Predicting core parameters in a pressurized water reactor using an artificial neural network

by

Scott E. Wendt

A Thesis Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Department: Mechanical Engineering
Major: Nuclear Engineering

Signatures have been redacted for privacy

Signatures have been redacted for privacy

Iowa State University
Ames, Iowa
1994

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Nuclear Power Reactor Core Loads

Every 12 to 24 months the reactor at a nuclear power plant is shutdown, partially dismantled and refueled. During the refueling process, approximately one third of the fuel assemblies in the core are removed and placed in a nearby storage facility for eventual disposal. The remaining fuel assemblies will remain in service in the core for an additional one or two fuel cycles until they too will be discharged. The new, fresh fuel assemblies and the old, partially spent fuel assemblies are "shuffled" and placed back into the core. This new core arrangement or pattern is not random or haphazard, but is the result of months of careful analysis by a team of nuclear engineers.

Depending on the reactor, the new core loading pattern (LP) must satisfy several design criteria based on safety and operational goals. Some of the possible criteria the designers may have to consider are: the maximization of the length of the next power cycle, the minimization of the neutron flux at the reactor vessel wall, and/or the minimization of the power peaks. Currently, the analysis of core LPs is accomplished with complex computer programs which use diffusion or transport theory, often employing Monte Carlo

methods, to calculate the various parameters which quantify the physical characteristics of the new core.

These computer codes, while accurate, often require large amounts of computing power and time. Often, while the computer is performing the calculations, the engineer must wait, pondering the next step. Without the feedback provided by the current calculation, the next step is a mystery. The development of a faster core parameter prediction system, which would give almost instant estimates of the values of thousands of new designs, could greatly speed the design of new core reloads.

Artificial Neural Networks

Artificial neural networks (ANNs) are computer programs which employ a distributed memory scheme to 'learn' such things as function mapping, pattern classification, pattern recognition, etc.. ANNs 'learn' or are trained through a process where internal memory parameters, or weights, are systematically altered until the network performs as desired. During the training phase, the ANN is presented with an input pattern and the correct answer. The answer is stored for future reference while the input pattern is 'fed' into the network. After many internal mathematical calculations, the network produces an answer. This calculated answer or output is compared to the correct answer stored in memory. If the calculated answer is not the same as the stored answer, which

is very likely early in the learning process, the internal memory parameters or weights are adjusted to produce a better answer the next time. This process is repeated many times on many input/answer pairs until the ANN learns to produce output 'close enough' to the true answers, then the training procedure is halted. The objective in training an ANN is to have it learn the underlying functionality between the input and the output by example. If this occurs there is a good probability that the ANN has learned the functionality between the input and the output and will produce an adequately accurate answer when novel input data (i.e., not part of the training set) is presented to the network.

Problem Statement

This work describes the use of ANNs to estimate or predict key physical parameters which are needed to validate a particular core LP design. The beginning of cycle (BOC) parameters which this work will try to predict are the critical boron concentration and the pin peaking ratio.

In an industrial setting, it is proposed that the engineers in charge of designing the new core LP would use the results from the previous core loading calculations to train the ANN for the current analysis. For this work, a database, developed by Studsvik of America[31], containing core LPs and the corresponding core parameters was used to train and test the ANN.

CHAPTER 2. ARTIFICIAL NEURAL NETWORKS

Introduction

The field of neural computing is new and ever expanding. Dr. L. M. Simmons, Chair and executive vice president of the Sante Fe Institute (SFI) wrote, "We are witnessing the creation of new sciences of complexity, sciences that may well occupy the center of intellectual life in the twenty-first century."[13](p. xiii). Indeed, as time progresses, scientists and engineers are finding more and more ways to put ANNs to work.

Multi-layer Perceptrons

Although there are many types of ANNs, this work will focus on the feed forward multi-layer perceptron (MLP). In the class of neural networks which include MLPs the convention is to construct a network consisting of 'neurodes' arranged in groups, or layers, with 'connections' between the various layers. The neurodes, also referred to as processing elements or just nodes, are the location where all of the calculations carried out within the network occur. In this work, the neurodes are defined to be everything within the region bounded by the dotted lines in Figure 2.1.

As stated above, the neurodes are arranged in layers with each neurode in one layer connected to each neurode in the

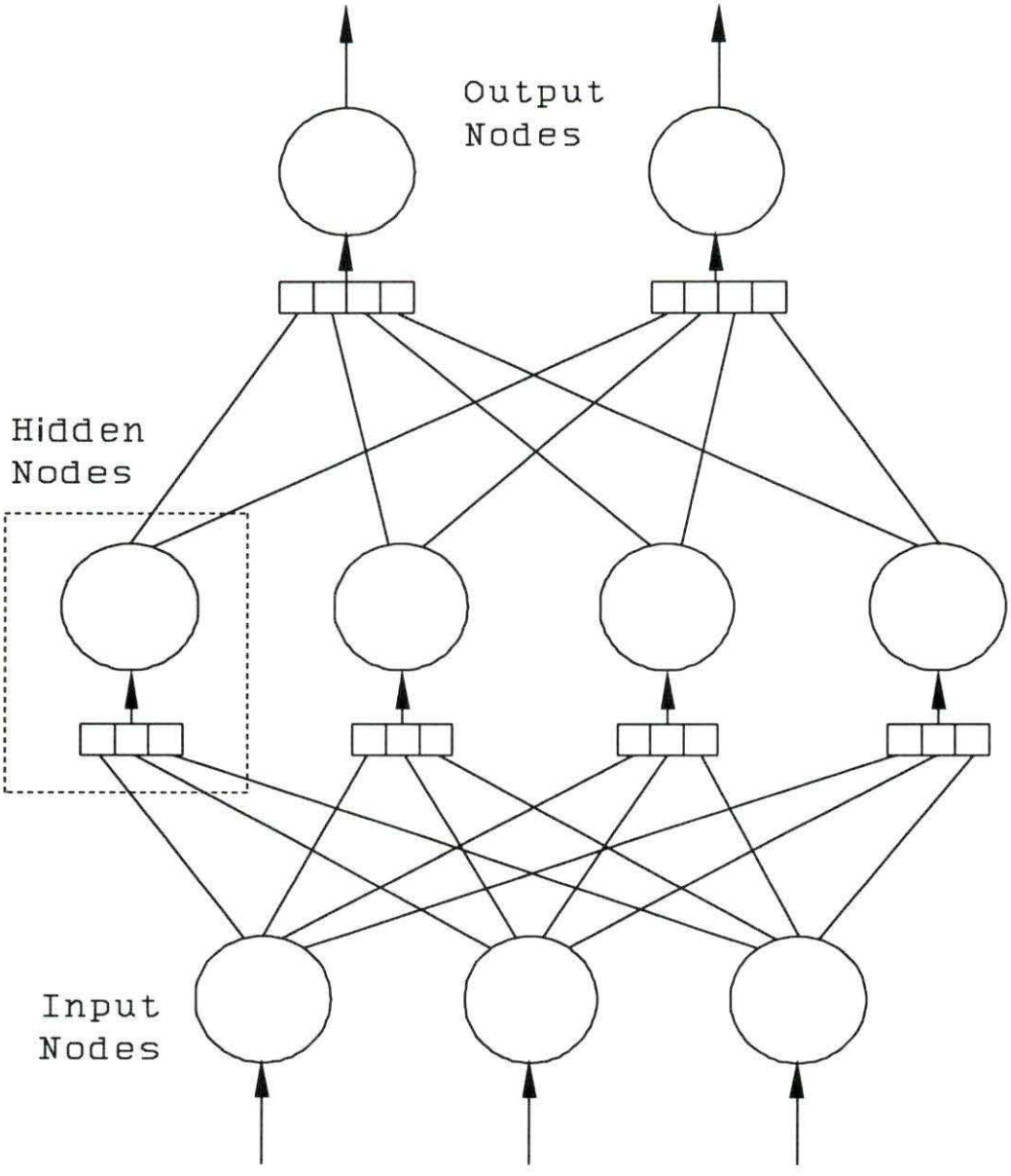


Figure 2.1 A three layer MLP with three inputs, four hidden and two output nodes.

previous layer and to each node in the successive layer. Because of this scheme of neurode connections, the MLP is said to be completely inter-connected between layers. Normally there are no connections between neurodes in the same layer.

The MLP architecture consists of multiple layers of neurodes. The layers are broken down into an input layer, one or more hidden layers and an output layer; the network shown in Figure 2.1 has a single hidden layer. In the figure, the row of solid boxes below the circle represent the adjustable learning parameters, or weights. The circle represents the neurode activation function and the solid lines between the circles and boxes are the interconnections.

During the training, or learning phase, input vectors with known output vectors are presented to the network. This process where only 'questions' with known 'answers' are put through the network is known as supervised learning. Repeated application of the various input/output vectors allows the network to learn to produce the correct output to each set of inputs by adjusting its weight values.

If enough examples are presented to the network, and these examples are representative of the data set as a whole, the network may be able to learn the underlying functionality between the input and the output. If the ANN does learn the input/output functionality correctly, it may then be able to correctly classify novel input data. This ability to learn

the correct classification for novel data is referred to as generalization. When a network fails to generalize, but instead learns to produce the correct answers only to the specific inputs it was trained on, it is said to have memorized the data.

The flow of data through a feed forward MLP can be divided into two phases. First the network receives an input, or question, from the outside world via the input nodes. The input nodes pass the data forward, or up the network structure, fanning out their data to all of the nodes in the hidden layer. No calculations are performed by the input nodes. The hidden nodes pass their outputs on to the output nodes and the output nodes display the network's output to the outside world. This stage, from input to output, is aptly called the feed-forward phase.

The second phase begins with a comparison of the calculated output and the desired output for each of the output nodes. The difference between the calculated and desired output is the network error. This error is used to alter the interconnection weight parameters so that the error will be smaller the next time. It is the systematic changing of these weights that is responsible for the ability of the ANN to 'learn'. The error is propagated backward through the network from the output nodes to the input nodes so that each weight's contribution to the error can be calculated. This

second phase is called the back propagation phase.

Feed Forward

The learning process begins in the feed forward phase, the various node inputs and outputs are depicted in Figure 2.2. An input vector is applied to the network via the input nodes. The input values are 'fanned out' by the input nodes to each node in the next layer. Again, no calculations are performed in the input layer. In the hidden layer neurodes, the inputs are multiplied by the connection weights and the resulting products are summed together, see Figure 2.3. A bias value, b_{ij}, is subtracted from the sum and the result, net_{ij}, is applied to the nodal activation function. The output of the activation function, x_{ij}, is the output of the node. The formula for calculating the nodal output is given by Equations 2.1 and 2.2.

$$x_{ij} = f(net_{ij}) \tag{2.1}$$

$$net_{ij} = \sum_{k=1}^{N_{i-1}} w_{ijk} x_{i-1,k} - b_{ij} \tag{2.2}$$

In these equations, the subscript *i* indicates the current layer, *i-1* indicates the previous or lower layer, and *i+1* indicates the next layer. The subscript *j* represents the number of the current node in layer *i*. The subscript *k* represents the number of the specific weight associated with node *j* in layer *i-1*. The node passes its output value to all

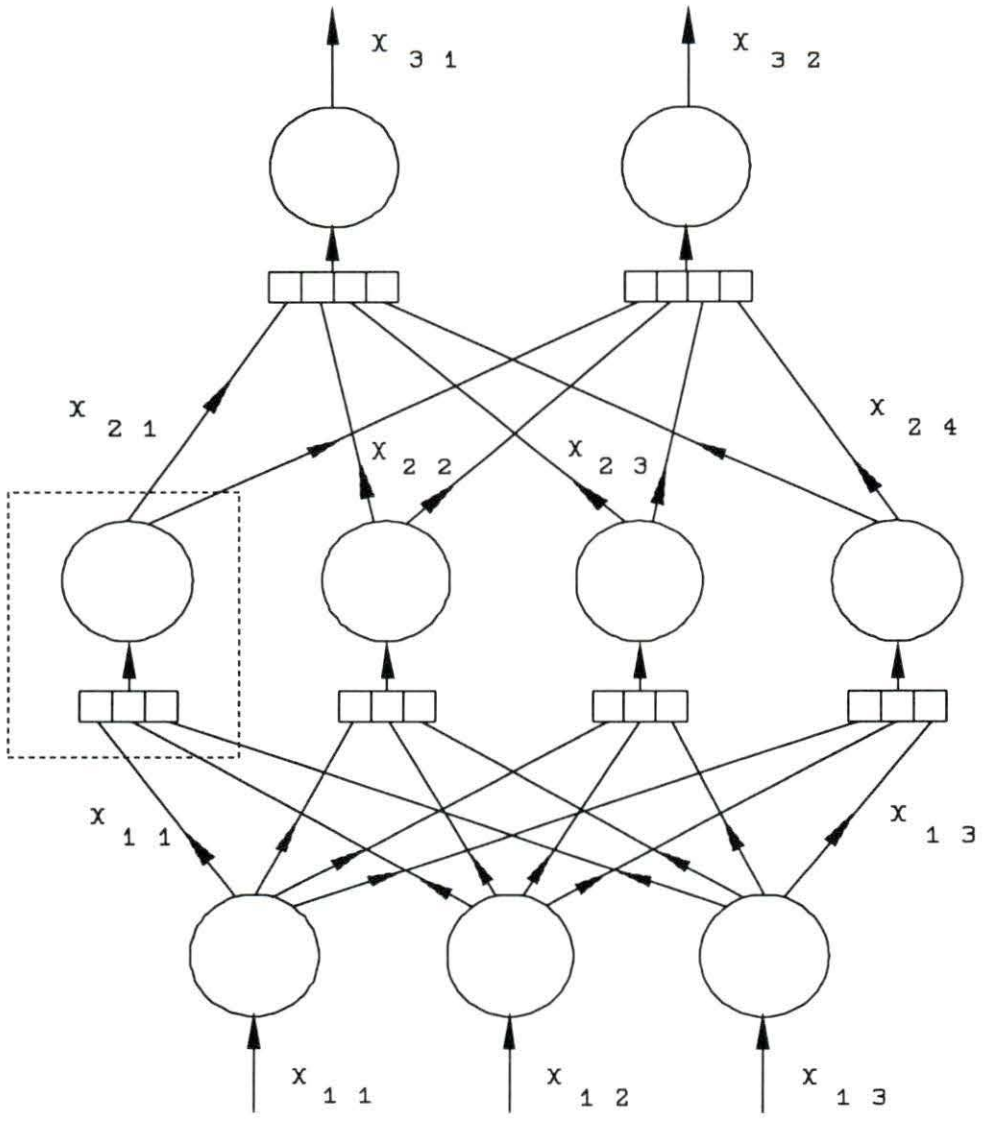


Figure 2.2 A three layer MLP during the feed-forward phase.

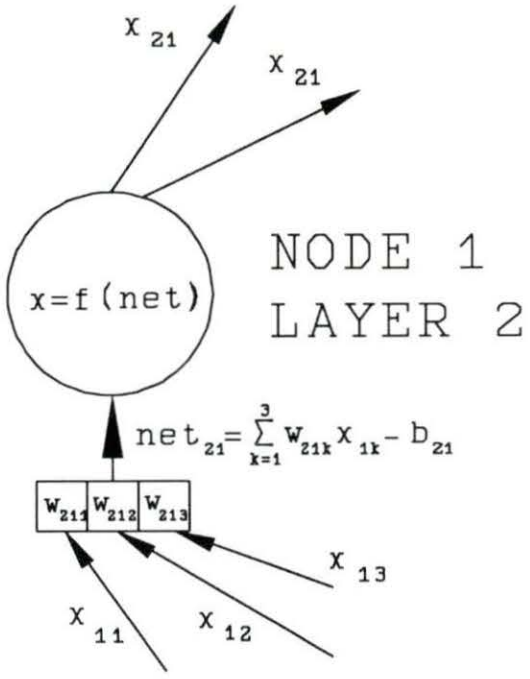


Figure 2.3 A single hidden node in an MLP.

of the nodes in the next layer where the process is repeated with the new input values and with the next layer's weight values. When the output layer is reached, the output from the nodal transfer function is also the output from the network.

