The network could then be tested on patterns from a set of parameters it had not seen. The networks ranged from 1-output networks, only sampling a line in parameter space, to 2-output networks where a whole 2-D parameter space was sampled.

In the back-propagation network structure, the size and number of hidden layers had to be decided. The network structure chosen was for a 2-layered back-propagation network with the number of hidden nodes (hn) depending on the number of input points.

The above statements will be explained in detail as we go through the process and explain the results. The details of how we went about achieving the goals of the thesis will be explained using the back-propagation results. After that the results of abductive modeling will be shown, with comparisons to back-propagation.

4.1 Back-Propagation

In order to implement back-propagation, various methods can be used with different learning rules and optimization techniques. First, simple back-propagation using the generalized delta rule was implemented on the problem. It gave good results but had a long run time. It was then modified using the momentum term. This modification did improve the performance but it was still slow. Finally, back-propagation using the Levenberg-Marquardt algorithm was implemented and the improvement in convergence speed was substantial. It was then decided on as the algorithm to be used for this problem. The desired sum squared error (SSE) in all the experiments was $SSE = 0.01$.

The number of input points describing the patterns was decided upon by looking at the results of experiments with varying values of input points. The number of hidden nodes to use was also decided in a similar manner. Usually, the number of hidden nodes was chosen as 10 for a 15 input network, while the number of hidden nodes was 20 for a 30 input network. The number of patterns used for the training was also decided by looking at the number of weights and various results. Table 4.1 shows some of the different experiment sets with all the values used in setting up the different patterns.

First, the focus of the experiments was to get double exposure patterns learned by the neural network so that it could recognize the one varying parameter among all the parameters used to generate the patterns. The various parameters to adjust were $d1$, $d2$, $e1$, $e2$, $\theta1$, and $\theta2$. Using a batch of training patterns, the network was trained so that if it saw patterns generated with the same varying parameter but different values, it would recognize the parameter. Fig. 4.1 shows the plots developed using the values shown in Table 4.1, where $d1$ was varied. Fig. 4.1(a) shows the result when the training data was run on the developed network. There is no visible difference in the plots. Fig. 4.1(b) shows a plot with test data of different values in the same range. It can be seen that the network does not work well in the range 0.4 to ± 0.5 , while it works well otherwise. The

Table 4.1 Values used in generating patterns for training and testing the Neural Networks

Figure	d1	d2	d3	e1	e2	e3	$\theta 1$	$\theta 2$	$\theta 3$	# runs	comments
4.1-train	0.36 (+0.05)	2	N/A	65	65	N/A	0	$\pi/2$	N/A	7	
-test	0.36 (+0.01)	2		65	65		0	$\pi/2$			
4.2-train	0.38 (+0.015 & .02)	2	0.36	65	65	65	0	$\pi/2$	0	11	# inputs-15 # hn nodes-10
-test	0.4 (+0.02)	2	0.36	65	65	65	0	$\pi/2$	0		
4.3-train	2 (+0.005)	2	0.36	65	65	65	0	$\pi/2$	0	8	
-test	2 (+0.01)	2	0.36	65	65	65	0	$\pi/2$	0		
4.4- train	0.4 (+0.015)	2	0.37	65	65	65	0	$\pi/2$	0	error= 1.0268	Did not converge
4.5-train	0.5	2	0.5	25 (+0.5)	25	25	0	$\pi/2$	0	300	test in Fig. 6 hn=10,inp=15
4.6-test	0.5	2	0.5	26.3 (+0.5)	25	25	0	$\pi/2$	0		
4.7-train	0.5	2	0.5	25	25	25	0-2 π	$\pi/2$	0	76	test in Fig. 8 hn=10,inp=15
4.8-test	0.5	2	0.5	25	25	25	0.1-2 π	$\pi/2$	0

(a) Plot of test data = train data

(b) Plot of test data $\neq$ train data

Fig. 4.1 Train and test data for double exposure pattern with one output network.

number of training patterns in the whole range was 36, while there were 175 test samples and so it was possible that the problem in 0.4-0.5 range was due to very complex patterns in the region or due to under sampling of the region when training. This network ran in 7 iterations, which was pretty fast.

As the final goal was to implement patterns with three exposures, the networks were trained to triple exposure patterns. Table 4.1 also describes the values for this set of experiments. Fig. 4.2 shows plots for the values where the number of training samples was increased from 36 to 100. This gave a finer sampling of the range. The trained network was able to give about the exact values on the training data (Fig. 4.2(a)), where the graph had different sampling intervals (see Table 4.1). It also performed well on the test data of 60 points (Fig. 4.2(b)), as the errors were smaller in the 0.4-0.5 range. This showed that this area needed finer sampling than other areas and so it was decided that it would be dropped in future patterns. The network was also trained on values in the 2-3 range for d1 and was seen to work very well (Fig. 4.3(a), (b)). At that time, attempts were made to see if the network could extrapolate as well as it could interpolate, and the test results were pretty bad as the network had never seen those kinds of patterns before. This showed a fundamental property of the networks which is that they cannot extrapolate well but can interpolate well.

The next step after seeing the ability of the network to learn well with one output was to use two outputs. However, one output value was kept constant while the other varied. The network seemed to have a hard time learning the patterns here, and the error could not go down to 0.01. Fig. 4.4 illustrates this. In Fig. 4.4(a), the network seems to have learned the values well except in the 30-40 and 70-80 range. The data was then studied to see what the problem could be. It was realized that there were not enough values of the pattern being seen, especially in that region, and the patterns were also not

(a) Plot of test data = train data

(b) Plot of test data $\neq$ train data

Fig. 4.2 Train and test data with output d1 in network in 0.38-2 range.

(a) Plot of test data = train data

(b) Plot of test data $\neq$ train data

Fig. 4.3 Train and test data with output d1 in network in 2-3 range.

(a) Plot of test data = train data - output d1 for node 1

(b) Plot of test data = train data - output d3 for node 2

Fig. 4.4 Train data for network with two outputs, where the outputs are periods and one output is constant.

having many variations, and hence the network was not able to train well (refer to chapter II for details).

