RESOURCE CONSTRAINED DESIGN OF ARTIFICIAL NEURAL NETWORKS

BY

HARISH KRIPLANI

B. Tech., Indian Institute of Technology, 1987

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1990

Urbana, Illinois

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

February 1990

WE HEREBY RECOMMEND THAT THE THESIS BY

Harish Kriplani

ENTITLED Resource Constrained Design of Artificial Neural Networks

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF Master of Science

Director of Thesis Research

Head of Department

Committee on Final Examination†

Chairperson

† Required for doctor's degree but not for master's.

ABSTRACT

In this thesis, the author addresses the problem of automating the design of artificial neural networks based on a gradient descent learning algorithm (the back error propagation algorithm) for solving a given application. There are many possible network configurations for solving the application. Training all of these configurations is impossible in any reasonable amount of time. In this thesis, the author presents a heuristic design method for selecting and training promising neural network configurations, with the goal of maximizing a given objective function of cost and training time, and satisfying given resource constraints. The proposed method consists of two parts. For a given amount of time divided into quanta, the first part of the method (called Type I experiments) systematically selects promising network configurations and trains them for various quanta of time. Unpromising configurations are pruned as the search proceeds. The second part of the method (called Type II experiments) refines the error measure used in Type I experiments. When the specified time limit is exceeded, a set of promising but possibly partially trained networks are reported. The method has been applied to a variety of benchmark problems collected from the connectionist mailing list (CMU) and the UCI repository of machine learning databases and found to be effective in identifying promising network configurations.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Benjamin Wah, for his patience, guidance and encouragement during the course of this thesis. I would also like to thank my colleagues Atef Alnajjar, Lon-Chan Chu, Pankaj Mehra and others for sitting through my long and frequent seminars and providing useful criticism. My special thanks to Srilata Raman for the discussion we had last summer. I would also like to acknowledge the help and support of my parents, without which this project may never have taken this shape.

Financial support by the National Science Foundation under grant MIP 88-10584 and the National Aeronautics and Space Administration under grant NCC 2-481 is gratefully acknowledged.

Finally, I would like to thank Donna Guzy for her secretarial help.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Back Error Propagation (BEP), or simply back-prop, is a powerful gradient descent training algorithm for artificial neural networks [19]. The algorithm has been successfully applied to a variety of problems [8, 13, 21]. The aim of the algorithm is to find a set of weights for a network which ensures that for each of the input patterns applied to the network, the output produced by the network is sufficiently close to the desired output pattern. The extent to which the actual output is to be close to the desired output is set by a heuristic stopping criterion. In the following presentation, we briefly discuss the algorithm and the type of network it can be applied to. Despite being a very popular algorithm, back-prop has many problems associated with it. Some of the problems encountered while training a network will be highlighted in this chapter. These problems become the motivation for taking up this study. In the following chapters, we describe a design methodology which remedies most of the problems mentioned.

1.1 The Neuron

A typical model of the neuron or the basic computational element in a neural network is shown in Figure 1.1. The current state of the neuron is known as its *activation*. Activation of a neuron is a function of the inputs that the neuron receives from other neurons (or the external world) and the weights associated with the incoming links. This relationship could be represented in any complex way and is decided heuristically for different applications. The following heuristic is widely reported in the literature:

$$Activation = a_k = \sum_{i=0}^{N-1} w_{ki} \times u_i + \theta_k \tag{1.1}$$

where N is the number of neurons from which the (k^{th}) neuron receives inputs, u_i, and w_{ki} is the weight

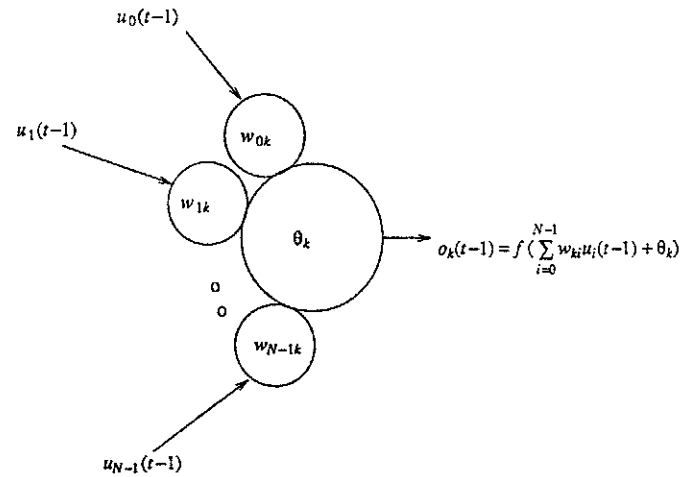

Figure 1.1: The basic computational element in a neural network.