The nodal transfer or activation function used in this work is the arctangent sigmoid function described in Equation 2.3.

The arctangent function is preferred over the more common

$$f(x) = \frac{1}{\pi}\arctan(x) + \frac{1}{2} \tag{2.3}$$

exponential sigmoid function of the form

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.4}$$

because the latter tends to cause underflow and overflow errors when the magnitude of the weights gets large. In this work, all of the neurodes in the hidden and output layers use the same activation function.

The process just described is the feed-forward pass of the MLP. For a trained ANN, the forward pass produces the answer completing the process. For an untrained ANN, however, the forward pass is just the first step, since the output is most likely incorrect.

Back Propagation

The backward pass, depicted in Figure 2.4, is the beginning of the process whereby the error is calculated and the weights that produced the error are changed. The back propagation (BP) algorithm, as derived by Hecht-Nielsen[11], is used to modify the weights.

Output Layer

The BP algorithm uses a gradient descent procedure[4] to adjust the weights. Plotting the error against various combinations of weights creates an N-dimensional surface with peaks where the weight combinations give large errors and

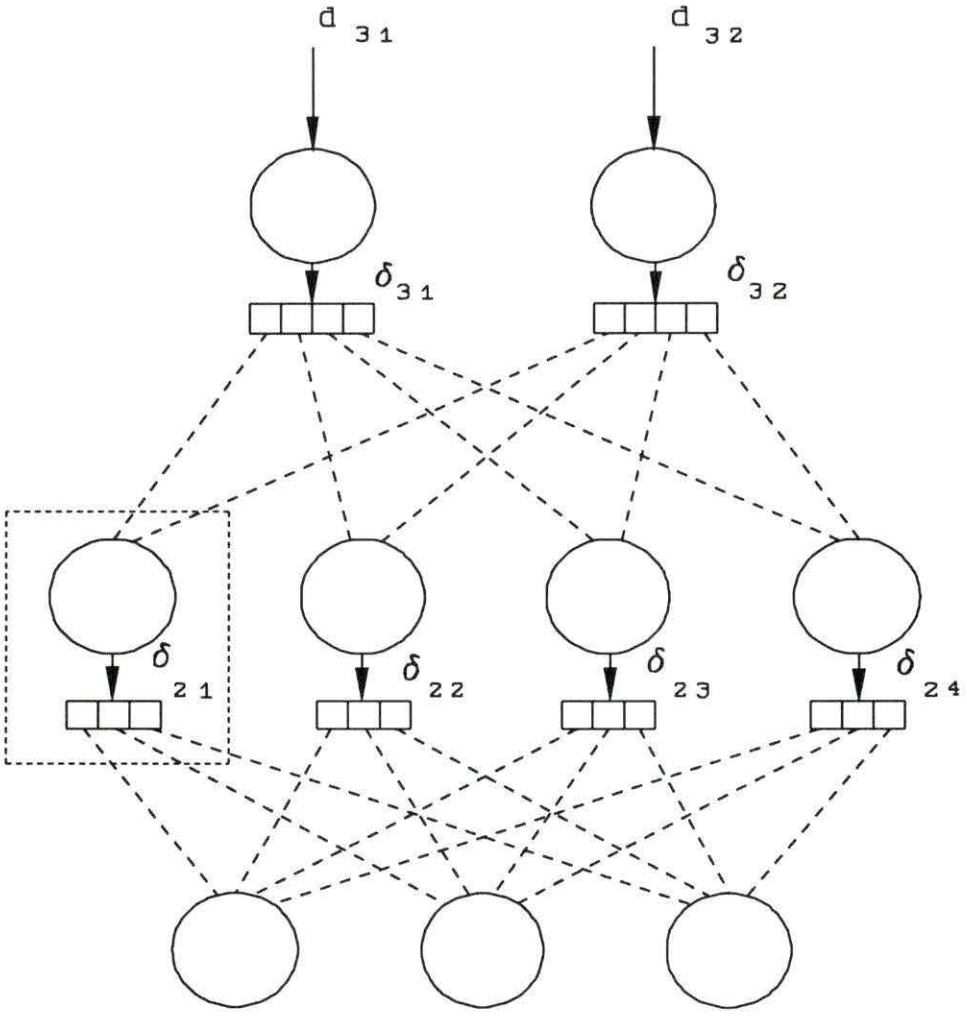


Figure 2.4 A three layer MLP during the back propagation phase.

valleys where the error is small. If the current weight vector is located on a peak on that error surface, the gradient descent algorithm attempts to move the network down the slope of the error surface by adjusting the weights in the direction of steepest descent.

The backward pass begins at the output layer. The error, defined as the difference between the desired output and the calculated output, is computed for each neurode in the output layer. To minimize the error function, the derivative of the activation function, Equation 2.5, is used to find the slope of the error surface.

$$f'(x) = \frac{1}{1+x^2} \tag{2.5}$$

Inserting net_{ij} into Equation 2.5, results in the slope of the error surface at that point. Then, multiplying by the difference between the desired and the calculated output, scales the δ_{ij} term to compensate for large or small errors. The estimate of the weight change is obtained by multiplying δ_{ij} by the initial input, see Figure 2.5.

The new output node weights, w_{ijk}^{new}, are found using

$$w_{ijk}^{new} = w_{ijk}^{old} + \eta e_{ijk} \tag{2.6}$$

where η is the learning rate ($0 < \eta < 1$) and e_{ijk} is the error for each weight. This error is calculated using

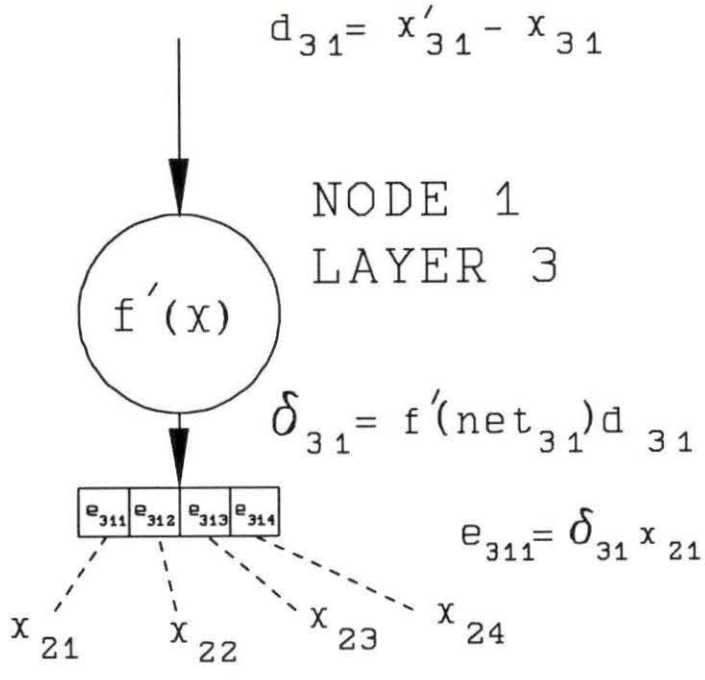


Figure 2.5 Output node during back propagation.

$$e_{ijk} = \delta_{ij} x_{i-1,j} \tag{2.7}$$

where $x_{i-1,j}$ is the output from the previous layer and δ_{ij} is defined as

$$\delta_{ij} = f'(net_{ij}) d_{ij} \tag{2.8}$$

where f' is defined in Equation 2.5 and d_{ij} is the difference

between the desired and actual outputs of node *j* in layer *i*.

The bias term is trained in a similar manner, Equation 2.6 becomes

$$b_{ij}^{new} = b_{ij}^{old} + \eta[eb]_{ij} \tag{2.9}$$

where η is the learning rate ($0 < \eta < 1$) and $[eb]_{ij}$ is the error for each bias which is calculated using

$$eb_{ij} = \delta_{ij} net_{ij} \tag{2.10}$$

where net_{ij} is the summed, weighted input to the node from the previous layer and δ_{ij} is defined as before in Equation 2.8.

Hidden Layer

Calculating δ_{ij} using Equation 2.8 is applicable for the output layer where d_{ij} is known, but what about the hidden layers where the correct output is not known? The back propagation algorithm deals with this problem by determining each hidden node's contribution to the error at the output nodes. This contribution is then used in the weight change calculations. Each hidden layer node's contribution to the output error is calculated using

$$\delta_{ij} = f'(net_{ij}) \sum_{i-1}^{N_{i+1}} w_{i+1,jk} \delta_{i+1,k} \tag{2.11}$$

see Figure 2.6. Here *i* signifies the current layer and *i+1* signifies the next layer (which is the output layer if the ANN has a three layers).

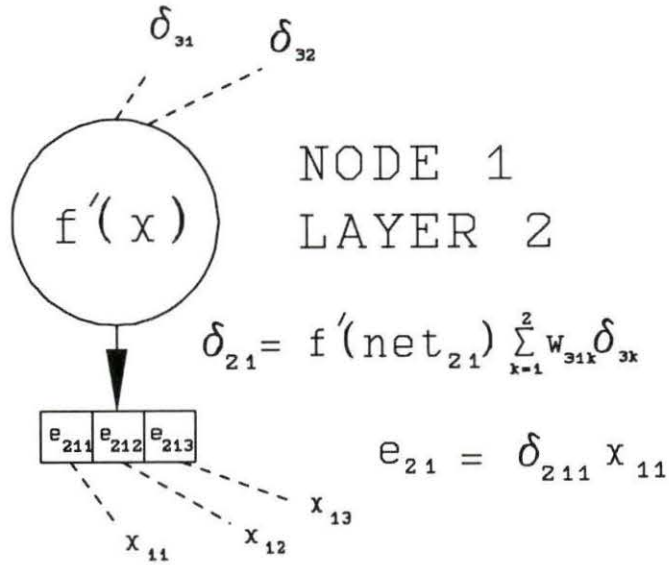


Figure 2.6 Hidden node during back propagation.

Batch Training and η

There are two ways to present the data to the ANN during the training phase, on-line and batch mode. When on-line training is employed, the weights are adjusted after each pattern from the training set has been passed through the network. By contrast, when batch training is employed the weights are adjusted only after all of the patterns in the training set have been fed through the ANN. In this work batch training was employed and the error terms, e_{ij}, act as

accumulators of the error from each training pattern. Only after all of the training patterns have been fed both forward and backward through the network are the weight values changed.

The learning rate, η, is used so that each of the weights are adjusted only a fraction of the amount computed by Equation 2.7. The larger the value of η the larger the size of the step down the slope on the error surface. η can range from 0.0 to 1.0, 0.3 is a typical value and was used in this work. Since batch training was used in this work, η was divided by the number of patterns in the training set to account for the accumulation of the error from each pattern. Also in this work, η was divided by the number of nodes in the previous layer in an attempt to balance the rate of learning in each layer.

In addition, Equation 2.6 was modified as follows

$$w_{ijk}^{new} = w_{ijk}^{old} + \eta e_{ijk}^{new} + \alpha e_{ijk}^{old} \qquad (2.12)$$

where η is, again, the learning rate and α is a number on the range [0.0,1.0] that is multiplied with the accumulated error from the previous training batch. The use of the additional term, called momentum, helps to speed convergence and may help the model escape from a local minimum, see Hecht-Nielsen[11]. In this work, the momentum term is applied only if the previous change was in the downhill direction.

RMS Error

To quantify the error of the network output over all of the output nodes, M, and for all of the training patterns, N, the root mean square (RMS) error is calculated

$$RMS = \sqrt{\frac{1}{N \cdot M} \sum_{j=1}^{N} \sum_{i=1}^{M} d_{ij}^2} \qquad (2.13)$$

where d_{ij} is the difference between the calculated and the desired output.

Network Architecture

The number of input and output nodes is usually determined by the problem to be solved. As a minimum there should be one input node for each unique independent variable and there should be at least one output node for each unique dependent variable. But, how should the number of hidden nodes and hidden layers be determined?

This question has long been the topic of heated debate in the artificial neural network community. In Hecht-Nielsen[11] Kolmogorov's Theorem is shown to prove that any continuous function $f:[0,1]^n \rightarrow \mathbf{R}^m$, $f(\mathbf{x}) = \mathbf{y}$, f can be implemented exactly by a three layer feed forward network that has n input nodes, (2n+1) hidden nodes and m output nodes. In general, it is widely believed that the smaller the number of hidden nodes the better the generalization and that the number of training patterns should be greater than the number of weights in the network[17].

Bartlett[1] and Basu[2] use a dynamic node architecture scheme to train problems with simulated condensation and back propagation, respectively. They start the training process with a few hidden nodes and add nodes until the ANN is able to learn the mapping to a pre-determined low error level. When the error is low enough, the least important hidden node is deleted and the network is retrained, if necessary. If the error again falls below the pre-determined limit, more nodes are deleted until the ANN will no longer learn the mapping. The process of adding and deleting nodes continues until the minimum number of hidden nodes required to map the input to the output is determined. They have shown that better generalization occurs with the fewest number of hidden nodes.

Verification

Since it is difficult to verify that an ANN has been correctly constructed, or programmed, an acceptable way to show that the ANN is performing properly is to model several well known problems. To verify that the ANN constructed for this work does indeed model problems correctly, three examples were used as benchmarks, these are: the exclusive-or problem (XOR), the eight-to-one decoder problem[2] and the Lippmann circle problem[20].

The first example, the exclusive-or (XOR) problem, is taken from the Boolean function in linear algebra. By definition, the XOR function returns a positive response if

one but not both of the inputs has the value 1.0. A network was constructed with two input nodes, three hidden nodes and one output node and was trained to an RMS error of 0.01. The input pairs and the desired and calculated output are shown in Table 2.1. Note that the desired outputs have been normalized

Table 2.1: XOR inputs and outputs.

Input 1	Input 2	Desired Output	Calculated Output
0.0	0.0	0.1	0.08227
0.0	1.0	0.9	0.89896
1.0	0.0	0.9	0.89889
1.0	1.0	0.1	0.10913

on the range [0.1,0.9] to aid in convergence of the ANN. As can be seen the ANN correctly classifies the four input points.

The second example problem is known as the eight-to-one decoder problem. The three inputs represent binary bits which take the values 0 or 1. When taken together the three digit binary number can represent the decimal numbers zero to seven. The network is trained to fire one of eight outputs depending on the particular combination of zeroes and ones in the input. An ANN with three input nodes, five hidden nodes and eight output nodes was trained to an RMS error of 0.026. Again, note that the desired outputs have been normalized on the

range [0.1,0.9] to aid in convergence of the ANN. Table 2.2 gives the input/output vector pairs and Table 2.3 gives the calculated output values after the network was trained.

The third example problem is the circle problem as described in Lippmann[20]. Two hundred points, defined by their location on the x-y plane, are chosen at random for the training set. One hundred points are chosen from within the unit circle ($0 < r < 1$) and one hundred points are chosen from the annular region described by $1 < r < 5$. The data points are shown in Figure 2.7. The x and y coordinates of each of the points are the inputs to the ANN. The desired output is 0.9 if the point falls within the unit circle and 0.1 if the point falls outside the unit circle. An ANN with two input nodes, eight hidden nodes, and one output node was trained to an RMS error of 0.04. The resulting decision region is shown in Figure 2.8.

From the success of the ANN models of the three benchmark problems, one can infer with an adequate degree of confidence that the network has been constructed properly.