The problem was solved by changing the exposure amplitude strength used in pattern generation. Fig. 4.5 shows the resulting training plots when the network was trained for two amplitudes e_1 and e_3 , where Fig. 4.5(a) is for e_1 and Fig. 4.5(b) is for e_3 . Fig. 4.6 shows the plots for the testing data. It can be seen that the results were fairly accurate for the training data and the testing data (note fine vertical scales for 4.5(b) and 4.6(b)), although it took 300 iterations to train the network. Figs. 4.7(a,b) and 4.8(a,b) show similar results where the phase θ_1 was varied and θ_3 was kept constant (Note the fine scales on 4.7(b) and 4.8(b)). Here too, the results were fairly accurate. Various other cases were attempted, including one where both output parameters varied almost equally. These cases also gave good results.

Finally, the whole sample space for any two varying parameters was tested. An example sample space of two variables d_1 and d_3 is shown in Fig. 4.9. In this example, a ten by ten sampling of the d_1 - d_3 space is being taken. Previously, only one line of samples was taken, but now the whole space was being sampled and so the resulting network would have to predict any combination in this space.

Using the above mentioned procedure, cases were trained for and the results were very strange. Fig. 4.10 shows the results of training data on the d_1 - d_3 sampled space. The equal length lines in 4.10(a) are for the desired curve and the staircase line is the actual result. In Fig. 4.10(b), the desired curve is a straight climbing line, while the actual result is similar to that of Fig. 4.10(a)'s result. The data was studied once again to see what the problem could be, and it was realized that since θ_1 and θ_3 were equal, there was some symmetry created in the patterns. The network was being told that two very similar patterns had two very different parameters. This was causing the network to average out the values. The phase for θ_3 was changed to $\pi/3$ to remove the symmetry.

(a) Plot of test data = train data -output e1 for node 1

(b) Plot of test data = train data -output e3 for node 2

Fig. 4.5 Train data for network with two outputs, where the outputs are amplitudes and one output is constant.

(a) Plot of test data $\neq$ train data -output e1 for node 1

(b) Plot of test data $\neq$ train data -output e3 for node 2

Fig. 4.6 Test data for network with two outputs, where the outputs are amplitudes and one output is constant.

(a) Plot of test data = train data - output θ_1 for node 1

(b) Plot of test data = train data - output θ_3 for node 2

Fig. 4.7 Train data for network with two outputs, where the outputs are phases and one output is constant.

(a) Plot of test data $\neq$ train data -output θ_1 for node 1

(b) Plot of test data $\neq$ train data -output θ_3 for node 2

Fig. 4.8 Test data for network with two outputs, where the outputs are phases and one output is constant.

Fig. 4.9 Plot of 2 parameters' sample space

Table 4.2 shows the various parameters that were varied and used for training and testing with the changed parameter. Here, the number of samples were also increased to 15X15 samples. Fig. 4.11 shows the error plot for case 1 with the d1-d3 sample space. As seen, the network trained in only 12 iterations. Fig. 4.12(a,b) shows the results of the training data, while Fig. 4.13(a,b) shows the results of running some test data. Again, the results were much better around the 1-2 range rather than below 1.

For case 2, the training data results were similar to case 1. The results of two different tests are also shown in Fig. 4.14 and Fig. 4.15. Fig. 4.16 shows the results of running some test data on the network trained in the d3-e1 space (case 3). The training results showed no visible difference from the actual values, and so are not shown. Fig. 4.17 shows the test result for the θ_1 -e1 space(case 4). Once again, the training results were excellent for this case. Fig. 4.18 shows the results obtained from testing the amplitude space e1-e2 (case 5). Note the fine vertical scale on Fig. 4.18(a). The training results here were similar to case 4. Finally, Fig. 4.19 and Fig. 4.20 show the training and test results obtained from learning the phase space θ_1 - θ_2 .

(a) Plot of test data = train data -output for node 1

(b) Plot of test data = train data -output for node 2

Fig. 4.10 Training data for network with two varying outputs

Table 4.2 Cases trained for in 2-D sample space

Case #	d1	d2	d3	e1	e2	e3	$\theta 1$	$\theta 2$	$\theta 3$	# runs
1-train	0.52-1.5 (+0.07)	2	0.52-1.5 (+0.07)	25	25	25	0	$\pi/2$	$\pi/3$	12
test	0.5 (+0.005)	2	1.501 (-0.0096)	25	25	25	0	$\pi/2$	$\pi/3$	
2-train	0.5	2	0.52-1.5 (+0.07)	25	25	25	0-1.8 π	$\pi/2$	$\pi/3$	43
test1	0.5	2	1.501 (-0.0096)	25	25	25	0(+0.05)	$\pi/2$	$\pi/3$	
test2	0.5	2	0.501 (+0.0096)	25	25	25	0.5	$\pi/2$	$\pi/3$	
3-train	0.5	2	0.52-1.5 (+0.07)	25-32 (+0.5)	25	25	0	$\pi/2$	$\pi/3$	14
test	0.5	2	0.501 (+0.0086)	25.8 (+0.05)	25	25	0	$\pi/2$	$\pi/3$	
4-train	0.5	2	0.5	25-32 (+0.5)	25	25	0-1.8 π	$\pi/2$	$\pi/3$	84
test	0.5	2	0.5	29.7(-0.034)	25	25	0(+0.055)	$\pi/2$	$\pi/3$	
5-train	0.5	2	0.5	25-32 (+0.5)	25-32 (+0.5)	25	0	$\pi/2$	$\pi/3$	15
test	0.5	2	0.5	30	25.3(+0.063)	25	0	$\pi/2$	$\pi/3$	
6-train	0.5	2	0.5	25	25	25	0-1.8 π	0-1.8 π	$\pi/3$	67
test	0.5	2	0.5	25	25	25	0(+0.05)	1.8 π (-0.043)	$\pi/3$

Fig. 4.11 Error plot for training network with two outputs, where the outputs are periods

(a) Plot of test data = train data - output d1 for node 1

(b) Plot of test data = train data - output d3 for node 2

Fig. 4.12 Train data for network with two outputs, where the outputs are periods (case 1).

(a) Plot of test data $\neq$ train data -output d1 for node 1

(b) Plot of test data $\neq$ train data -output d3 for node 2

Fig. 4.13 Test data for network with two outputs, where the outputs are periods (case 1).

(a) Plot of test data $\neq$ train data -output θ_1 for node 1

(b) Plot of test data $\neq$ train data -output d_3 for node 2

Fig. 4.14 Test data for network with two outputs (case 2).

(a) Plot of test data $\neq$ train data -output θ_1 for node 1

(b) Plot of test data $\neq$ train data -output d_3 for node 2

Fig. 4.15 Test data for network (case 2) with one output constant.

(a) Plot of test data $\neq$ train data -output e_1 for node 1

(b) Plot of test data $\neq$ train data -output d_3 for node 2

Fig. 4.16 Test data for network with outputs e_1 and d_3 (case 3).

(a) Plot of test data $\neq$ train data -output e_1 for node 1

(b) Plot of test data $\neq$ train data -output θ_1 for node 2

Fig. 4.17 Test data for network with outputs e_1 and θ_1 (case 4).

(a) Plot of test data $\neq$ train data -output e1 for node 1

(b) Plot of test data $\neq$ train data -output e2 for node 2

Fig. 4.18 Test data for network with outputs e1 and e2 (case 5).

(a) Plot of test data = train data - output θ_1 for node 1

(b) Plot of test data = train data - output θ_2 for node 2

Fig. 4.19 Train data for network with outputs θ_1 and θ_2 (case 6).

(a) Plot of test data $\neq$ train data -output θ_1 for node 1

(b) Plot of test data $\neq$ train data -output θ_2 for node 2

Fig. 4.20 Test data for network with outputs θ_1 and θ_2 (case 6).

It can be concluded from looking at these results that back-propagation does work well in training these patterns but the interpolation results are not always good across the entire parameter range. Next, the results obtained from abductive modeling using AIM will be presented.

4.2 Abductive Modeling

The results for abductive modeling were obtained using AIM to train the networks and encode the results in "C" functions. The networks were then tested by using the functions in a "C" program and running different patterns through it. Appendix C shows a sample program used to test the network.

AIM does not use the sum squared error (SSE) to decide how low the training error can be, as it uses the complexity penalty factor (KP) to decide how complex a network can be. The only way to get a comparison between the two methods is by running the statistical evaluation function that AIM has. Table E.1 in Appendix E shows the statistics that were found for one simulation. As seen in the table, AIM generates the average squared error, which is the SSE divided by the number of patterns. Using this variable, a comparison of the errors found using Matlab and AIM when training the networks can be made. Note that it is harder to make an exact comparison when there are two outputs, as AIM will generate two different sets of statistics for the two outputs, while Matlab has only one.

As was done with back-propagation, first double exposure patterns were used to train a one output network. Fig. 4.21 illustrates the results of the same case as Fig. 4.1, where period d1 was varied. It can be seen that although the error is larger than back-propagation (0.2 vs 0.01) for training, it does give a better result for the test. The resulting network is shown in Fig. 4.22. It can be seen that the network did not use all 15

(a) Plot of test data = train data

(b) Plot of test data $\neq$ train data

Fig. 4.21 Train and test data for double exposure pattern with one output network

Fig. 4.22 Network for Fig. 4.21

of the input pattern values. This is a distinct difference with back-propagation (Matlab) which uses all the values. Another difference to note is that all the networks developed here will be separate, where each network is for one output. This is also unlike back-propagation which uses one network for all outputs.

Fig. 4.23 shows the resulting networks after running AIM on the same data as Matlab was for Fig. 4.5. Here, the patterns were triple exposures and $KP = 1$. From the networks, it can be seen that AIM needed only one pattern value to map the input to the output for the varying amplitude parameter. The constant valued output(node 2) was just given that value and none of the inputs were used. Figs. 4.24 and 4.25 show the results from training and testing both networks for both set of outputs.

Fig. 4.23 Network for Fig. 4.24

Fig. 4.26 shows the resulting networks after running AIM on the same data as Matlab for Fig. 4.7. Here, the patterns were also triple exposures, where the parameters being tested for were the phases and one phase was kept constant. As with the above example with the amplitudes, the constant phase output had a network with only the phase

(a) Plot of test data = train data -output e_1 for node 1

(b) Plot of test data = train data -output e_3 for node 2

Fig. 4.24 Train data for network with two outputs, where the outputs are amplitudes and one output is constant

(a) Plot of test data $\neq$ train data -output e1 for node 1

(b) Plot of test data $\neq$ train data -output e3 for node 2

Fig. 4.25 Test data for network with two outputs, where the outputs are amplitudes and one output is constant

Fig. 4.26 Network for Fig. 4.27

value, and no inputs. Therefore, only the varying phase(θ_1) result is shown in Figs. 4.27 and 4.28, where θ_1 is from 0 - 2π .

As seen in Figs. 4.27 and 4.28, the value for the phase around 0 and 2π has averaged out around 3 radians. This will happen due to the fact that the two different values will generate the same pattern. It is interesting to note that Matlab did not show the same error in its results for this case. It could just mean that Matlab is not as sensitive to this error as AIM is.

Fig. 4.27 Plot of test data = train data -output θ_1 for node 1 where, node 2 output is constant valued θ_3

Fig. 4.28 Plot of test data $\neq$ train data -output θ_1 for node 1 where, node 2 output is constant valued θ_3

The next set of results will be for the 2-D sample space, where the same cases were trained and tested for as for the back-propagation case. Therefore, the same case numbers will be referred to. All the AIM results in this set took a shorter period of time to run as compared to back-propagation with Matlab. Also note that AIM did not use all the input values to learn the mappings for these networks.

Fig. 4.29 shows the two networks that were created for case 1, where two periods were varied.

Fig. 4.29 Network for case 1

The complexity penalty factor used for case 1 and the rest of the cases was $KP = 0.6$. It was decided upon as it was almost halfway in between the simple network with $KP = 1$ and the most complex network with $KP = 0.1$. The run time for training the networks also increased when KP was decreased.

Fig. 4.30 shows the training results. It can be seen that the training error for AIM was higher than for Matlab. Since AIM has two networks for the two outputs, there were two sets of errors. For node 1, the $SSE = 0.099$ while for node 2, the $SSE = 0.45$. Fig. 4.31 shows the results obtained when testing the network as shown in Table 4.2 for case 1. It can be seen that the errors are less than those obtained from Matlab. Fig. 4.32 shows the networks obtained for case 2, where T stands for Triple. The networks obtained from AIM are a lot more complex for this case compared to case 1.