Table 2.2: Eight to one decoder inputs and outputs.

Input			Desired Output							
0.0	0.0	0.0	0.9	0.1	0.1	0.1	0.1	0.1	0.1	0.1
0.0	0.0	1.0	0.1	0.9	0.1	0.1	0.1	0.1	0.1	0.1
0.0	1.0	0.0	0.1	0.1	0.9	0.1	0.1	0.1	0.1	0.1
0.0	1.0	1.0	0.1	0.1	0.1	0.9	0.1	0.1	0.1	0.1
1.0	0.0	0.0	0.1	0.1	0.1	0.1	0.9	0.1	0.1	0.1
1.0	0.0	1.0	0.1	0.1	0.1	0.1	0.1	0.9	0.1	0.1
1.0	1.0	0.0	0.1	0.1	0.1	0.1	0.1	0.1	0.9	0.1
1.0	1.0	1.0	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.9

Table 2.3: Eight to one decoder trained outputs.

#	Calculated Output							
1	.975	.101	.106	.099	.099	.109	.107	.038
2	.116	.896	.104	.104	.113	.104	.030	.098
3	.106	.108	.888	.102	.103	.039	.095	.112
4	.048	.098	.106	.884	.039	.097	.107	.086
5	.116	.104	.088	.047	.906	.088	.105	.101
6	.097	.106	.037	.099	.095	.889	.098	.111
7	.104	.041	.107	.106	.107	.102	.896	.101
8	.016	.101	.104	.107	.101	.112	.101	.894

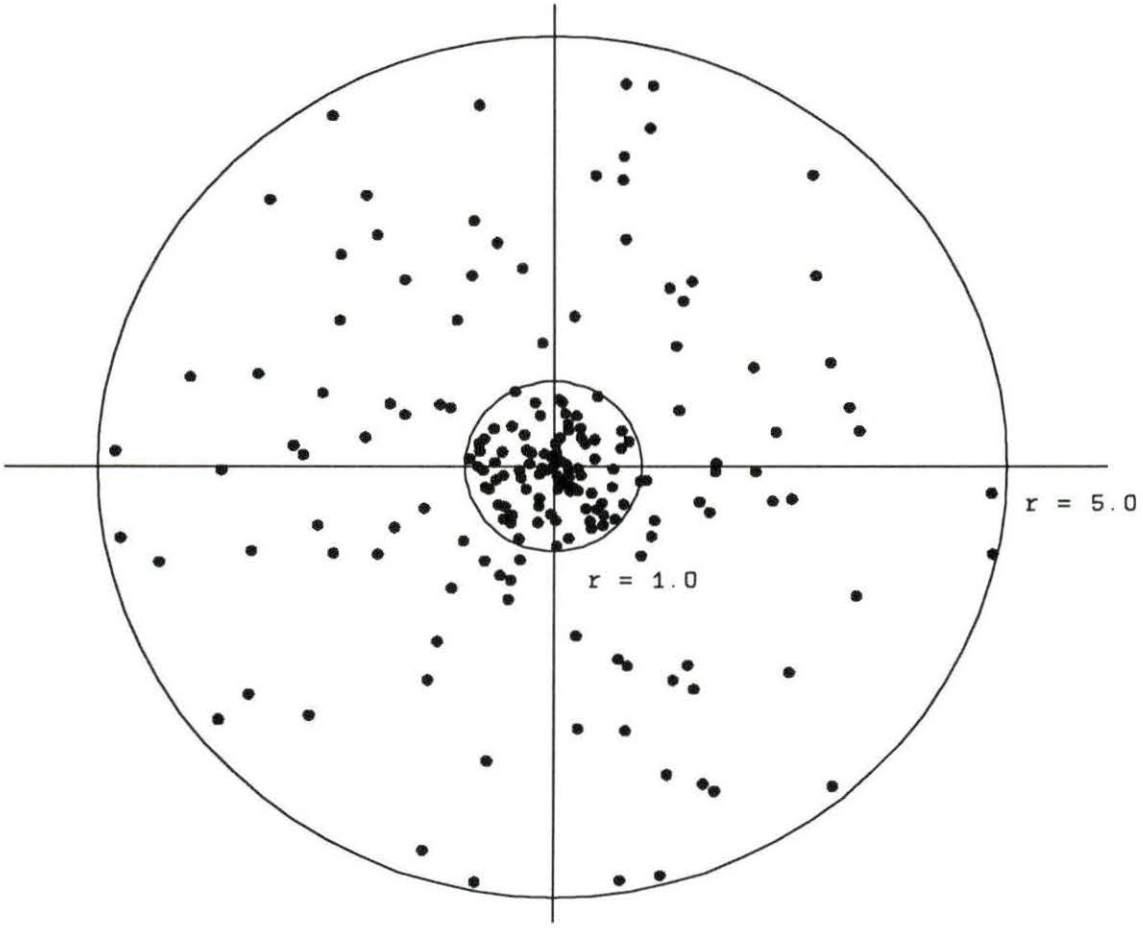


Figure 2.7 Input data for Lippmann's circle problem.

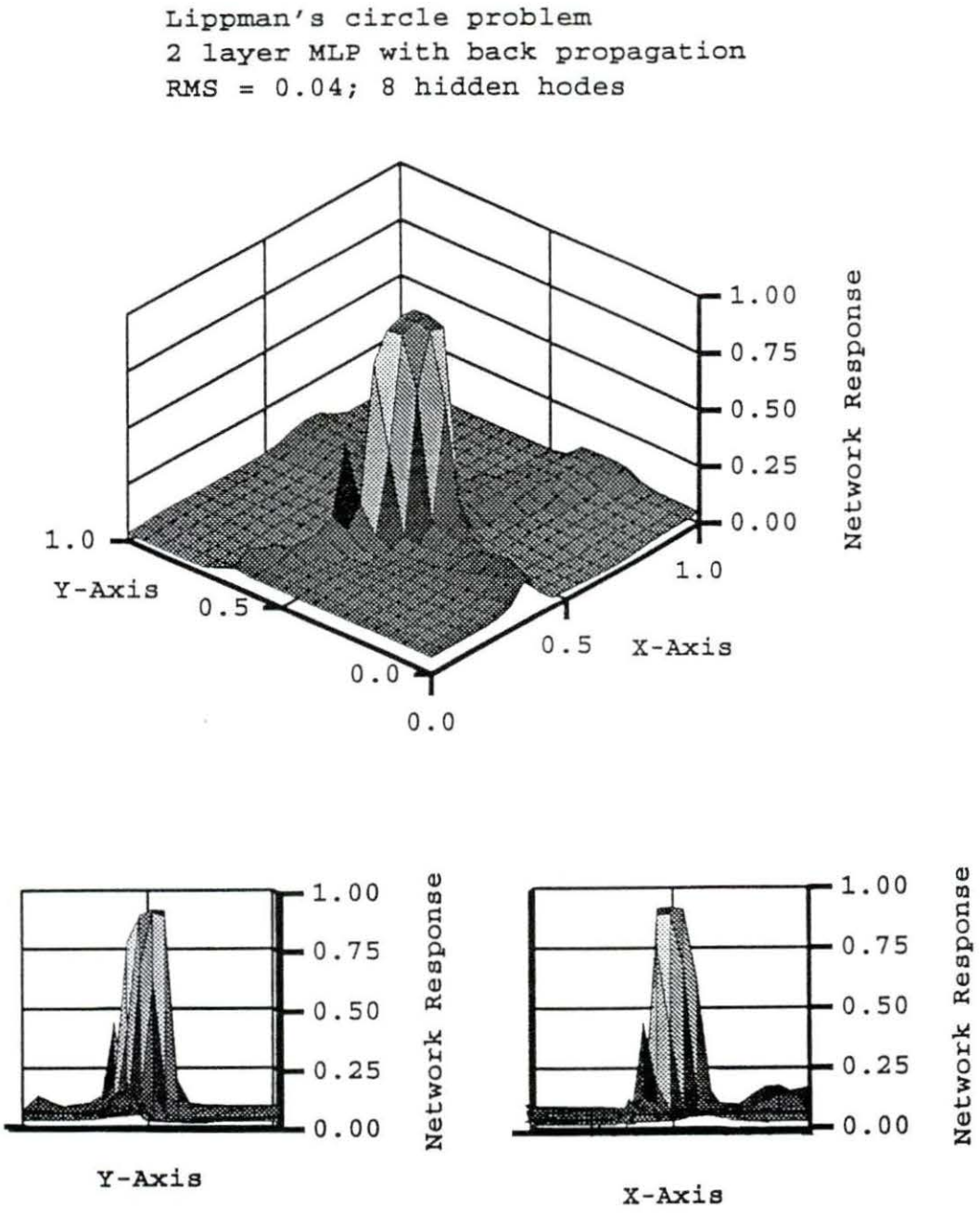


Figure 2.8 ANN decision region for Lippmann's circle problem.

CHAPTER 3. NUCLEAR REACTOR CORE RELOADS

Introduction

The periodic core reloading of nuclear reactors has long been a topic of study and research[6,7,10,14-16,18,21,25,26,32]. In general, the goal of these studies has been to find a method to optimize reactor performance and safety by way of "shuffling" the core reload pattern. The parameters to be optimized may vary depending on the particular reactor or specific circumstances at hand. For example, if there is a concern about pressure vessel embrittlement, then the objective may be to minimize peripheral neutron leakage while constraining cycle length and relative power peaking to specified design limits. Nevertheless, regardless of the criteria employed, the idea is the same, to determine an optimum set and arrangement of fuel assemblies and burnable poisons in the core for each fuel cycle.

Nuclear Fuel Management

The job of acquiring the nuclear fuel, placing it into the reactor core, storing and/or disposing of the spent fuel and all related aspects to these tasks is called nuclear fuel management. The term out-of-core fuel management is used to describe the overall long term strategy to purchase nuclear fuel for several fuel cycles in the planning horizon. The

contracts between the utility and the fuel manufacturer usually specify fabrication details such as fuel enrichment, pellet size, cladding composition, quantity and type of burnable poisons, etc.. In contrast, the term in-core fuel management, restricted to a single fuel cycle, is used to describe the process whereby the existing fresh and used fuel assemblies (and burnable poisons) are arranged into the best loading pattern (LP) possible. In this work, the term fuel management will refer to in-core nuclear fuel management.

A brief description of fuel management terminology such as batch sizes, loading configurations, pin peaking, boron concentrations, burnup, k_{eff} and burnable poisons follows.

A fuel batch is a group of fuel assemblies which have been in the reactor for the same number of cycles. The four batches discussed in this work are: new fuel, new fuel with burnable poison, once burned fuel and twice burned fuel. Once burned fuel, as the name implies, has been in the core for one fuel cycle and is about to enter its second cycle of service. Similarly, twice burned fuel has been in the core for two fuel cycles and is about to enter its third, and usually last, cycle of service. Since about one third of the core is removed each cycle, batch sizes are roughly one third of the total number of fuel assemblies in the core.

Loading configurations refer to the various patterns in which the fuel assemblies may be arranged. Two of the most

common schemes for loading a reactor core are referred to as the in-out and the out-in loading strategies. The out-in strategy places the new fuel assemblies towards the periphery of the core and has the advantageous effect of flattening the power profile across the core. These fuel assemblies then 'migrate' toward the center of the reactor during subsequent cycles. The in-out strategy, on the other hand, places the new fuel assemblies more toward the center of the core from which they 'migrate' toward the periphery with each cycle. The advantage of this latter strategy is a lower neutron leakage at the periphery, leading to less neutron damage at the reactor vessel wall.

Some loading configurations can be eliminated out of hand by means of engineering intuition and experience, for example: non-symmetric patterns, patterns with new assemblies adjacent to control rods, patterns with two new assemblies side by side, patterns with new assemblies at the core periphery (in the in-out strategy) or patterns with a new assembly in the center location, etc. are all untenable. The reason for eliminating LPs in this manner is to reduce the number of possible candidates to a more manageable size.

Pin peaking refers to the ratio of the maximum power level at a specific fuel pin in a fuel assembly to the core average power level. In general, a flat power profile across the core is desired and thus upper design limits for the pin

peak ratio must be obeyed. These criteria are conservatively set to prevent the temperature of any fuel pin from exceeding thermal limits designed to preserve the integrity of the fuel pin and the fuel assembly.

Reactivity is a term used to describe the state of balance between the production and loss of neutrons (and thus fissions) in a nuclear reactor. The mathematical parameter used to define the reactivity level of the core is k_{eff}, also known as the multiplication factor. In the absence of burnable poisons (discussed later), k_{eff} at beginning of cycle (BOC) is proportional to the cycle length, so a longer cycle length can be implied by maximizing the BOC k_{eff}.

Boron dissolved in the coolant is a neutron absorber and is employed to adjust k_{eff} to a value of 1.0, denoted "critical". Boron dilution, also known as "chemical shim", compensates for the depletion of the fuel and for the buildup of fission products. K_{eff} can be adjusted with control rods, chemical shims and burnable poisons. The use of a chemical shim reduces the number of expensive control rods that are required to adjust k_{eff} and control the reactor. Also, since the shim is dissolved in the coolant and is evenly distributed throughout the core, the concentration can be reduced to account for fuel burnup without altering the power distribution across the core. The amount of boron required, measured in parts per million (ppm), is proportional to the

unadjusted k_{eff} thus, maximizing the ppm of boron also implies maximizing the unadjusted k_{eff} and cycle length.

There are several definitions of burnup. In general, burnup of the fuel is defined as the total energy released by the fission of a given amount of fuel and is measured in terms of megawatt-days(MWd). Specific burnup, on the other hand, is defined as the total energy released by the fission per unit mass of fuel and is measured in terms of megawatt-days per metric ton (MWd/t). Finally, fractional burnup is defined as the total number of atoms of fuel that undergo fission per total number of fuel atoms initially present in the fuel.

A burnable poison (BP) is constructed from a material that, like boron, has a negative effect on reactivity. The isotope that is formed when the BP absorbs neutrons does not have a large negative effect on reactivity, thus the term "burnable". Placing burnable poison pins within the fuel assemblies or blending it within the fuel itself has two major objectives: (1) to hold down the positive reactivity of the fuel at BOC thus allowing more fuel (higher enrichment) to be loaded, thereby giving rise to a longer power cycle and (2) to shape the power distribution in the reactor.

Core Reload Methods

For in-core fuel management, the massive number of the possible core LPs, makes it highly improbable that an optimal solution can be found by means of an exhaustive search method.

For that reason, many techniques have been developed as alternatives to the trial and error approach of an experienced engineer. Some of these methods are: backward diffusion[6], linear empirical core models[25], expert systems[10], constant power depletion[16] and simulated annealing[26]. These methods will be discussed in the following sections.

Backward Diffusion

The diffusion equation is well known to nuclear engineers and is typically used to calculate the neutron flux and power distribution in a reactor core for a given core loading. The backward diffusion method[6] was derived so that the engineer could assume a desired power distribution and then generate the corresponding reactivity distribution. From this, the core loading is then inferred from the available fuel assemblies by best matching the reactivities of fuel assemblies to the computed reactivity distribution. A forward diffusion calculation is then performed to obtain the corresponding power distribution which may or may not satisfy constraints on cycle length, power peaking, etc.. If one or more of the constraints are violated, the LP is discarded and the process is repeated.

Although the backward diffusion method has been extensively employed in an actual industrial scenario, several assumptions are made which limit the accuracy of the calculation. Also, there is no optimization of the LP which

best matches the target power distribution.

Linear Empirical

Fuel management via linear empirical core models[25] is a method whereby a model is created which relates state variables to control variables to determine the optimal BOC k_∞. The state variables are assembly power fractions and burnup increments; the control variables are the zone enrichments. The method tries to indirectly model the reactor core by treating it as a linear programming problem, bypassing the computationally expensive direct calculations. The method assumes uniform poison distribution, linearity between state and control variables and zero BOC burnup. The assumption of linearity between the state and control variables limits the accuracy of the predictions of the highly nonlinear reactor core to first order at best.

Expert Systems

A heuristic, as defined by Webster[37](p. 568), is "involving or serving as an aid to learning, discovery, or problem solving by experimental and especially trial-and-error methods; also: of or relating to exploratory problem-solving techniques that utilize self-educating techniques to improve performance <a ~ computer program>". The expert system method in fuel management uses a computer algorithm to heuristically search for near optimal solutions to the core reload problem. Rules based on past experience are developed for and

implemented by the expert system. These rules limit the solution space by eliminating LPs which are untenable such as, patterns with a fresh fuel assembly in the core's center position. The expert system must be used in conjunction with a core neutronics code to evaluate the candidate LPs produced by the expert system.

In Galperin[10] *et al.* the expert system was programmed to avoid loading fuel schemes that promoted high local power peaks while keeping the lowest possible power at the periphery. The solution space was assumed to be divided into groups or regions characterized by specific patterns, and it was also assumed that within these groups there were large numbers of almost identical solutions. The core loading begins with fresh fuel assemblies followed by the once and twice burned assemblies. Examples of some of the rules employed are: no loading of fresh fuel into the inner part of the core, no two fresh fuel assemblies should have a common surface, no twice-burned fuel assembly should be loaded into the outermost positions.

The probability of finding a global optimum core reload pattern using this method is lessened by the fact that the solution space is restricted. For example, suppose a specific search method uses binary fuel exchanges to develop new LPs. If THE optimal core LP happens to be a binary fuel exchange away from an LP which has already been rejected, the expert

system will never find it.

Constant Power Depletion

The constant power, or Haling, depletion method[16] assumes the optimum power distribution is constant throughout the entire cycle and thus can deplete the core in a single time step. This method is usually used in the core reload design of boiling water reactors (BWRs) where greater flexibility in reactivity control is possible. In the design of core reloads for pressurized water reactors (PWRs), the Haling depletion method provides a consistent means of comparing new reload patterns without taking into account optimal control policy. With the core loading and control policy decoupled, the two factors can be optimized separately.

Kim *et al.* first performs an eighth core, two dimensional, Haling power calculation on the initial core reload pattern using SIMULATE-3[29], a commercially available nodal diffusion code. Next, the objective function is evaluated to find the end of cycle (EOC) critical boron concentration, after which the nodal power peaking is calculated. The results are recorded and a binary fuel exchange (two fuel assemblies switch positions) is performed using heuristics to ensure that untenable LPs are eliminated. The process is repeated until the core LP which maximizes cycle length and stays within the constraints for the initial LP is determined.

After the best LP has been found, the second phase begins by estimating the distribution of burnable poisons (BPs) necessary to match the Haling distribution. The optimal distribution of BPs is determined separately using a first-order accurate perturbation approximation. The drawbacks to the Haling depletion method are similar to those of backward diffusion in that the assumptions about the optimum LP are made a priori and no optimization is done.

Simulated Annealing

Simulated annealing is a form of artificial intelligence that is analogous to the annealing process in metals. In the annealing of a metal, the atoms align themselves as the metal cools - reaching an arrangement which minimizes the energy state of the solid. In simulated annealing, the trainable parameters or weights are adjusted so as to reach a lower error level. A weight vector which reduces the error is accepted with 100% probability. Entrapment into a local minimum is avoided by allowing the acceptance of weight vectors with a higher error level at a low probability. This probability of acceptance is reduced with time and is analogous to the reduction of the temperature of the metal being annealed.

In fuel management, Parks[26] first used simulated annealing (SA) to optimize the performance of a fuel stringer for the British Advanced Gas Reactor (BAGR). A neutronics

code provided a measure of the cost of the fuel stringer to be minimized by the SA algorithm. Heuristics were used to decrease the size of the solution space.

Kropaczek[18] and Maldonado[21] combined simulated annealing optimization with second order nodal generalized perturbation theory to generate families of near optimal core reload patterns without the need for heuristics. The combination of these techniques into the FORMOSA code results in a method of core reload optimization which is computationally accurate and efficient with a minimal number of assumptions made[21]. The expense of the direct calculation of the core parameters is considerably lessened by the use of nodal generalized perturbation theory.

Stevens[32] recently presented work on the optimization of reactor core reload designs by simulated annealing with the inclusion of heuristics for candidate LP generation. The commercially available SIMULATE-3[29] is used to evaluate each LP which is generated by the combination of heuristics and artificial intelligence.

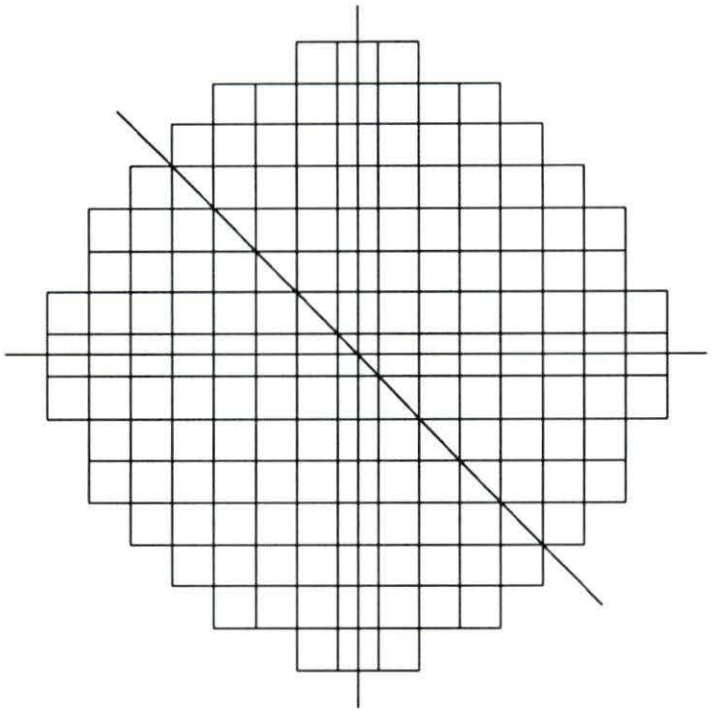
Scope of Problem

With the advent of faster and more powerful computers, methods of finding optimal core reload patterns which were not feasible a few years ago are now becoming tractable. Despite that, however, the core reload problem is still too massive to be solved by a purely exhaustive search technique. Given a

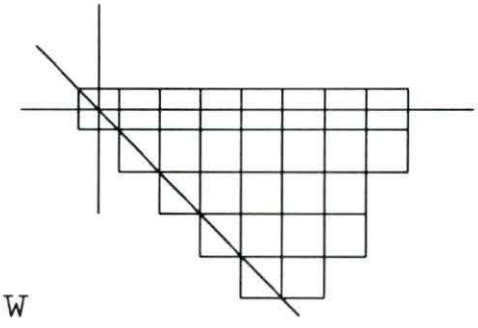
reactor with 157 fuel assemblies, approximately 105 of these assemblies are once or twice burned and are therefore unique (different burnups, power histories, etc.) and 52 are identical new fuel assemblies (assuming no burnable poison is used). The number of possible combinations is 157!/52! or 1.45×10^{210}. If one half of the new assemblies have burnable poisons then the number of possible combinations is 157!/(26!·26!) or 7.21×10^{224}.

To direct the focus of this research more so upon the ANN application, several assumptions were adopted to reduce the extent of the search space. Since the reactor core is geometrically symmetric the core LP and its parameters (i.e., assembly reactivity, burnup, neutron flux and thus reactor power) are also assumed to be symmetric. Depending on the placement of control rods, it can usually be assumed that the reactor is symmetric down to an eighth of the core, see Figure 3.1. When one eighth core symmetry is assumed, using the same core with 157 fuel assemblies, the number of available fuel locations is reduced to 26, nine are for fresh fuel leaving 17 positions for once and twice burned assemblies. The number of possible combinations drops to 26!/9! or 1.11×10^{21}.

An additional simplifying assumption was that all of the fuel assemblies in a batch have similar characteristics. The 1/8 core is then loaded with the three batches of identical fuel assemblies. The 'typical' once and twice burned fuel



Full Core - top view



Eighth Core - top view

Figure 3.1 Plan view of full and eighth core models.

assemblies are determined by calculating the average characteristics over the entire batch. When this further simplification is made, the number of possible combinations decreases to 26!/(8!·8!·9!) or 6.84×10^{11}; if burnable poisons are present the number of possible combinations is 26!/(8!·8!·5!·4!) or 8.61×10^{13}. These simplifying assumptions were drawn in agreement with the SOA1 Database described next.

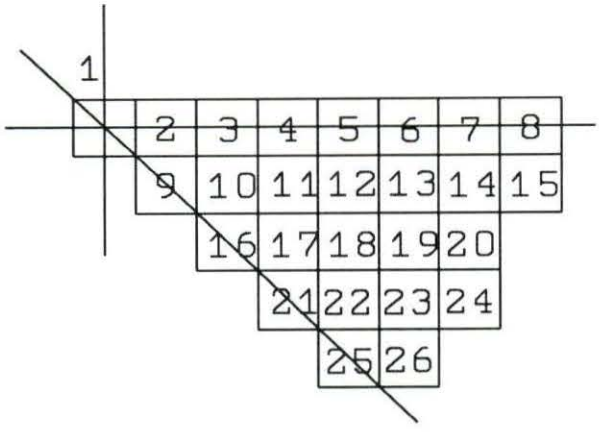
SOA1 Database

To aid nuclear utilities in developing a system for designing a near optimal core reload pattern, Studsvik of America developed a database[31] containing over 300,000 core reload patterns and their respective core parameters as calculated by the Studsvik package SIMULATE-3[29]. The core modeled for the database was an unspecified PWR with 157 fuel assemblies. To simplify matters, an average fuel assembly was assumed for each batch and eighth core symmetry was used. Parameters such as critical boron concentration, k_{eff}, pin peaking ratio, and burnup were calculated at four depletion steps in the fuel cycle and recorded in the database.

The description of the format for the database follows. The first entries are 26 integers ranging in value from one to four. A one indicates twice burned fuel, a two indicates once burned fuel, a three indicates new fuel and a four indicates new fuel with burnable poison. The position of the integer in the string of 26 is important because it indicates the

location of the particular fuel assembly in the eighth core. In Figure 3.2, the top figure shows the numbering system for the eight core model - the position in the string of integers is indicated by the number shown in the fuel locations. The bottom figure shows a typical LP with the various "flavors" of fuel (here flavors describe the fuel batch: twice burned - 1, once burned - 2, fresh - 3, and fresh with burnable poison - 4).

Following the 26 LP integers are thirty floating point numbers. These numbers represent the calculated core parameters for that particular LP at four different depletion steps. The first and second numbers are the hot zero power (HZP) critical boron concentration and the HZP moderator temperature coefficient, respectively for the LP. The next five numbers are: critical boron concentration, k_{eff}, core pin peaking factor, ratio of assembly pin peak to assembly averaged power in the peak assembly and the location of the pin peak assembly. These five parameters are calculated for the first depletion step in the fuel cycle and the next fifteen numbers are the parameters for the last three depletion steps. The final eight numbers in the database entry are the LP's average end of cycle (EOC) assembly average burnup and peak EOC assembly average burnup for each of the four batches.



Eighth core numbering system

1	2	1	2	2	4	2	3
	1	4	2	2	2	4	3
		1	4	2	4	2	
			2	2	4	3	
				2	3		

Typical loading pattern

Figure 3.2 Eighth core model numbering system and example core loading.

CHAPTER 4. THE PROBLEM AND ITS SOLUTION

Introduction

In the optimization of core reload patterns, a good estimate of the key core parameters such as k_{eff} (or critical boron concentration) and pin peaking is crucial in evaluating candidate patterns. Current methods either require direct computation of these parameters for each pattern under consideration or make simplifying assumptions which decrease the accuracy of the calculations.

The commercial computer codes that are used in fuel management today must be robust enough to be used by many engineers. These codes are used on various designs of reactors at many facilities across the country, each reactor and each facility having its own peculiarities. Often different facilities will have relevant information, such as cross section data, stored in different locations or in computer files which have different formats depending on the vendor or the utility.

This work investigates the use of an ANN to estimate the pin peaking ratio and the critical boron concentration for a typical reactor core. The network would be trained initially on data from core reload efforts for previous cycles. The data required to train the network would be the core LP and

its associated parameters - libraries of data would not be needed. As estimates are generated for the current core reload, any and all miscues could be added to the training data so that the model can be refined to better reflect the current core. In other words, the ANN learns the mapping from core reload pattern to core parameter for each specific core.

Data Collection and Processing

The SOA1 Countm Suite and Database[31] was used in this work to train the an ANN to map core LPs to their respective core parameters. The training sets were made up of 3068 or 3069 randomly chosen loading patterns. The first LP was chosen at random followed by every 100^{th} successive LP. There were 306,884 loading patterns in the database to choose from - dividing by 100 gives 3068 or 3069 depending on the number of the initial pattern. The validation set consisting of 3069 different LPs was generated randomly (a random long integer was generated and then the MOD function was applied to obtain the number of the LP).

Normalization of the Data

As is common in the training of ANNs, both the output data and the input data were normalized. The input data, consisting of a series of ones, twos, threes and fours, was converted to 0.2, 0.4, 0.6 and 0.8. The desired output values, continuous real valued numbers, were normalized on the range [0.1,0.9]. The normalization was accomplished by first

finding the largest and smallest values for each parameter over the entire set of 306,884 LPs. The range of each parameter was calculated as the difference between the largest and smallest core parameter. The smallest value was subtracted from all of the patterns in the training set. The resulting numbers were then divided by the range giving a set of numbers normalized on the range [0,1]. These numbers were then multiplied by 0.8 and 0.1 was added to the product. The result of these manipulations was a set of numbers normalized on the range [0.1,0.9].

Final Data Set

The final data set consists of 3068 or 3069 lines of input and output data. Each line contains 26 decimal numbers [0.1,0.9] (25 input values and one output value). The first 25 numbers represent the fuel batch number and its location in the eighth core. The last number on the line of data is the normalized parameter for pin peaking or critical boron concentration depending on the output of interest.

The number representing the center position in the reactor core was dropped from the training set due to the fact that all of the loading patterns have a twice burned fuel assembly in that location. Only unique inputs are important to the network mapping between loading pattern and core parameter, thus the inclusion of the center fuel assembly would have been redundant.

Parameter Prediction

Training

An ANN model was developed for the problem of predicting core parameters from the loading pattern. The inputs to the ANN are the 25 normalized fuel batch numbers. Thus, the 25 input nodes each represent a particular location in the eighth core model. The output is the desired core parameter, i.e. critical boron concentration or pin peaking ratio. To increase the probability of network generalization, it is desirable to keep the total number of weights in the ANN model less than the number of training patterns[17]. The number of hidden nodes was chosen to be 17 resulting in an ANN which has 442 weights and 18 bias values. With 3068 training patterns, there are over six training patterns per weight value. If the model can correctly classify a number of training patterns that is larger that the number of weights in the model, the implication is that the model has learned the functionality between the input and the output and has not just memorized the correct outputs for the given inputs.

Results

Due to the large number of training patterns, and LPs in general, presentation of the results is not simple. The RMS error gives a measure of the cumulative error over the entire training set, but does not give detailed information about specific LPs. Table 4.1 contains information about the RMS

Table 4.1 Training and recall results.

Core Parameter	Training RMS	Validation RMS
Boron concentration	0.0243	0.0266
Pin peaking	0.0411	0.0449

error of the training and validation sets for the critical boron concentration and pin peaking ratio.