Figs. 4.33, 4.34 and 4.35 show the training and testing results for case 2 in Table 4.2. The errors obtained for the network were as follows: SSE for node 1=7.65, while for node 2=0.2. Note that although the error is pretty large for node 1, it is not illustrated very well in the figures. This was one problem encountered in the experiments as the training figures could not illustrate the errors very well.

The results and networks for case 3 will now be shown. Fig. 4.36 shows the networks generated by AIM. These networks are relatively simple, illustrating the ease AIM had in learning an amplitude-period pattern. The training results are not shown as they were not visibly different from the desired results. The errors generated here were: for node 1- $SSE = 0.2$, for node 2- $SSE = 0.017$.

(a) Plot of test data = train data - output d1 for node 1

(b) Plot of test data = train data - output d3 for node 2

Fig. 4.30 Train data for network with two outputs, where the outputs are periods (case 1)

(a) Plot of test data $\neq$ train data -output d_1 for node 1

(b) Plot of test data $\neq$ train data -output d_3 for node 2

Fig. 4.31 Test data for network with two outputs, where the outputs are periods (case 1)

Fig. 4.32 Network obtained for case 2.

(a) Plot of test data = train data -output phi1 for node 1

(b) Plot of test data = train data -output d3 for node 2

Fig. 4.33 Train data for network with two outputs (case 2)

(a) Plot of test data $\neq$ train data -output phi1 for node 1

(b) Plot of test data $\neq$ train data -output d3 for node 2

Fig. 4.34 Test data for network (case 2)

(a) Plot of test data $\neq$ train data -output phi1 for node 1

(b) Plot of test data $\neq$ train data -output d3 for node 2

Fig. 4.35 Test data for networks (case 2) with one output constant

Fig. 4.36 Networks for case 3

The test results are shown in Fig. 4.37. These results are much better than those obtained from Matlab.

Fig. 4.38 shows the network generated for case 4 which is the e1-phil case. Here the network was more complex, probably due to the phase variation. T stands for the Triple function and D stands for the Double function. The training results for this case are shown in Fig. 4.39, while some test results are shown in Fig. 4.40. The errors for this case are: SSE for node 1 = 48.37, SSE for node 2 = 2.25. The high error in node 1 is reflected in the test results.

The training results for case 5 with AIM were excellent, with no visible difference between the training results and the desired error. The errors for the networks were as follows. SSE for node 1 = 0.0014, SSE for node 2 = 0.24. The test results from this case are shown in Fig. 4.41 (note the fine scale). The similarity of the test results can be compared with the Matlab result for this case. Here, Matlab seems to have performed better. Fig. 4.42 shows the networks generated by AIM for case 5. The amplitude-amplitude networks were very simple.

(a) Plot of test data $\neq$ train data -output e1 for node 1

(b) Plot of test data $\neq$ train data -output d3 for node 2

Fig. 4.37 Test data for networks (case 3) with amplitude-period pattern

Fig. 4.38 Network for case 4

(a) Plot of test data = train data -output e1 for node 1

(a) Plot of test data = train data -output phi1 for node 2

Fig. 4.39 Train data for networks in case 4.

(a) Plot of test data $\neq$ train data -output e1 for node 1

(b) Plot of test data $\neq$ train data -output phi1 for node 2

Fig. 4.40 Test data for Networks with case 4.

(a) Plot of test data $\neq$ train data -output e1 for node 1

(a) Plot of test data $\neq$ train data -output e1 for node 1

Fig. 4.41 Test data for networks for case 5.

Fig. 4.42 Networks for case 5

The final set of results to be shown is for case 6, which was the phase-phase network. This was a difficult case for the networks to learn as shown by the complex networks generated. Fig. 4.43 shows these networks. The training results can be seen in Fig. 4.44, while the test results can be seen in Fig. 4.45.

Fig. 4.43 Networks for case 6

(a) Plot of test data = train data - output ϕ_1 for node 1

(b) Plot of test data = train data - output ϕ_2 for node 2

Fig. 4.44 Train data for Networks for case 6.

(a) Plot of test data $\neq$ train data -output ϕ_1 for node 1

(b) Plot of test data $\neq$ train data -output ϕ_2 for node 2

Fig. 4.45 Test data for networks for case 6

The errors for these networks were as follows. For node 1 $SSE = 17.1$; for node 2 $SSE = 13.5$. The test results for the networks showed better performance for θ_1 in the AIM result and better results for θ_2 in Matlab.

In order to see how badly the networks were actually performing when the parameters were off the actual parameter values, both the desired input patterns and the patterns obtained using the network parameters were plotted for case 1, using the test results in Fig. 4.46. The patterns were plotted in the 25-36 pattern number range. Fig. 4.47 shows the resulting patterns, where the dashed lines represent the desired result while the solid line represents the actual result. It can be seen that the results are pretty close in most of the cases, as this set of results is for a very bad test result.

All the results shown, both for back-propagation and abductive modeling have shown the ability of the neural networks to learn the nonlinear inverse mappings from the desired 1-D photoresist profiles to the laser beam exposure parameters. There were differences in performance according to the abilities of each method. The concluding chapter will talk about all of this in more detail.

(a) Plot of test data $\neq$ train data for case 1 -output for node 1

(b) Plot of test data $\neq$ train data for case 1 -output for node 2

Fig. 4.46 Plot of test results for showing how the test results varied from the actual results

(a) Pattern # 25

(c) Pattern # 30

(d) Pattern # 35

CHAPTER V

CONCLUSION

The aim of this thesis was to implement neural networks for learning the nonlinear inverse mapping between input wafer photoresist patterns and the laser beam exposure parameter outputs that are used to generate them. In order to achieve this, two different types of neural networks, back-propagation and abductive modeling, were studied and applied to the problem. The lithography model being used for the problem was also studied in order to understand the characteristics of the data. Both types of networks showed the ability to learn the mapping, as both performed well with different tests after training on a 2-D sample space of 2 varying input parameters. In conclusion, the following points can be made about the two types of neural networks.

Although abductive modeling seems to be more powerful since it gave better test results, even with poorer training results, i.e., it interpolates better, it does use two distinct parallel networks to develop two output nodes. However, it does not require much storage space to run an application once it is developed as it uses a few polynomial equations to map the inputs to the outputs. The network also did not use all of the input variables in the results shown. This might change if the network is made more complex by changing the complexity penalty factor. More input points may be used then and the training error might also go down.

The training set results obtained from back propagation are quite accurate although the interpolation (test) results are not as good, and the main advantage is that two networks are not needed to get two output nodes. One drawback is that more storage space is needed to store all the weights when used as an application. Also, the training time for back-propagation is much longer in comparison to abductive modeling.

The back-propagation network can be improved by adding more training patterns and by increasing the number of hidden nodes.

Further work is still needed in order to develop highly complex 2-D patterns and in order to improve the networks used, but this work does show the potential capability of this kind of multiparameter inverse application.

REFERENCES

- [1] Zaidi, Saleem H., & Brueck, S.R. "Multiple Exposure Interferometric Lithography." J. Vac. Sci. Technology, May/June 1993.
- [2] Zieger, David H., & Mack, Chris A. "Generalized Approach Toward Modeling Resist Performance." AIChE Journal, Dec 1991.
- [3] Wasserman, Philip D. Neural Computing: Theory and Practice. Van Nostrand Reinhold, New York. 1989.
- [4] Zurada, Jacek M. Introduction to Artificial Neural Systems. West Publishing Company, Minneapolis, MN. 1992.
- [5] Neural Network Toolbox User's Guide. Mathworks, Inc. , Natick, Mass. 1994.
- [6] Press, W.H., & Teukolsky, S.A., Vetterling, W.T. , Flannery, B.P. Numerical Recipes in C: The art of scientific computing. Cambridge University Press, New York,. 2nd Edition.
- [7] Bazaraa, M.S., Sherali, H.D., & Shetty, C.M. Nonlinear Programming: Theory and Algorithms. John Wiley & Sons, Inc, New York. 1979.
- [8] Farlow, S. J. , ed. Self-Organizing Methods in Modeling: GMDH Type Algorithms. Marcel-Dekker, Inc., New York, 1984.
- [9] AIM, Abduce Superior Models With Ease, User's Manual AIM for Windows Version 2.01, AbTech Corporation, Charlottesville, VA, 1994.

APPENDIX A
PROGRAM FOR GENERATING PATTERNS

```

/*****
/* Manisha Mahendra          */
/* This program generates the output given */
/* a certain value for d1,d2 and d3      */
*****/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PI (3.141592654)
main()
{
double x,z,n,k,ep,t,eo,e1,e2,e3,d1,d2,atf,ct,d3,ph1,ph2,ph3;

FILE *out,*out1,*out2;
int i,j;

if ((out=fopen("test3","w"))==NULL)
{
printf("cant open \"test3\" file.\n");
exit(1);
}
if ((out1=fopen("test3d","w"))==NULL)
{
printf("cant open \"test3d\" file. \n");
exit(2);
}
if ((out2=fopen("test3d1","w"))==NULL)
{
printf("cant open \"test3d1\" file. \n");
exit(3);
}

d1=0.5;
d2=2.0;
d3=0.5;
ph1=0;
ph2=PI/2.0;
ph3=0;

```

```

e1=25;
e2=25;
e3=25;
for (j=0;j<100;j++)
{
    d1=d1+0.01;
    d3=d3+0.01;

fprintf(out1,"%f\n",d1);
fprintf(out2,"%f\n",d3);

n=5.0;
k=0.015;
eo=89.0;
ct=0;
for (i=0;i<30;i++)
{

ep=e1*(1+cos(2.0*(i/10.0)*PI/d1+ph1))+e2*(1+cos(2.0*(i/10.0)*PI/d2+ph2))+e3*(1+cos(2.0*(i/10.0)*PI/d3+ph3));
x=((1.0-exp(-k*ep))/(1.0-exp(-k*eo)));
z=pow(x,n);
/* printf("x=%f, n=%f, z=%f\n",x,n,z);*/
t=1.0-z;
if (t>0.0)
    atf=t;
else
    atf=0.0;
    ct=ct+atf;
    fprintf(out,"%f\n",atf);
}
if (ct==0)
{
    fprintf(out,"the above array has zeroes \n");
    fprintf(out1,"the above number produced zeroes \n");
    fprintf(out2,"the above number produced zeroes \n");
}
}
}
•

```

SAMPLE MATLAB PROGRAM

[illegible]


```

[wm1,bm1,wm2,bm2] =initff(ind,20,'logsig',2,'purelin');

echo on
clc
% TRAINING THE NETWORK
% =====

df = 25; % Frequency of progress displays (in epochs).
me = 30000; % Maximum number of epochs to train.
eg = 0.01; % Sum-squared error goal.

tp=[df me eg];
% Training begins...please wait...

[wm1,bm1,wm2,bm2,tme,tmr] =trainlm(wm1,bm1,'logsig',wm2,bm2,'purelin',po,to,tp);

% ...and finishes.

%

[x1,x2]=simuff(po,wm1,bm1,'logsig',wm2,bm2,'purelin');

x=1:1:100;
figure(2)
plot(x,x2(1,:),x,to(1,:))
title('plot of trdat1=test30 with ph1=0-2pi,ph2=pi/2,ph3=0,e=0.01')
pause;

figure(3)
plot(x,x2(2,:),x,to(2,:))
title('plot of trdat2=test30 with ph1=0-2pi,ph2=pi/2,ph3=0,e=0.01')

echo off
disp('End')

```

APPENDIX C

SAMPLE AIM PROGRAM

```
#include <stdio.h>
#include <stdlib.h>

#define pow1(x) (x)
#define pow2(x) ((x)*(x))
#define pow3(x) ((x)*(x)*(x))
#define LIMIT(v,mn,mx) ((v>(mx))?(mx):((v<(mn))?(mn):v))
#define SIZE 30

double sp31(double inarr[]);
double sp32(double inarr[]);
int main(void)
{
    FILE *fp1,*fp2,*fp3;
    int i,j;
    double arr[SIZE];
    double var;

    if ((fp1=fopen("test32","r"))==NULL)
    {
        printf("cant open \"test32\" file.\n");
        exit(1);
    }

    if ((fp2 = fopen("answ2","w"))==NULL)
    {
        printf("cant open \"answ2\" file.\n");
        exit(2);
    }
    if ((fp3 = fopen("answ12","w"))==NULL)
    {
        printf("cant open \"answ12\" file.\n");
        exit(3);
    }

    for (i=0;i<110;i++)
    {
        for (j=0;j<SIZE;j++)
```

```

    {
        fscanf(fp1,"%lf\n",&var);
        arr[j]=var;
    }
    fprintf(fp2,"%lf\n",sp31(arr));
    fprintf(fp3,"%lf\n",sp32(arr));
}
return 0;

}
/*****