The RMS error reported in Table 4.1 is that of the normalized data, therefore its magnitude is not the important feature. A better measure of the ANN's ability to predict the correct output will be presented later. However, with RMS errors for critical boron concentration of 0.0243 and 0.0266 on the training and validation sets, respectively, one can infer that the network performed the desired mapping between input and output almost equally well for both data sets. It follows, therefore, that the ANN model has learned to predict the critical boron concentration of the validation set from training on the data in the training set.

For the case of pin peaking, the ANN had more trouble learning the functionality between input and output, as indicated by the larger value of the RMS errors of 0.0411 and 0.0409. However, there was still good agreement between the training and validation sets. And, it again follows, that the ANN model has learned to predict the pin peaking of the validation set to a similar degree of accuracy as the training

set.

The RMS error is, however, not the only, or even the best, way to measure the acceptability of the ANN model. Another method of presenting the results of the ANN modelling is to plot the un-normalized predicted output against the un-normalized actual output. A perfect mapping of input to output would be represented by a line on the plot of slope one and y-intercept of zero. When the appropriate error bounds are placed on the plots, it is relatively easy to see where the ANN performs well and where it doesn't.

Figures 4.1 and 4.2 show the actual or target boron concentration plotted against the predicted output from the ANN for the training and the validation sets, respectively. The lines drawn on the figures are the ±3% (approximately 57 ppm) error bands, more than 99% of the data points in the training set and more than 98% of the data points in the validation set are within these bands. This corresponds to a Pearson's product-moment correlation coefficient of 0.97995 and 0.97618 between the predicted and the actual output for the training and validation sets, respectively.

Figures 4.3 and 4.4 show the actual or target pin peaking plotted against the predicted output from the ANN for the training and the validation sets, respectively. The lines drawn on the figures are the ±10% error bands, more than 90% of the data points from the training set and more than 87% of

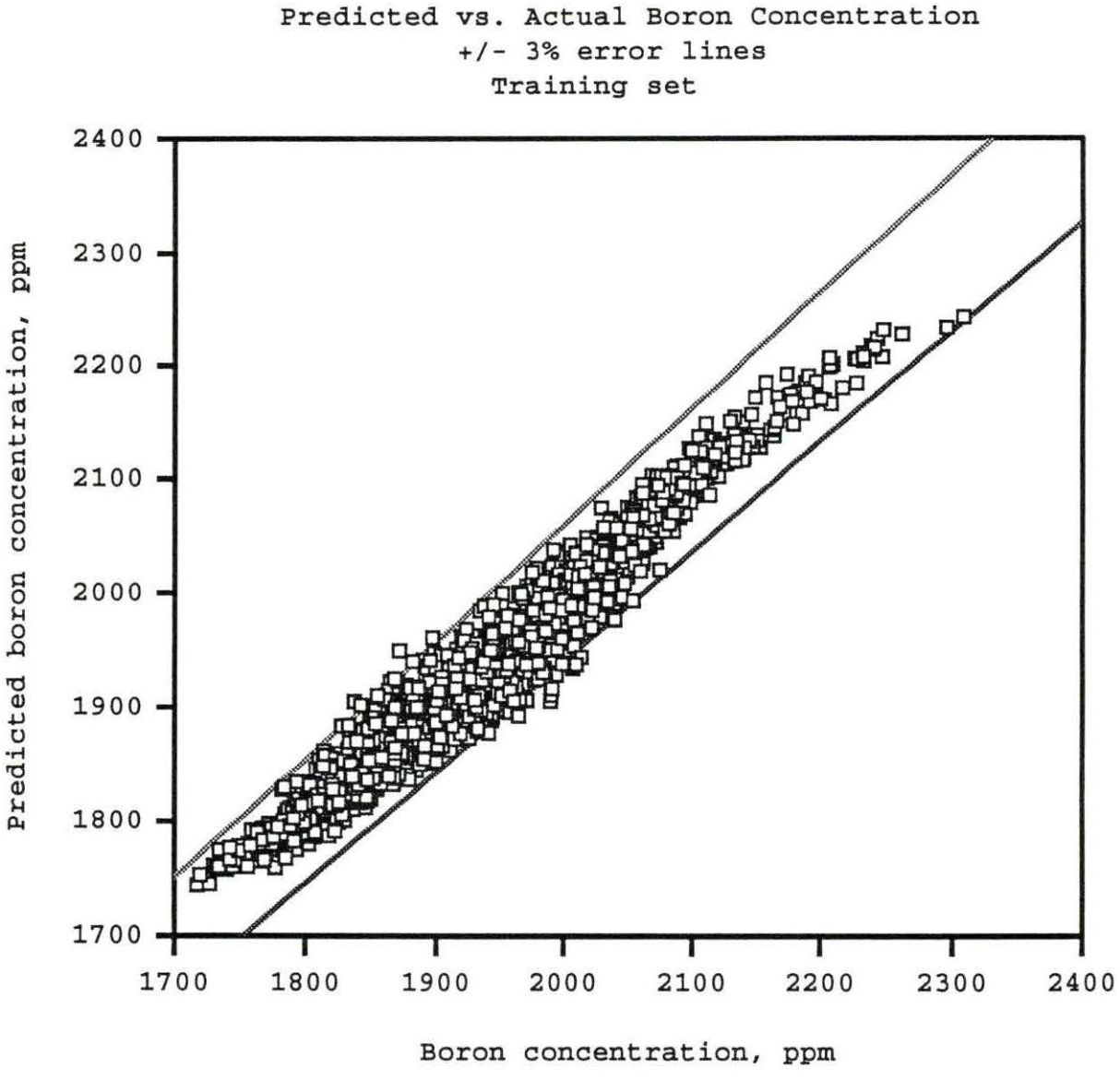


Figure 4.1 Prediction results - critical boron concentration training set.

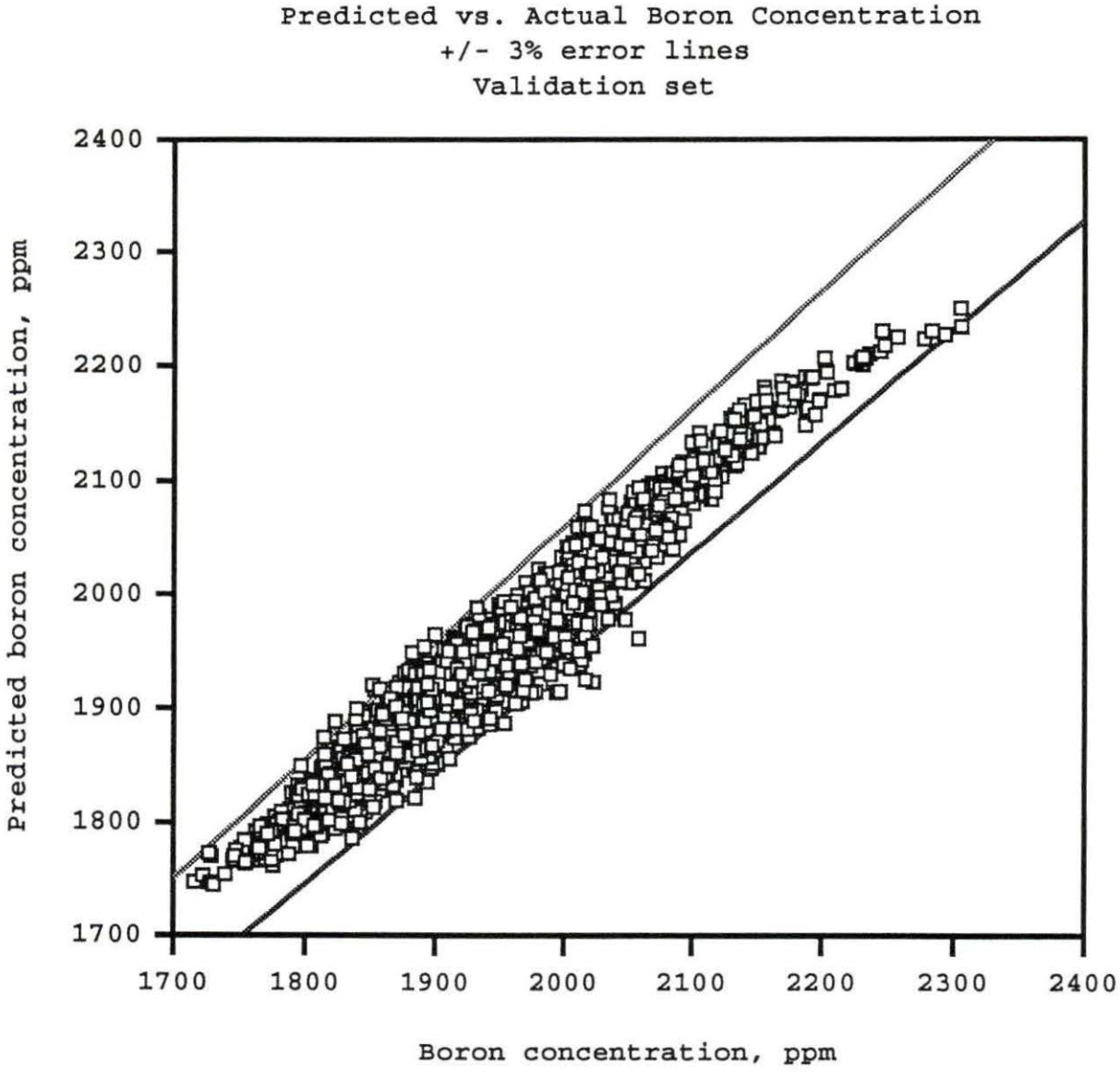


Figure 4.2 Prediction results - critical boron concentration validation set.

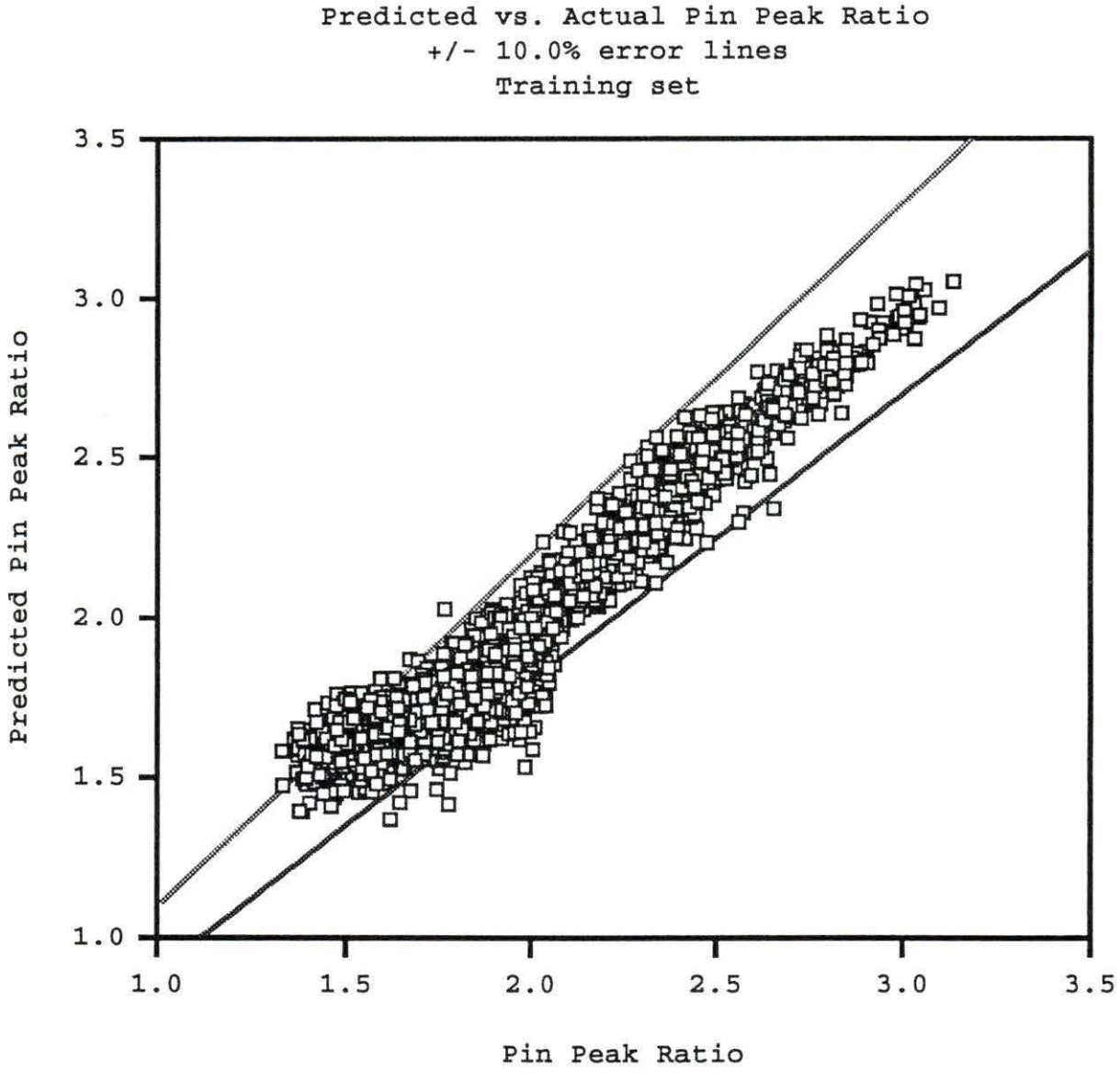


Figure 4.3 Prediction results - pin peaking ratio training set.

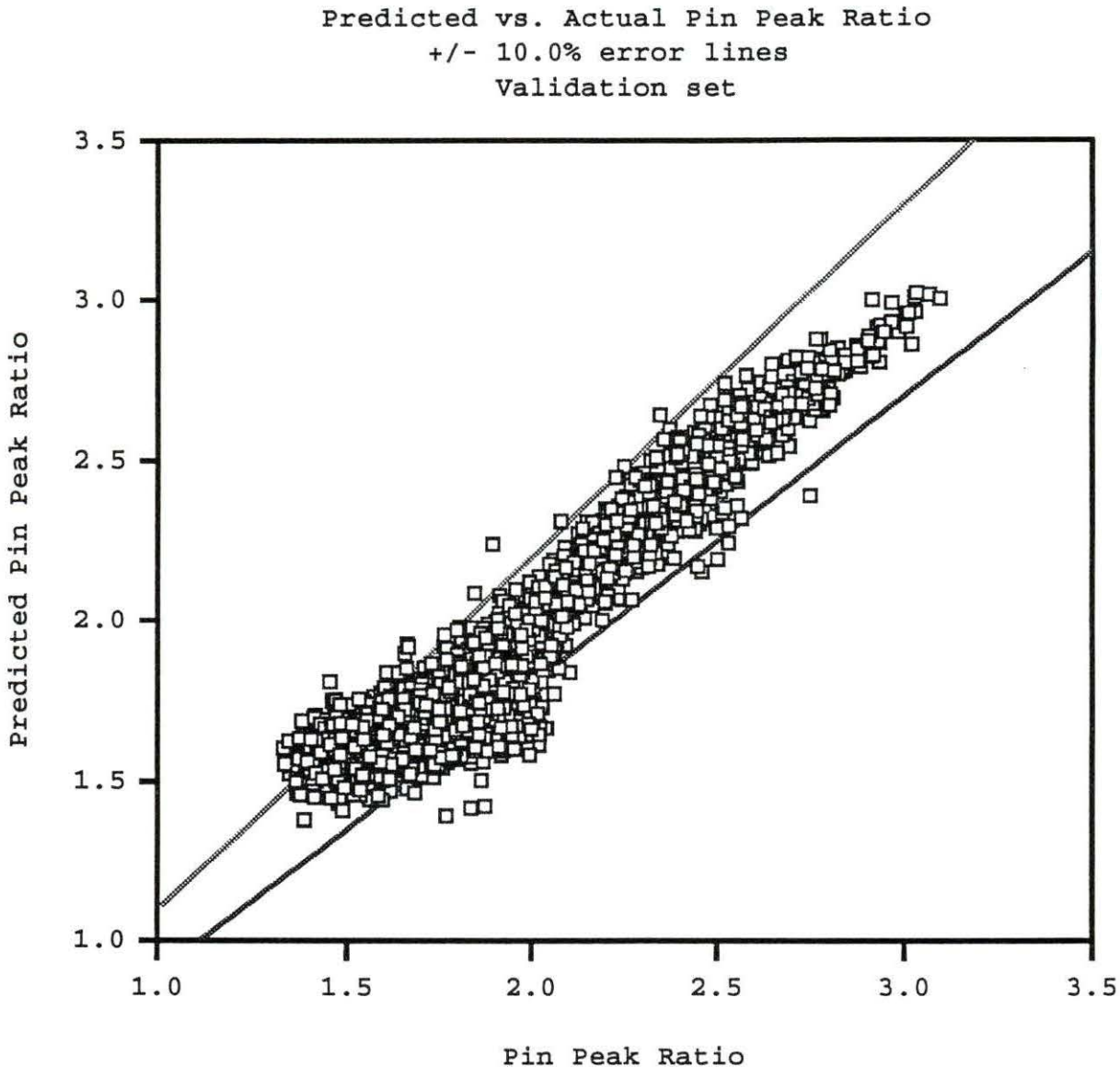


Figure 4.4 Prediction results - pin peaking ratio validation set.

the data points from the validation set are within these bands. This corresponds to a Pearson's product-moment correlation coefficient of 0.96597 and 0.95848 between the predicted and the actual output for the training and validation sets, respectively.

Discussion

When the accuracy of the results reported above are compared with the accuracy of other methods of calculating core parameters, the results are mixed. For the critical boron concentration, the reported accuracy of SIMULATE-3 is within 5-10 ppm[33] which is around 1% of a typical value for the critical boron concentration. The ANN model in this work was able to predict 75% and 71% of the core reload patterns to within 1% in the training and validation sets, respectively.

Typical errors in the calculated pin peaking ratios are also reported to be approximately 1%[33]. In this area, the ANN obviously does not perform well. A possible reason for this is in the problem definition. The critical boron concentration is related to k_{eff}, which is a global core parameter and the ANN was able to learn the mapping from input to output in the training set. The pin peak ratio, however, is a local condition and is in fact accompanied in the SOA1 Database by the location of the pin peak in the core. The inclusion of the location of the maximum pin peak ratio as a parameter in the training (and recall) set may allow the ANN

to learn a more correct functionality.

While the accuracy of the ANN predictions is not as good as the conventional methods, the speed at which the ANN can make those predictions is unparalleled. The computational inefficiency in ANN models occurs in the training phase which is separated from the production phase. Therefore, when the user wants to use the ANN to predict core parameters, the prediction for any given core LP is almost instantaneous on almost any modern personal computer or workstation. This is not true in methods where the parameter is calculated directly.

Comparison with Similar Work

In the work by H. Kim[14] *et al.* two reactor core parameters, pin peaking and k_{eff}, were predicted using an ANN trained with back propagation. The exponential sigmoid transfer function was used to construct a network with 21 inputs (the size of their eighth core), 500 hidden nodes and 18 output nodes. The input values for fresh fuel were modified by the neutron importance function under the impression that this was necessary to further distinguish the physical location of the fuel assemblies in the core in relation to position in the input vector. The eighteen output values form a 1x18 binary vector that is broken into two groups of nine bits each. The two nine bit numbers are converted into a real valued number by the group-and-weight

scheme as presented in the paper. One thousand random loading patterns were generated to train the ANN and one hundred other loading patterns were generated at random to test the ANN. The loading patterns consisted of twice burned, once burned and fresh fuel assemblies and were represented as -1, 0, 1, respectively.

With their 19,500 node ANN, Kim *et al.* were able to train the network in roughly 300 iterations to predict 90% of the power peaks and 95% of the k_{eff} values to within ±6.0% and ±0.3%, respectively. Table 4.2 summarizes the differences between the work by Kim *et al.* and the work in this paper.

Table 4.2 ANN comparison table.

Comparison	This work	Kim et al.
Architecture	25-17-1	21-500-18
Pre-processing	Normalize	Neutron imp. function
Output	Normalized real no.	18 bit binary
Batch	1,2,3,4	-1,0,1
Training patterns	> 3000	1000
1/8 core size	26	21
Transfer function	arctangent	sigmoid
Results: k_{eff}	> 98% < ±.3%*	> 90% < ±.3%
boron concentration	> 98% < ±3%	----
pin peaking	> 87% < ±10%	> 95% < ±6%

* The results for k_{eff} for this work are an approximation based on a personal communication with G.I. Maldonado[22]. The method by which the approximation was arrived at is shown in Appendix C.

CHAPTER 5. CONCLUSIONS

Based on the results presented in the previous chapter, the conclusion drawn from this work is that PWR core parameters can be predicted fairly accurately with ANNs, as shown by the predictions of the critical boron concentration. While the error level in the ANN predictions is larger than that achieved by direct calculation of the parameters, there is a considerable time savings in the ANN technique. Further time savings will be realized in ANN predictions of EOC core parameters since nodal diffusion codes must do separate calculations for each depletion step. However, until the accuracy level of the ANN parameter prediction is improved for local parameters, the usefulness of this method in nuclear fuel management will be limited.

Possible Future Work

Further work on increasing the accuracy of the ANN parameter predictions is necessary. One possible way to improve the accuracy of the prediction would be to research and develop a new ANN architecture. There is considerable knowledge about the nature of the core reload problem and the neutron diffusion equation that could be used to construct a specialized ANN architecture. An ANN which more closely models the diffusion equation should be able to learn the

underlying functionality of the core reload problem more efficiently than an ANN with a standard architecture.

The ANN model developed in this work was used to model an eighth core of a PWR. Although the core reload problem is much more complicated for BWR's, an ANN method would have the same speed advantage over direct methods used in BWR core reload design as was demonstrated in this work. Therefore, some investigation into using ANNs to predict BWR core parameters would be warranted.

In the broader scheme of designing core LPs, the development of an ANN to generate core parameters is just the first step. The development of a core reload system (COS) which would automate the process of finding a new core reload pattern is the ultimate goal. The proposed COS would employ a second ANN, or some other optimization method, which would be trained to re-order the core to maximize a given parameter or parameters. The optimization method that is eventually chosen would use the core parameter predictor, developed as a result of this work, to evaluate the core reload patterns that it generates.

BIBLIOGRAPHY

[1] E.B. Bartlett. "Nuclear Power Plant Status Diagnostics Using Simulated Condensation: An Auto-Adaptive Learning Technique." Ph.D. Dissertation, University of Tennessee at Knoxville. (1990).

[2] A. Basu. "Nuclear power plant status diagnostics using a neural network with dynamic node architecture." M.S. Thesis, Iowa State University, (1991).

[3] M. Caudill. "Neural Networks Primer, Part 1." AI Expert 6 (Dec, 1987): 46-52.

[4] M. Caudill. "Neural Networks Primer, Part 2." AI Expert 7 (Feb, 1988): 55-61.

[5] M. Caudill. "Neural Networks Primer, Part 3." AI Expert 7 (June, 1988): 53-59.

[6] Y.A. Chao, C.W. Hu and C.A. Suo. "A Theory of Fuel Management via Backward Calculation." Nucl. Sci. Eng. 93 (1986): 78-87.

[7] J. Colletti, S.H. Levine, and J.B. Lewis. "Iterative Solution to the Optimal Poison Management Problem in Pressurized Water Reactors" Nucl. Technol. 63 (Dec, 1983): 415-425.

[8] "CITATION Code Manual." National Energy Software Center, Argonne National Laboratory (Oct, 1971).

[9] W.J. Freeman. "The Physiology of Perception." Scientific American (Feb, 1991): 78-85.

[10] A. Galperin, S. Kimhi and M. Segev. "A Knowledge-Based System for Optimization of Fuel Reload Configurations." Nucl. Sci. Eng. 102 (1989): 43-53.

[11] R. Hecht-Nielsen. Neurocomputing; Addison-Wesley Publishing Company: Reading, Massachusetts, (1990).

[12] R. Hecht-Nielsen. "Counterpropagation networks." Proc. of the Int. Conf. on Neural Networks II, IEEE Press, New York (June 1987): 19-32.

[13] J. Hertz, A. Krogh and R.G. Palmer. Introduction to the Theory of Neural Computation; Addison-Wesley Publishing Company: Reading, Massachusetts, (1991).

[14] H.G. Kim, S.H. Chang and B.H. Lee. "Pressurized Water Reactor Core Parameter Prediction Using an Artificial Neural Network." Nucl. Sci. Eng. 113 (1993): 70-76.

[15] H.G. Kim, S.H. Chang and B.H. Lee. "Optimal Fuel Loading Pattern Design Using an Artificial Neural Network and a Fuzzy Rule-Based System." Nucl. Sci. Eng. 115 (1993): 152-163.

[16] Y.J. Kim, T.J. Downar and A. Sesonske. "Optimization of Core Reload Design for Low-Leakage Fuel Management in Pressurized Water Reactors." Nucl. Sci. Eng. 96 (1987): 85-101.

[17] M.A. Kramer and J.A. Leonard. "Diagnosis Using Backpropogation Neural Networks - Analysis and Criticism." Computers chem. Engng. 14 12 (1990): 1323-1338.

[18] D.J. Kropaczek and P.J. Turinsky. "In-Core Nuclear Fuel Management for Pressurized Water Reactors Utilizing Simulated Annealing." Nucl. Technol. 95 (1991): 9-32.

[19] J.R. LaMarsh. Introduction to Nuclear Engineering, 2nd Edition; Addison-Wesley Publishing Company: Reading, Massachusetts, (1983).

[20] R. Lippmann. "An Introduction to Computing with Neural Nets.", IEEE Acoustics Speech and Signal Processing Magazine 4 (Apr, 1987): 4-22.

[21] G.I. Maldonado. "Non-Linear Nodal Generalized Perturbation Theory Within the Framework of PWR In-Core Nuclear Fuel Management Optimization." Ph.D. Dissertation, North Carolina State University, (1993).

[22] G.I. Maldonado. Personal communication. Iowa State University, Ames, Iowa, (July 7, 1994).

[23] H. Motoda. "Optimal Control Rod Programming of Light Water Reactors in Equilibrium Fuel Cycle" Nucl. Sci. Eng. 46 (1971): 88-111.

[24] H. Motoda. "Optimization of Control Rod Programming and Loading Poison in a Multiregion Nuclear Reactor by the Method of Approximation Programming" Nucl. Sci. Eng. 49 (1972): 515-524.

[25] K.C. Okafor and T. Aldemir. "Construction of Linear Empirical Core Models for Pressurized Water Reactor In-Core Fuel Management." Nucl. Technol. 81 (1988): 381-392.

[26] G.T. Parks. "An Intelligent Stochastic Optimization Routine for Nuclear Fuel Cycle Design." Nucl. Technol. 233 (1990): 233-246.

[27] C.J. Pfeifer. "PDQ-7 Reference Manual II." WAPD-TM-947(L), Bettis Atomic Power Laboratory (June, 1972).

[28] K. Sekimizu. "Optimization of In-Core Fuel Management and Control Rod Strategy in Equilibrium Fuel Cycle" J. Nucl. Sci. Technol. 12 5 (May, 1975): 287-296.

[29] "SIMULATE-YA Code Manual." YAEC-1518, Yankee Atomic Energy Corporation (Nov, 1985).

[30] D.A. Sprecher. "On the structure of continuous functions of several variables." Trans. Am. Math. Soc. 115 (Mar, 1965): 340-355.

[31] J.G. Stevens, K.S. Smith and T.J. Downar. "The COUNTM Suite and SOA1 Loading Pattern Database." Studsvik of America, Idaho Falls, Idaho, (June, 1992).

[32] J.G. Stevens, K.S. Smith, K.R. Rempe and T.J. Downar. "Optimization of PWR Shuffling by Simulated Annealing with Heuristics." Proc. of the American Nuclear Society I, (Apr, 1994).

[33] J.G. Stevens. Personal communication. Studsvik of America, Idaho Falls, Idaho, (June 9, 1994).

[34] M. Takeda and J. W. Goodman. "Neural Networks for Computation: Number Representations and Programming Complexity." Appl. Optics 25, 18 (1986): 3033.

[35] P.J. Werbos. "Backpropagation Through Time: What It Does and How to Do It." Proceedings of the IEEE 79.10 (Oct, 1990): 1550-1560.

[36] B. Widrow and M.A. Lehr. "30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation." Proceedings of the IEEE 78.9 (Sept, 1990): 1415-1441.

[37] Merriam-Webster Inc.. Webster's Ninth New Collegiate Dictionary; Publishers: Springfield, Massachusetts, (1985).

APPENDIX A. COMPUTER CODES

This chapter contains the computer codes used in this work. The back propagation program, called *ann1*, is made up of three files: *main1.c*, *io1.c* and *bckprp1.c*. The program is initiated by *main1.c* which calls various routines from *io1.c* and *bckprp1.c*. The routines in *io1.c* are mostly input and output routines which handle reading in values which control the execution of the program, the network input and output, and the weights. The routines in *bckprp1.c* deal mainly with the various stages of the back propagation algorithm, such as: feed forward, back propagation of errors and change of weights.