```

ABDUCTIVE NETWORK SUBROUTINE: sp31()

Generated by AbTech's AIM 2.0 for Windows

Format: Standard

INPUTS: An array of 34 double(s):

```

    inarr[0] = UnusedInput1
    inarr[1] = UnusedInput2
    inarr[2] = UnusedInput3
    inarr[3] = UnusedInput4
    inarr[4] = UnusedInput5
    inarr[5] = a2
    inarr[6] = a3
    inarr[7] = UnusedInput8
    inarr[8] = UnusedInput9
    inarr[9] = a6
    inarr[10] = UnusedInput11
    inarr[11] = UnusedInput12
    inarr[12] = a9
    inarr[13] = a10
    inarr[14] = UnusedInput15
    inarr[15] = UnusedInput16
    inarr[16] = a13
    inarr[17] = UnusedInput18
    inarr[18] = UnusedInput19
    inarr[19] = UnusedInput20
    inarr[20] = UnusedInput21
    inarr[21] = UnusedInput22
    inarr[22] = UnusedInput23
    inarr[23] = UnusedInput24
    inarr[24] = UnusedInput25
    inarr[25] = UnusedInput26
    inarr[26] = UnusedInput27
    inarr[27] = UnusedInput28

```

```

inarr[28] = UnusedInput29
inarr[29] = UnusedInput30
inarr[30] = UnusedInput31
inarr[31] = UnusedInput32
inarr[32] = UnusedInput33
inarr[33] = UnusedInput34

```

OUTPUTS: A returned double representing out

STATISTICS:

```

a2      : Mean = 0.546367, Sigma = 0.332437
          Min =    0, Max  = 0.982006

a3      : Mean = 0.8001, Sigma = 0.198802
          Min = 0.297449, Max  = 0.999667

a6      : Mean = 0.768952, Sigma = 0.292333
          Min =    0, Max  =    1

a9      : Mean = 0.497111, Sigma = 0.34786
          Min =    0, Max  = 0.99969

a10     : Mean = 0.382506, Sigma = 0.342989
          Min =    0, Max  = 0.996786

a13     : Mean = 0.182977, Sigma = 0.286936
          Min =    0, Max  = 0.910577

out     : Mean = 2.82743, Sigma = 1.74902
          : Min =    0, Max  = 5.65487
          : KP  = 73.0%, FSE  = 27.0%
          : Predicted Error = 0.366249

```

*****/

```

double sp31(double inarr[])
{
double    a2 = inarr[1]; /* rename input for clarity */
double    a3 = inarr[2]; /* rename input for clarity */
double    a6 = inarr[5]; /* rename input for clarity */
double    a9 = inarr[8]; /* rename input for clarity */
double    a10 = inarr[9]; /* rename input for clarity */
double    a13 = inarr[12]; /* rename input for clarity */
double out ; /* output value to return */

```

```

double node6 ; /* working variable */
double node7 ; /* working variable */
double node10 ; /* working variable */
double node38 ; /* working variable */
double node37 ; /* working variable */
double node13 ; /* working variable */
double node14 ; /* working variable */
double node36 ; /* working variable */
double node17 ; /* working variable */
double node35 ; /* working variable */

```

```

#ifdef DEBUG

```

```

    printf("sp31: received a2 = %g.\n", (double)a2);
    printf("sp31: received a3 = %g.\n", (double)a3);
    printf("sp31: received a6 = %g.\n", (double)a6);
    printf("sp31: received a9 = %g.\n", (double)a9);
    printf("sp31: received a10 = %g.\n", (double)a10);
    printf("sp31: received a13 = %g.\n", (double)a13);

```

```

#endif

```

```

/* node6 -- a2 */

```

```

node6 = -1.64352075569169 + 3.00809175130043*LIMIT(a2,0,0.982006);

```

```

/* node7 -- a3 */

```

```

node7 = -4.02461631595707 + 5.03014000662262*LIMIT(a3,0.297449,0.999667);

```

```

/* node10 -- a6 */

```

```

node10 = -2.63039854645942 + 3.42076000743575*LIMIT(a6,0,1);

```

```

/* node38 -- Triple */

```

```

node38 = 0 - 0.576564667098692 - 0.194829635923298*node6
        - 1.2202636755746*node7 - 0.778791385193452*node10
        + 0.899806268009733*pow2(node6)
        + 0.269137359306644*pow2(node7)
        + 0.141814321373881*pow2(node10)
        - 0.489989607767395*node6*node7
        + 0.107247918837327*node6*node10
        + 0.784756983102255*node7*node10
        + 0.209069792952563*node6*node7*node10
        + 0.116936681292796*pow3(node6)
        + 0.104487377466194*pow3(node7)
        + 0.0154081230699754*pow3(node10);

```

```

/* node37 -- Triple */

```

```

node37 = 0 + 0.170432579890291 + 1.40034427341134*node38
        + 0.159102116657954*node6 - 0.0433899022639199*node7
        - 0.0603513111844523*pow2(node38)
        - 0.312527759531542*pow2(node6)
        + 0.20685408056696*pow2(node7)
        - 0.586213029513549*node38*node6
        + 0.883752823313029*node38*node7
        + 0.274527667496605*node6*node7
        + 0.135909600362422*node38*node6*node7
        - 0.445255868286545*pow3(node38)
        - 0.0658896385787673*pow3(node6)
        + 0.0779129253031304*pow3(node7) ;

```

```

/* node13 -- a9 */

```

```

node13 = -1.42905503474626 + 2.874721719218*LIMIT(a9,0,0.99969);

```

```

/* node14 -- a10 */

```

```

node14 = -1.11521522652575 + 2.91554614006975*LIMIT(a10,0,0.996786);

```

```

/* node36 -- Triple */

```

```

node36 = 0 + 0.0700183053341925 + 1.06280199941093*node37
        + 0.098632419352548*node13 - 0.0651631813454581*pow2(node37)
        + 0.169727135930176*node37*node13
        - 0.100581130926253*node13*node14
        - 0.0792295857670221*pow3(node37)
        - 0.0376184954082308*pow3(node13) ;

```

```

/* node17 -- a13 */

```

```

node17 = -0.637694697966435 + 3.48510232020609*LIMIT(a13,0,0.910577);