MAIN1.C

```
/*              * * *  BACK PROPAGATION NEURAL NETWORK  * * *
                * * *  WRITTEN BY:   SCOTT E. WENDT     * * *
                * * *  IOWA STATE UNIVERSITY, AMES, IA  * * *

*****       This is the main program from which control is
            transferred to the appropriate subroutines.  The
            file 'net.inp' contains all the important parameters
            required for execution and is read almost
            immediately.
***** */

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

```
#include <string.h>
#include "ann1.h"  /* Contains dimension info for arrays   */

int      KASE,       /* Number of patterns int training set  */
         mode;       /* Training or recall mode              */
float    rms = 0.0,  /* Accumulator of RMS err per data set  */
         StpRMS;     /* Target RMS value               .INP  */
long int
         SAFE,       /* # of iters between saves       .INP  */
         count = 0;  /* # of iters                           */

main()
{
 extern void init1(), init2(), input(), initerr(), fdfwd(),
             deltarule(), backprop(), sdiff(), tstsav(),
             RMS(), outp(), cnt();

 int i;
 time_t t;

srand((unsigned) time(&t)); /* Randomize using system clock */
init1();                    /* Read "net.inp"               */
init2();                    /* Read wts file                */

switch (mode) {
 case 0: { /* ============== Train wts. ================== */
  input(0);                /* Read input and answers       */
  initerr();               /* Init. error terms            */
  do {                     /* Main program loop            */
    count++;               /* Incr counter for wgt sav     */
    deltarule();           /* Back propagate the errors    */
    initerr();               /* Init. error terms          */
    for (i = 0; i < KASE; i++) { /* Loop thru training set */
      fdfwd(i);            /* Process hidden/output layers */
      backprop(i);         /* Calculate the errors         */
      }                    /* i loop - KASE                */
    RMS();                 /* Function to calc RMS error   */
    tstsav();              /* Test if time to save wgts    */
    } while (rms > StpRMS); /* End main program loop       */
  count = SAFE;            /* Force program to save wgts   */
  tstsav();                /* Test if time to save wgts    */
  break;
  } /* End case 1 */
 case 1: { /* ======= Recall wts on validation set ======= */
  input(1);                /* Read input and answers       */
  for (i = 0; i < KASE; i++) {   /* Loop thru training set */
    fdfwd(i);              /* Process hidden/output layers */
    sdiff(i);              /* Calculate the errors         */
    }                      /* i loop - KASE                */
  RMS();                   /* Function to calc RMS error   */
```

```
  printf("\nRMS err on the recall set = %f; %5d patterns.\n",
                                                    rms,KASE);
  outp(1);                  /* Write .out file                */
  cnt();                    /* Count patterns by error        */
  break;
  } /* End case 1 */
 case 2: { /* ====== Recall wts on unknown data set ====== */
  input(2);                 /* Read input and answers         */
  for (i = 0; i < KASE; i++) {  /* Loop thru training set     */
    fdfwd(i);               /* Process hidden/output layers   */
    }                       /* i loop - KASE                  */
  outp(2);                  /* Write .out file                */
  break;
  } /* End case 2 */
 }  /* End switch */
return;
}
```

IO1.C

```
*****     This part of the program delas mostly with input and
          output functions, such as: reading weight and data
          files and writing new weights to a file.
***** */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "ann1.h"

FILE *textfile1, *textfile2,
     *textfile3;              /* Pointer to file being used   */
char
     inbuf[50],               /* Input buffer                 */
     fname1[8],fname2[8],     /* Data file names              */
     wname1[16],wname2[16],   /* Weight file names            */
     name[16];                /* 4 letter prefix & dummy      */
int
     nodes[MXLYRS],           /* Number of nodes per layer    */
     MX,                      /* One less than MXLYRS         */
     save,                    /* Save when 1                  */
     sav = 1,                 /* flag for wt file to use      */
     prn,                     /* Scrn print: 0=no, 1=yes .INP */
     indx[MXHNDES],           /* Index array for hidden node  */
     mxnds[4];                /* No of nodes arrays can hold  */
float
     x1[MXKASE][MXINDES],     /* Input array - read from file */
     w2[MXHNDES][MXINDES],    /* Hidden layer weights         */
     w3[MXONDES][MXHNDES],    /* Output layer weights         */
     L2[MXHNDES],              /* Bias term for second layer  */
     L3[MXONDES],              /* Bias term for third layer   */
     x3[MXKASE][MXONDES],      /* Output layer                */
     ans[MXKASE][MXONDES],     /* Correct answer from file    */
     beta2, beta3,             /* Learning rates - wts        */
     bl2, bl3,                 /* Learning rates - biass      */
     alpha,                    /* momentum term               */
     savrms = 2.,              /* Best RMS err - to be saved  */
     oldrms = 2.,              /* RMS err from prev iter      */
     oldrms2 = 2.,             /* RMS err prior to prev iter  */
     delrms = 0.0,             /* Accum diff btwn rms/oldrms  */
     pi = 3.1415926535;        /* pi                          */
extern int
     KASE,                     /* No. of patterns             */
     mode;                     /* training or recall mode .INP*/
extern float
     rms,                      /* Accum RMS err per data set  */
     StpRMS;                   /* Target RMS value       .INP */
extern long int
```

```
    SAFE,                    /* # of iters between saves.INP*/
    count;                   /* # of iters                  */

/* ---------------------------------------------------------
   Read "net.inp" file for current instructions
   --------------------------------------------------------- */
void init1()
{
int i,dum,lyrs;

  MX = MXLYRS - 1;
  mxnds[0] = MXINDES;
  mxnds[1] = MXHNDES;
  mxnds[MX] = MXONDES;

  if ((textfile3 = fopen("net.inp", "r")) == NULL ) {
     printf("net.inp not found\n");
     return;
     }

  fscanf(textfile3, "%s %s\n", fname1, inbuf);
  fscanf(textfile3, "%d %s\n", &KASE, inbuf);
  fscanf(textfile3, "%s %s\n", fname2, inbuf);
  fscanf(textfile3, "%d %s\n", &lyrs, inbuf);
  for (i = 0; i < lyrs; i++)
    fscanf(textfile3, "%d ", &nodes[i]);
  fscanf(textfile3, "%s\n", inbuf);
  fscanf(textfile3, "%d %s\n", &mode, inbuf);
  fscanf(textfile3, "%d %s\n", &SAFE, inbuf);
  fscanf(textfile3, "%d %s\n", &prn, inbuf);
  fscanf(textfile3, "%f %s\n", &beta2, inbuf);
  fscanf(textfile3, "%f %s\n", &beta3, inbuf);
  fscanf(textfile3, "%f %s\n", &alpha, inbuf);
  fscanf(textfile3, "%f %s\n", &StpRMS, inbuf);
  fscanf(textfile3, "%d %s\n",  &dum, inbuf);
  fscanf(textfile3, "%d %s\n",  &dum, inbuf);
  fclose(textfile3);

  beta2 /= (float)KASE;
  beta3 /= (float)KASE;
  bl2 = beta2/(float)nodes[0];
  bl3 = beta3/(float)nodes[1];
  if (lyrs > MXLYRS) {
    printf("ERROR: Specified no. of layers
            exceeds array formatting.\n");
    exit(0);
    }
  for (i = 0; i < lyrs; i++)
    if (nodes[i] > mxnds[i]) {
      printf("ERROR: Spec. # of nodes in layer
```

```
                %d exceeds array formatting.\n",i);
        exit(0);
        }
  return;
}

/* ------------------------------------------------------
   Initialize weights - random or from previous file
   ------------------------------------------------------ */
void init2()
{
int i,j,dum;
int nds[MXLYRS], nlyrs;

  strcpy(wname1,fname2);
  strcpy(wname2,fname2);
  strcat(wname2,".bak");
  strcat(wname1, ".wts");
  strcat(wname2, ".wts");

/* Generate new random wts  */

  for (i = 0; i < nodes[1]; i++) {
    for (j = 0; j < nodes[0]; j++)
      w2[i][j] = ((float)rand()/(float)RAND_MAX - 0.5);
    L2[i] = ((float)rand()/(float)RAND_MAX - 0.5);
    }
  for (i = 0; i < nodes[MX]; i++) {
    for (j = 0; j < nodes[1]; j++)
      w3[i][j] = ((float)rand()/(float)RAND_MAX - 0.5);
    L3[i] = ((float)rand()/(float)RAND_MAX - 0.5);
    }

/* Check to see if "name".wts exists from a previous run.  */
  strcpy(name,fname2);
  strcat(name, ".wts");
  if ((textfile2 = fopen( name, "r")) == NULL ) {
    printf("%s not found\n", name);
    return;
    }

/* File exists so read in previous values */

  fscanf(textfile2, "%f %s\n", &savrms, inbuf);
  fscanf(textfile2, "%d %s\n", &nlyrs, inbuf);
  for (i = 0; i < nlyrs; i++)
    fscanf(textfile2, "%d ", &nds[i]);
  fscanf(textfile2, "%s\n", inbuf);
  oldrms = savrms;
  oldrms2 = savrms;
```

```
  if (nlyrs > MXLYRS) {
    printf("ERROR: Specified no. of layers
            exceeds array formatting.\n");
    exit(0);
    }
  for (i = 0; i < nlyrs; i++)
    if (nds[i] > mxnds[i]) {
      printf("ERROR: Spec. # of nodes in layer
              %d exceeds array formatting.\n",i);
      exit(0);
      }
  for (i = 0; i < nlyrs; i++)
    if (nds[i] > nodes[i]) {
      printf("WARNING: Spec. # of nodes in layer
              %d exceeds previous nodes.\n",i);
      printf("Continue anyway?  Enter 1 for yes.  \n");
      gets(inbuf);
      sscanf(inbuf,"%d",&dum);
      if (dum != 1) exit(0);
      }

  for (i = 0; i < nds[1]; i++) {
    for (j = 0; j < nds[0]; j++)
      fscanf(textfile2, "%f %s\n",&w2[i][j], inbuf);
    fscanf(textfile2, "%f %s\n",&L2[i], inbuf);
    }
  for (i = 0; i < nds[MX]; i++) {
    for (j = 0; j < nds[1]; j++)
      fscanf(textfile2, "%f %s\n",&w3[i][j], inbuf);
    fscanf(textfile2, "%f %s\n",&L3[i], inbuf);
    }
  fclose(textfile2);

  return;
}

/* ------------------------------------------------------
      Read input values and answers
   ------------------------------------------------------ */
void input(int tst)
{
int i, j, dum;

  strcpy(name,fname1);
  strcat(name, ".dat");
  if ((textfile1 = fopen(name, "r")) == NULL ) {
     printf("%s not found\n", name);
     return;
     }
```

```
switch (tst) {
 case 0:
 case 1: {
  for (i = 0; i < KASE; i++) {
    for (j = 0; j < nodes[0]; j++)
      fscanf(textfile1, "%f ", &x1[i][j]);
    for (j = 0; j < nodes[MX]; j++)
      fscanf(textfile1, "%f ", &ans[i][j]);
    }
  break;
  }
 case 2: {
  for (i = 0; i < KASE; i++) {
    for (j = 0; j < nodes[0]; j++)
      fscanf(textfile1, "%f ", &x1[i][j]);
    }
  }
 }
 fclose(textfile1);

/* Init index array */
  for (j = 0; j < MXHNDES; j++)
    indx[j] = j;

  return;
}

/* ------------------------------------------------------------
  Test if rms error is a min. and SAFE iterations have passed
  ------------------------------------------------------------ */
void tstsav()
{
extern void wgtsav(), imp();

  if (prn == 1) {
    printf("%10.8f,%10.8f\n",rms, delrms);
    }
  if (count % SAFE == 0)
     save = 1;
  if (rms <= savrms && save == 1) {
    wgtsav();
    save = 0;
    count = 0;
    savrms = rms;
    }
  oldrms2 = oldrms;
  oldrms = rms;

  return;
}
```

```
/* ------------------------------------------------------
     Save current network values
   ------------------------------------------------------ */
void wgtsav()
{
int i,j;

/* Open file, testing for success */
  if (sav == 1) {
    if ((textfile2 = fopen(wname1, "w")) == NULL ) {
     printf("Error opening %s for writing\n", wname1);
     exit(0);
     }
    sav = 2;
    }
  else {
    if ((textfile2 = fopen(wname2, "w")) == NULL ) {
     printf("Error opening %s for writing\n", wname2);
     exit(0);
     }
    sav = 1;
    }

  fprintf(textfile2, "%f\t$SavRMS\n", rms);
  fprintf(textfile2, "%d\t\t$No_layers\n", MXLYRS);
  for (i = 0; i < MXLYRS; i++)
    fprintf(textfile2, "%2d ", nodes[i]);
  fprintf(textfile2, "\t$Inodes_Hnodes_Onodes\n");

  for (i = 0; i < nodes[1]; i++) {
    for (j = 0; j < nodes[0]; j++)
      fprintf(textfile2, "%f\t$w2[%d][%d]\n",
                         w2[indx[i]][j],indx[i],j);
    fprintf(textfile2, "%f\t$L2[%d]\n",L2[indx[i]],indx[i]);
    }
  for (i = 0; i < nodes[MX]; i++) {
    for (j = 0; j < nodes[1]; j++)
      fprintf(textfile2, "%f\t$w3[%d][%d]\n",
                         w3[i][indx[j]],i,indx[j]);
    fprintf(textfile2, "%f\t$L3[%d]\n",L3[i],i);
    }

  fclose(textfile2);

  return;
}

/* ------------------------------------------------------
     Write out output
   ------------------------------------------------------ */
```

```
void outp(int tst)
{
int i,j;

/* Open file, testing for success */
  strcpy(name,fname1);
  strcat(name, ".out");
  if ((textfile2 = fopen(name, "w")) == NULL ) {
     printf("%s not found\n", name);
     return;
     }

switch (tst) {
 case 1: {
  fprintf(textfile2, "RMS error on recall %f\n",rms);
  for (j = 0; j < nodes[MX]; j++)
    fprintf(textfile2, "  Ans[%2d]   Out[%2d] ",j,j);
  fprintf(textfile2, "\n");
  for (i = 0; i < KASE; i++) {
    for (j = 0; j < nodes[MX]; j++)
      fprintf(textfile2, "%f, %f, ",ans[i][j],x3[i][j]);
    fprintf(textfile2, "\n");
    }
  break;
  }
 case 2: {
  for (j = 0; j < nodes[MX]; j++)
    fprintf(textfile2, "  Out[%2d] ",j);
  fprintf(textfile2, "\n");
  for (i = 0; i < KASE; i++) {
    for (j = 0; j < nodes[MX]; j++)
      fprintf(textfile2, "%f, ",x3[i][j]);
    fprintf(textfile2, "\n");
    }
  }
 }
 fclose(textfile2);

  return;
}

/* ------------------------------------------------------
     Counts number of patterns in certain categories
   ------------------------------------------------------ */
void cnt()
{
int i,j,COUNT[6];
float limit[7], dum;
float ABS(float);
```

```
  limit[0] = 0.00;  /* Set limits on categories */
  limit[1] = 0.001;
  limit[2] = 0.005;
  limit[3] = 0.01;
  limit[4] = 0.05;
  limit[5] = 0.1;
  limit[6] = 0.1;
  for (i = 0; i < 6; i++)
    COUNT[i] = 0;

  for (i = 0; i < KASE; i++)
    for (j = 0; j < nodes[MX]; j++)  {
      dum = ABS(ans[i][j]-x3[i][j]);
      if (dum < limit[1])
        COUNT[0]++;
      else if (dum < limit[2])
        COUNT[1]++;
      else if (dum < limit[3])
        COUNT[2]++;
      else if (dum < limit[4])
        COUNT[3]++;
      else if (dum < limit[5])
        COUNT[4]++;
      else {
        COUNT[5]++;
        if (dum > limit[6])
          limit[6] = dum;
        }
      }

  printf("Group\t    Error Range\t\t Number\t  Percent \n");
  for (j = 0; j < 6; j++)
    printf("  %d\t %5.3f < x < %5.3f\t %5d\t  %5.2f%%\n",
            j+1,limit[j],limit[j+1],COUNT[j],
            (float)COUNT[j]/(float)KASE*100.);

  return;
}

/* ---------------------------------------------------------
     Returns absolute value
   --------------------------------------------------------- */
float ABS(float x)
{

  if (x < 0.0)
     return ( -x );
  else
     return ( x );
}
```

BCKPRP1.C

```
*****     This part of the program contains the 'meat' of the
          ANN - the feed forward, back propagation and weight
          change routines.
***** */


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include "ann1.h"

float
      i2[MXKASE][MXHNDES],       /* Hidden layer inputs       */
      x2[MXKASE][MXHNDES],       /* Hidden layer outputs      */
      i3[MXKASE][MXONDES],       /* Output layer inputs       */
      x3[MXKASE][MXONDES],       /* Output layer outputs      */
      e2[MXHNDES][MXINDES],      /* Error for hidden nodes    */
      e3[MXONDES][MXHNDES],      /* Error for output nodes    */
      eL2[MXHNDES],              /* Error for hid lyr bias    */
      eL3[MXONDES],              /* Error for out lyr bias    */
      delta2[MXHNDES][MXINDES], /* Momntm for hid nodes      */
      delta3[MXONDES][MXHNDES]; /* Momntm for out nodes      */
extern int
      KASE,                      /* No. of patterns           */
      nodes[MXLYRS],             /* No. of nodes per layer    */
      MX,                        /* One less than MXLYRS      */
      indx[MXHNDES];             /* Used to order nodes       */
extern float
      x1[MXKASE][MXINDES],       /* Input array - from file   */
      w2[MXHNDES][MXINDES],      /* Hidden layer weights      */
      w3[MXONDES][MXHNDES],      /* Output layer weights      */
      L2[MXHNDES],               /* Bias for hid nodes        */
      L3[MXONDES],               /* Bias for out nodes        */
      ans[MXKASE][MXONDES],      /* Answer from file          */
      beta2, beta3,              /* Learning rates - wts      */
      bl2, bl3,                  /* Learning rates - bias     */
      alpha,                     /* momentum term             */
      rms,                       /* Accum of RMS for data set*/
      delrms,                    /* Accum diff rms & oldrms   */
      oldrms,                    /* RMS from prev iter        */
      pi;                        /* pi                        */


/* ------------------------------------------------------
   Initialize error terms
   ------------------------------------------------------ */
```

```
void initerr()
{
int i, j;

  for (i = 0; i < nodes[1]; i++) {
    for (j = 0; j < nodes[0]; j++)
      e2[i][j] = 0.0;
    eL2[i] = 0.0;
    }
  for (i = 0; i < nodes[MX]; i++) {
    for (j = 0; j < nodes[1]; j++)
      e3[i][j] = 0.0;
    eL3[i] = 0.0;
    }
  rms = 0.0;

  return;
}

/* ------------------------------------------------------
      Hidden and output layer feed-forward
   ------------------------------------------------------ */
void fdfwd(int set)
{
int i,j;
float sigmd(float);

  for (i = 0; i < nodes[1]; i++) {
    i2[set][i] = 0.0;
    for (j = 0; j < nodes[0]; j++)
      i2[set][i] += w2[i][j]*x1[set][j];
    i2[set][i] -= L2[i];
    x2[set][i] = sigmd(i2[set][i]);
    }

  for (i = 0; i < nodes[MX]; i++) {
    i3[set][i] = 0.0;
    for (j = 0; j < nodes[1]; j++)
      i3[set][i] += w3[i][j]*x2[set][j];
    i3[set][i] -= L3[i];
    x3[set][i] = sigmd(i3[set][i]);
    }

  return;
}

/* ------------------------------------------------------
      Back propagate the errors
   ------------------------------------------------------ */
void backprop(int set)
```

```
{
float deriv(float);
float sum, d[MXONDES], dum;
int i, j;

  for (i = 0; i < nodes[MX]; i++) {
    dum = ans[set][i] - x3[set][i];
    rms += dum*dum;
    d[i] = deriv(i3[set][i])*dum;
    for (j = 0; j < nodes[1]; j++)
      e3[i][j] += d[i] * x2[set][j];
    eL3[i] += d[i] * i3[set][i];
    }

  for (i = 0; i < nodes[1]; i++) {
    sum = 0.0;
    for (j = 0; j < nodes[MX]; j++)
      sum += w3[j][i]*d[j];
    dum = sum * deriv(i2[set][i]);
    for (j = 0; j < nodes[0]; j++)
      e2[i][j] += dum * x1[set][j];
    eL2[i] += dum * i2[set][i];
    }

  return;
}

/* ------------------------------------------------------
      Calc square of the error
   ------------------------------------------------------ */
void sdiff(int set)
{
float dum;
int i;

  for (i = 0; i < nodes[MX]; i++) {
    dum = ans[set][i] - x3[set][i];
    rms += dum*dum;
    }

  return;
}

/* ------------------------------------------------------
      Adjust wgts using delta rule
   ------------------------------------------------------ */
void deltarule()
{
float dummy;
int i, j;
```

```
  for (i = 0; i < nodes[MX]; i++) {
    for (j = 0; j < nodes[1]; j++) {
      dummy = w3[i][j];
      if (delrms > 0.0)
     w3[i][j] += (beta3*e3[i][j] + alpha*delta3[i][j]);
      else
     w3[i][j] += (beta3*e3[i][j]);
      delta3[i][j] = w3[i][j] - dummy;
      }
    L3[i] += bl3 * eL3[i];
    }
  for (i = 0; i < nodes[1]; i++) {
    for (j = 0; j < nodes[0]; j++) {
      dummy = w2[i][j];
      if (delrms > 0.0)
     w2[i][j] += (beta2*e2[i][j] + alpha*delta2[i][j]);
      else
     w2[i][j] += (beta2*e2[i][j] + alpha*delta2[i][j]);
      delta2[i][j] = w2[i][j] - dummy;
      }
    L2[i] += bl2 * eL2[i];
    }
  return;
}

/* ------------------------------------------------------
     Function 1 - Sigmoid function
   ------------------------------------------------------ */
float sigmd(float mu)
{
  return(atan(mu)/pi + 0.5);
}



/* ------------------------------------------------------
     Function 2 - Inverse function for backprop
   ------------------------------------------------------ */
float deriv(float mu)
{
  return(1.0/(1.0+mu*mu));
}



/* ------------------------------------------------------
     Find RMS, delrms  and delrms2
   ------------------------------------------------------ */
void RMS()
{
```

```
  rms = (float)pow(rms/(double)KASE/(double)nodes[MX],0.5);
  delrms = oldrms-rms;

  return;
}
```

ANN1.H

```
*****     This is the header file for the ANN.  It contains
          information on array subscripting.
***** */

#define MXKASE 3100    /* max # of training sets/files       */
#define MXLYRS 3       /* max # of layers                    */
#define MXINDES 26     /* max # of input layer nodes         */
#define MXHNDES 26     /* max # of hidden layer nodes        */
#define MXONDES 26     /* max # of output layer nodes        */
#define MXNODES 26
```

NET.INP

```
*****     This is the input file for the ANN.  It contains all
          pertinent information on the execution of ANN1.
***** */

name1    $Prefix_for_data_file
8        $No_of_patterns_to_read
name2    $Prefix_for_wts_files
3        $No_of_layers
3 3 8    $No_of_input_hidden_output_layers
0        $Training,_validation_or_unknown_(0,1,2)
500      $Iterations_btwn_saves
1        $Print_to_screen_(0=no,1=yes)
.5       $Hidden_layer_learning_rate
.5       $Output_layer_learning_rate
.1       $Momentum_learning_rate
.01      $Stopping_RMS
0        $Undefined
0        $Undefined
0        $Undefined
0        $Undefined
0        $Undefined
```

APPENDIX B. SAMPLE DATA FILES

This chapter contains a sample of the data from the SOA1 database and the normalized and abbreviated data files which were created from the database. Ten lines from the SOA1 database appear in ten.dat. The data files created from the SOA1 database are: boron10.dat and pin10.dat.

TEN.DAT - sample from SOA1 database.

```
121324122424242142332242 23  .194917E+04  .328458E+01
.145279E+04 .999993 1.774000 1.089113 811.  .105580E+04
1.000020 1.6140001.074097 811.  .466003E+03 1.000001 1.433000
1.055167 811. -.694324E+02 1.000004 1.326000 1.041435 811.
.367388E+02  .384481E+02  .311928E+02  .361563E+02
.162734E+02  .227378E+02  .185455E+02  .212195E+02

121324221424242242421233 23  .190085E+04  .255812E+01
.140061E+04 .999996 1.823000 1.088243 811.  .101469E+04
1.000003 1.637000 1.074664 811.  .443510E+03 1.000007 1.439000
1.057877 811. -.852874E+02 1.000005 1.330000 1.043800 811.
.370418E+02  .379102E+02  .314673E+02  .363399E+02
.146373E+02  .229464E+02  .190479E+02  .212648E+02

121421321423243242422423 22  .192821E+04  .293691E+01
.142241E+04 .999994 1.808000 1.100316 912.  .102115E+04
1.000002 1.619000 1.083774 912.  .441953E+03 1.000006 1.422000
1.063972 912. -.863070E+02 1.000005 1.319000 1.034393 1011.
.370379E+02  .376806E+02  .316241E+02  .362784E+02
.143405E+02  .225466E+02  .189813E+02  .215323E+02

121422132424242242432223 13  .183408E+04  .206408E+01
.132532E+04 1.000000 1.545000 1.078865 910.  .959150E+03
1.000005 1.473000 1.053214 910.  .408913E+03 1.000007 1.387000
1.043862 811. -.111558E+03 1.000002 1.311000 1.039119 912.
.354271E+02  .387097E+02  .327194E+02  .364472E+02
.112912E+02  .146098E+02  .196054E+02  .216102E+02
```

```
121423212423242142422243 23  .197695E+04  .345457E+01
.146764E+04 .999998 1.888000 1.098934 912.  .104099E+04
1.000000 1.664000 1.080372 912.  .447003E+03 1.000002 1.429000
1.053242 912. -.837359E+02 1.000002 1.315000 1.036031 912.
.349685E+02  .386034E+02  .319949E+02  .369268E+02
.156200E+02  .230817E+02  .181719E+02  .214939E+02
```

BORON10.DAT - normalized data for boron concentration.

```
0.400000 0.200000 0.400000 0.200000 0.800000 0.200000 0.600000
0.400000 0.800000 0.400000 0.800000 0.400000 0.600000 0.600000
0.400000 0.800000 0.400000 0.800000 0.400000 0.400000 0.400000
0.800000 0.400000 0.400000 0.600000 0.238887

0.400000 0.200000 0.400000 0.200000 0.800000 0.400000 0.200000
0.400000 0.800000 0.400000 0.800000 0.400000 0.800000 0.600000
0.400000 0.600000 0.400000 0.800000 0.400000 0.400000 0.400000
0.800000 0.400000 0.600000 0.600000 0.418360

0.400000 0.200000 0.400000 0.200000 0.800000 0.400000 0.400000
0.400000 0.800000 0.400000 0.800000 0.400000 0.400000 0.600000
0.400000 0.800000 0.400000 0.800000 0.600000 0.200000 0.400000
0.800000 0.400000 0.600000 0.600000 0.226818

0.400000 0.200000 0.400000 0.200000 0.800000 0.400000 0.600000
0.200000 0.800000 0.400000 0.400000 0.400000 0.800000 0.400000
0.400000 0.800000 0.400000 0.800000 0.600000 0.400000 0.800000
0.400000 0.600000 0.400000 0.600000 0.190077

0.400000 0.200000 0.400000 0.200000 0.800000 0.400000 0.600000
0.400000 0.400000 0.400000 0.800000 0.400000 0.800000 0.600000
0.400000 0.800000 0.400000 0.800000 0.400000 0.400000 0.400000
0.800000 0.600000 0.200000 0.600000 0.137289

0.400000 0.200000 0.400000 0.200000 0.800000 0.400000 0.600000
0.400000 0.800000 0.400000 0.800000 0.400000 0.400000 0.600000
0.400000 0.800000 0.400000 0.800000 0.400000 0.200000 0.400000
0.800000 0.600000 0.400000 0.600000 0.157181

0.400000 0.200000 0.400000 0.200000 0.800000 0.400000 0.600000
0.400000 0.800000 0.400000 0.800000 0.400000 0.800000 0.400000
0.400000 0.800000 0.400000 0.800000 0.600000 0.400000 0.600000
0.400000 0.400000 0.200000 0.600000 0.349818

0.400000 0.200000 0.400000 0.200000 0.800000 0.400000 0.600000
0.400000 0.800000 0.400000 0.800000 0.400000 0.800000 0.600000
0.400000 0.800000 0.400000 0.800000 0.600000 0.400000 0.600000
0.400000 0.400000 0.200000 0.400000 0.339752

0.400000 0.200000 0.400000 0.400000 0.800000 0.200000 0.400000
```

```
0.400000 0.800000 0.400000 0.800000 0.400000 0.600000 0.600000
0.400000 0.800000 0.400000 0.800000 0.600000 0.200000 0.400000
0.800000 0.400000 0.600000 0.400000 0.306682

0.400000 0.200000 0.400000 0.400000 0.800000 0.200000 0.600000
0.200000 0.800000 0.400000 0.600000 0.400000 0.800000 0.400000
0.400000 0.800000 0.400000 0.800000 0.600000 0.400000 0.800000
0.400000 0.600000 0.400000 0.400000 0.401232
```

PIN10.DAT - normalized data for pin peaking.

```
0.400000 0.200000 0.400000 0.200000 0.800000 0.200000 0.600000
0.400000 0.800000 0.400000 0.800000 0.400000 0.600000 0.600000
0.400000 0.800000 0.400000 0.800000 0.400000 0.400000 0.400000
0.800000 0.400000 0.400000 0.600000 0.151531

0.400000 0.200000 0.400000 0.200000 0.800000 0.400000 0.200000
0.400000 0.800000 0.400000 0.800000 0.400000 0.800000 0.600000
0.400000 0.600000 0.400000 0.800000 0.400000 0.400000 0.400000
0.800000 0.400000 0.600000 0.600000 0.377086

0.400000 0.200000 0.400000 0.200000 0.800000 0.400000 0.400000
0.400000 0.800000 0.400000 0.800000 0.400000 0.400000 0.600000
0.400000 0.800000 0.400000 0.800000 0.600000 0.200000 0.400000
0.800000 0.400000 0.600000 0.600000 0.163358

0.400000 0.200000 0.400000 0.200000 0.800000 0.400000 0.600000
0.200000 0.800000 0.400000 0.400000 0.400000 0.800000 0.400000
0.400000 0.800000 0.400000 0.800000 0.600000 0.400000 0.800000
0.400000 0.600000 0.400000 0.600000 0.159134

0.400000 0.200000 0.400000 0.200000 0.800000 0.400000 0.600000
0.400000 0.400000 0.400000 0.800000 0.400000 0.800000 0.600000
0.400000 0.800000 0.400000 0.800000 0.400000 0.400000 0.400000
0.800000 0.600000 0.200000 0.600000 0.137592

0.400000 0.200000 0.400000 0.200000 0.800000 0.400000 0.600000
0.400000 0.800000 0.400000 0.800000 0.400000 0.400000 0.600000
0.400000 0.800000 0.400000 0.800000 0.400000 0.200000 0.400000
0.800000 0.600000 0.400000 0.600000 0.187434

0.400000 0.200000 0.400000 0.200000 0.800000 0.400000 0.600000
0.400000 0.800000 0.400000 0.800000 0.400000 0.800000 0.400000
0.400000 0.800000 0.400000 0.800000 0.600000 0.400000 0.600000
0.400000 0.400000 0.200000 0.600000 0.265153

0.400000 0.200000 0.400000 0.200000 0.800000 0.400000 0.600000
0.400000 0.800000 0.400000 0.800000 0.400000 0.800000 0.600000
0.400000 0.800000 0.400000 0.800000 0.600000 0.400000 0.600000
0.400000 0.400000 0.200000 0.400000 0.230095
```

0.400000 0.200000 0.400000 0.400000 0.800000 0.200000 0.400000
0.400000 0.800000 0.400000 0.800000 0.400000 0.600000 0.600000
0.400000 0.800000 0.400000 0.800000 0.600000 0.200000 0.400000
0.800000 0.400000 0.600000 0.400000 0.145618

0.400000 0.200000 0.400000 0.400000 0.800000 0.200000 0.600000
0.200000 0.800000 0.400000 0.600000 0.400000 0.800000 0.400000
0.400000 0.800000 0.400000 0.800000 0.600000 0.400000 0.800000
0.400000 0.600000 0.400000 0.400000 0.302323

APPENDIX C. APPROXIMATING k_{eff} FROM CRITICAL BORON CONCENTRATION

This chapter contains a description of how I converted my prediction accuracy in critical boron concentration to a prediction accuracy in k_{eff}.

The following rules-of-thumb for critical boron concentration and k_{eff} during a 12 month cycle were related to me by G.I. Maldonado[22]. The critical boron concentration is approximately 1000-1200 ppm at BOC and is roughly 0 ppm at EOC. This corresponds to a percent change in boron concentration of 8.3% to 10% per month. The change in k_{eff} per month is roughly 0.01 which corresponds to a percent change in k_{eff} of 0.83% or 1.0%.

Based on the above comparisons of thumb rules, the conclusion is that a 3% accuracy rate in critical boron concentration is equivalent to a 0.3% accuracy rate in k_{eff}.