```

```

/* node35 -- Triple */

```

```

node35 = 0 + 0.975079593755855*node36 + 0.0221560699104322*pow2(node10)
        - 0.0306748134690992*node36*node10
        - 0.0418742976404631*node36*node10*node17 ;

```

```

/* node3 -- out */

```

```

out = 0 + 2.82743333333333 + 1.74902096197454*node35 ;

```

```

/* perform output limiting on out */

```

```

out = LIMIT( out, 0, 5.654867 );

```

```

#ifdef DEBUG

```

```

    printf("sp31: returning out = %g.\n", out);

```

```
#endif
```

```
    return( out ); /* output value */  
}
```

```
/******  
ABDUCTIVE NETWORK SUBROUTINE: sp32()  
Generated by AbTech's AIM 2.0 for Windows  
Format: Standard  
*****  
*****
```

INPUTS: An array of 34 double(s):

```
inarr[0] = UnusedInput1  
inarr[1] = UnusedInput2  
inarr[2] = UnusedInput3  
inarr[3] = UnusedInput4  
inarr[4] = UnusedInput5  
inarr[5] = UnusedInput6  
inarr[6] = a3  
inarr[7] = a4  
inarr[8] = UnusedInput9  
inarr[9] = a6  
inarr[10] = UnusedInput11  
inarr[11] = a8  
inarr[12] = a9  
inarr[13] = UnusedInput14  
inarr[14] = UnusedInput15  
inarr[15] = UnusedInput16  
inarr[16] = a13  
inarr[17] = UnusedInput18  
inarr[18] = UnusedInput19  
inarr[19] = UnusedInput20  
inarr[20] = UnusedInput21  
inarr[21] = UnusedInput22  
inarr[22] = UnusedInput23  
inarr[23] = UnusedInput24  
inarr[24] = UnusedInput25  
inarr[25] = UnusedInput26  
inarr[26] = a23  
inarr[27] = UnusedInput28  
inarr[28] = a25  
inarr[29] = UnusedInput30  
inarr[30] = UnusedInput31  
inarr[31] = UnusedInput32  
inarr[32] = a29
```

inarr[33] = UnusedInput34

OUTPUTS: A returned double representing out1

STATISTICS:

a3 : Mean = 0.8001, Sigma = 0.198802
Min = 0.297449, Max = 0.999667

a4 : Mean = 0.874731, Sigma = 0.146464
Min = 0.265604, Max = 0.999992

a6 : Mean = 0.768952, Sigma = 0.292333
Min = 0, Max = 1

a8 : Mean = 0.610417, Sigma = 0.338187
Min = 0, Max = 0.999988

a9 : Mean = 0.497111, Sigma = 0.34786
Min = 0, Max = 0.99969

a13 : Mean = 0.182977, Sigma = 0.286936
Min = 0, Max = 0.910577

a23 : Mean = 0.553766, Sigma = 0.356292
Min = 0, Max = 0.999669

a25 : Mean = 0.634078, Sigma = 0.33377
Min = 0, Max = 1

a29 : Mean = 0.563348, Sigma = 0.357851
Min = 0, Max = 0.999621

out1 : Mean = 1.01, Sigma = 0.303109
: Min = 0.52, Max = 1.5
: KP = 63.8%, FSE = 36.2%
: Predicted Error = 0.0625728

*****/

double sp32(double inarr[])

```
{
double    a3 = inarr[2]; /* rename input for clarity */
double    a4 = inarr[3]; /* rename input for clarity */
double    a6 = inarr[5]; /* rename input for clarity */
double    a8 = inarr[7]; /* rename input for clarity */
double    a9 = inarr[8]; /* rename input for clarity */
double    a13 = inarr[12]; /* rename input for clarity */
double    a23 = inarr[22]; /* rename input for clarity */
```



```

double    a25 = inarr[24]; /* rename input for clarity */
double    a29 = inarr[28]; /* rename input for clarity */
double out1 ; /* output value to return */
double node7 ; /* working variable */
double node10 ; /* working variable */
double node13 ; /* working variable */
double node38 ; /* working variable */
double node8 ; /* working variable */
double node27 ; /* working variable */
double node37 ; /* working variable */
double node12 ; /* working variable */
double node29 ; /* working variable */
double node36 ; /* working variable */
double node17 ; /* working variable */
double node33 ; /* working variable */
double node35 ; /* working variable */

```

```

#ifdef DEBUG

```

```

    printf("sp32: received a3 = %g.\n", (double)a3);
    printf("sp32: received a4 = %g.\n", (double)a4);
    printf("sp32: received a6 = %g.\n", (double)a6);
    printf("sp32: received a8 = %g.\n", (double)a8);
    printf("sp32: received a9 = %g.\n", (double)a9);
    printf("sp32: received a13 = %g.\n", (double)a13);
    printf("sp32: received a23 = %g.\n", (double)a23);
    printf("sp32: received a25 = %g.\n", (double)a25);
    printf("sp32: received a29 = %g.\n", (double)a29);

```

```

#endif

```

```

    /* node7 -- a3 */

```

```

    node7 = -4.02461631595707 + 5.03014000662262*LIMIT(a3,0.297449,0.999667);

```

```

    /* node10 -- a6 */

```

```

    node10 = -2.63039854645942 + 3.42076000743575*LIMIT(a6,0,1);

```

```

    /* node13 -- a9 */

```

```

    node13 = -1.42905503474626 + 2.874721719218*LIMIT(a9,0,0.99969);

```

```

    /* node38 -- Triple */

```

```

    node38 = 0 - 1.61410789453999 + 1.84817834960559*node7
              + 3.17202804548603*node10 + 1.12070058356962*node13
              + 0.0794808518920241*pow2(node7)
              + 0.646309909631693*pow2(node10)
              + 0.0309884384395944*pow2(node13)
              - 1.76060105410842*node7*node10
              - 0.766607003288969*node7*node13
              - 0.868400916421445*node10*node13
              + 1.18800422469877*node7*node10*node13

```

```

        - 0.0450158705149009*pow3(node7)
        + 0.116239595889962*pow3(node10)
        + 0.156938180326867*pow3(node13) ;
/* node8 -- a4 */
node8 = -5.97231095356993 + 6.8275951149931*LIMIT(a4,0.265604,0.999992);

/* node27 -- a23 */
node27 = -1.55424701673167 + 2.80668633646971*LIMIT(a23,0,0.999669);
/* node37 -- Triple */
node37 = 0 - 0.120600792289027 + 1.08762154566748*node38
        + 0.130868027792014*node8 + 0.172715632835956*pow2(node8)
        - 0.313271211221688*node38*node8
        + 0.102394135332358*node8*node27
        + 0.130727654614689*node38*node8*node27
        + 0.0342202172696078*pow3(node8) ;
/* node12 -- a8 */
node12 = -1.80496721907357 + 2.95694017177404*LIMIT(a8,0,0.999988);

/* node29 -- a25 */
node29 = -1.89974753770567 + 2.99607817661504*LIMIT(a25,0,1);
/* node36 -- Triple */
node36 = 0 + 0.0619708579923122 + 1.11760124788251*node37
        + 0.0588254203672312*node12 - 0.142710046518429*pow2(node12)
        + 0.22486499931835*node37*node12
        + 0.0992402152076234*node37*node29
        - 0.0315898658800017*node12*node29
        - 0.112730427045406*pow3(node37)
        - 0.0578027135811514*pow3(node12) ;
/* node17 -- a13 */
node17 = -0.637694697966435 + 3.48510232020609*LIMIT(a13,0,0.910577);
/* node33 -- a29 */
node33 = -1.5742541251732 + 2.79445989406852*LIMIT(a29,0,0.999621);
/* node35 -- Triple */
node35 = 0 + 0.965456918400576*node36 - 0.0689011532923853*node36*node17
        + 0.0789782955572265*node36*node17*node33 ;
/* node4 -- out1 */
out1 = 0 + 1.01 + 0.303108891324553*node35 ;
/* perform output limiting on out1 */
out1 = LIMIT( out1, 0.52, 1.5 );
#ifdef DEBUG
    printf("sp32: returning out1 = %g.\n", out1);
#endif
return( out1 ); /* output value */
}

```

APPENDIX D

DETAILED GUIDE TO USING AIM

The following guide is given to enable a user to implement AIM for any problem. The procedure is summarized in a few simple steps. For further details, please consult [9].

1. Create a database. The data set to be implemented by AIM should have the following format. It should be sequential and separated either by commas, tabs, or spaces. It should also have all the variable names across the top with the observations in columns. The variable names can be anything but must start with a letter of the alphabet. The desired output of the networks should also be included in this database, along with all the inputs. (Note: AIM automatically looks for a database with the .raw extension and so it is desirable to have the same extension to make it easier to import the file.)

2. Open a new file. Go into AIM and select **New** from the **File** menu. This allows you to name a new application. The application can then be opened later by choosing **Open** in the **File** menu. A Database window is created and will be seen. This is where you can see your database files, the models, and the networks.

3. Import Data. From the **Data** menu, choose **Import**. The Import data window opens and you can select the file you wish to import. As mentioned above, AIM looks for files with the .raw extension by default. To look for other files, select *.* from the **List files of type** box. AIM returns you to the database window after selecting the data and you will see the file in the window with a .dat extension if you click on the **Data** button. The file can be opened by clicking on the **Open** button shown. (Note: Another variable is called Idx is automatically assigned by AIM, which is useful for feature extraction.)

4. Define Model. Since AIM does not know what the data means, the **Define Model** option is used to tell AIM which variables are the inputs and which variables are the outputs. Click on the **Model** button or select **Model Definitions** under the **View**

menu. If you do not already have a model definition, click on the **New button**. You then have to enter a model definition name and hit **OK**. After that, you will see a Model Definition Table with the assigned name. Go under **Table** and choose the data you wish to model. Next, click under the **Field** heading and select the asterisk (*) symbol. This will list all the variable names as well as fill up all the table names. You will see that each variable is assigned an input status by default. To change the status, click on the arrow and three choices will be seen. For output variables, the output choice can be used. You will also see that Idx was assigned the unused choice for status. The basic model has now been made and you can go to the **File** menu and **Save** it. You can also go and change some parameters (such as CPM) using **Synthesis Settings** in the **Model** menu or leave the AIM defaults. If you do make the changes, you have to save the model definition again.

5. Generate Data. Next, choose the **Generate** option in the **Data** menu. This brings up a data generation window. You can choose whether you want your data split into training and evaluation parts or only training. This function generates the data that will be used to train the networks.

6. Synthesize Network. Go under the **Model** menu and select the **Synthesize** option. A window will come up and ask for the model table to use (see 4), which must then be specified. Then, AIM proceeds to build the networks by displaying the process as it goes along. After the synthesis is over, AIM returns to the database screen. The networks can be seen by clicking on the **Net button** and selecting the appropriate file.

7. Incorporating the Result. After the network is synthesized, it can be tested by using the **Query** option in the **Model** menu. It can also have evaluation statistics run using the **Evaluate** option in the **Model** menu. The networks can be encoded into "C" functions using the **Encode** option in the **Model** menu. These "C" subroutines can be used to implement the networks for any real-world problems.

APPENDIX E

STATISTICS OF AIM PROGRAM

Table E.1 Statistics of AIM Program

Evaluation of Network 'C:\USR\MANISHA\JANFEB\
Using Data from Table 'GEN30_TRAIN'

Summary Statistics for output 'out'		
	evaluation	training
number of observations	225	225
average absolute error	0.20868	
error standard deviation	0.18074	
average squared error	0.076067	0.076067
normalized mean squared error	0.024977	
squared error standard deviation	0.20378	
maximum absolute error	1.1358	
database output minimum	0.00000	0.00000
database output maximum	5.6549	5.6549
database output mean	2.8274	2.8274
database output standard deviation	1.7451	1.7232
network output mean	2.8274	
network output standard deviation	1.7232	
R-squared	0.97502	
root of predicted squared error	0.43433	
predicted squared error	0.18864	

PERMISSION TO COPY

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Texas Tech University or Texas Tech University Health Sciences Center, I agree that the Library and my major department shall make it freely available for research purposes. Permission to copy this thesis for scholarly purposes may be granted by the Director of the Library or my major professor. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my further written permission and that any user may be liable for copyright infringement.

Agree (Permission is granted.)

Manisha Mahandaz

Student's Signature

04/11/95

Date

Disagree (Permission is not granted.)

Student's Signature

Date