associated with the link that connects the neuron to the i^{th} one. Each neuron is characterized by a number, known as its *threshold* or *bias* (θ_k), and a function, called the *activation function* or the *associated non-linearity*. This activation function maps the activation of the neuron to its output, as shown in the following equation:

$$Output = o_k = F(a_k) = F(\sum_{i=0}^{N-1} w_{ki} \times u_i + \theta_k) \tag{1.2}$$

A few typical activation functions are shown in Figure 1.2. The choice of the activation function is heuristically made. In this thesis, we have based our study on the following popular activation function, known as the sigmoidal non-linearity:

$$F(a) = \frac{1}{1 + e^{-a}} \tag{1.3}$$

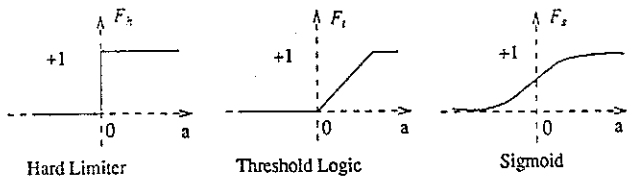

Figure 1.2: A few typical non-linearities used in neural networks.

1.2 The Neural Network.

The back propagation algorithm is applicable to any general neural network, but in this study, we confine ourselves to *layered feed-forward networks*. These networks have also come to be known as *back-prop networks*, after the algorithm. The networks have one layer of input units at the bottom and one layer of output units at the top. Between input and output layers, there might be one or more layers of hidden units. Connections are allowed only from units in lower layers to those in higher layers. Units within the same layer are not connected. A typical feed-forward network is shown in Figure 1.3.

1.3 The Training Algorithm

The back-prop algorithm is a supervised training algorithm that minimizes the total sum of squared errors (or TSSE) between the actual outputs produced by a network and the desired outputs by repeatedly adjusting the weights $w(t)$ by an amount proportional to the negative of the gradient of the error-weight surface. The TSSE is the squared error accumulated at all the output units over all the input-output training patterns, as given by the following equation.

$$TSSE = \sum_p \sum_j (t_{pj} - o_{pj})^2 \tag{1.4}$$

where t_{pj} and o_{pj} are the desired and the actual outputs of the j^{th} output unit, when the p^{th} training pattern is applied to the network. The back propagation algorithm adjusts weights in a network by the following

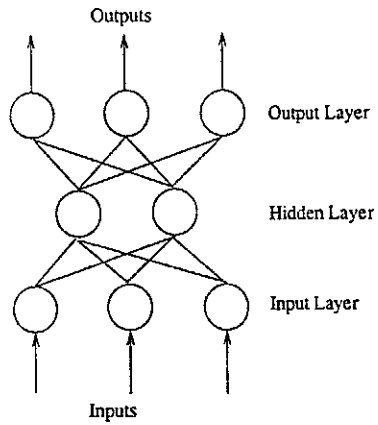

Figure 1.3: A typical feed forward network.

update equation.

$$\Delta w(t) = -\eta \frac{\partial TSSE(t)}{\partial w(t)} \tag{1.5}$$

where η is a constant, known as the *learning rate* constant.

The application of back-prop involves two phases. During the first phase, an input pattern is presented to and propagated forward through the network to calculate the output value for each output unit. During the second phase, error signals are passed backwards from the output units to the hidden units, and from the hidden units to the input units, and appropriate changes in weights are made to minimize TSSE. Details of the algorithm, including derivation of the important equations, can be found in a chapter by Rumelhart, Hinton, and Williams in [20].

Convergence properties of the BEP algorithm could be greatly improved by adding a momentum term to the weight-update equation. The new weight-update equation becomes

$$\Delta w(t) = -\eta \frac{\partial TSSE(t)}{\partial w(t)} + \alpha \Delta w(t-1) \tag{1.6}$$

where α is the *momentum constant*.

1.4 Problems Associated with Back-Prop

Back-prop is a heuristic training algorithm (Equations (1.1), (1.3) and (1.6) are heuristic in nature, for example). There are several problems associated with applying it to train networks. First, the algorithm does not indicate when to stop training. Various heuristic stopping criteria have been used in practice. A popular criterion, also used in this thesis, is to stop training when TSSE is below a specified threshold, *ecrit*.

The second problem with using BEP is that, before the algorithm can be applied, one needs to specify a network to be trained and assign values to a number of initial parameters [17]. The algorithm does not help guide the selection of the network or its initial weights. Due to a lack of theoretical results, one heuristically chooses a network and assigns initial weights to it. One popular heuristic for assigning initial weights is to choose random weights between *-wrange* and *+wrange*, where *wrange* is a parameter of the algorithm. The algorithm also requires various other training parameters, such as η and α, to be specified at the start. Again, these parameters are heuristically assigned some values. Further, a set of input-output patterns for training the network must be supplied. These patterns are application-dependent and are generally obtained heuristically by the users.

Back-prop is a heuristic gradient descent algorithm and, therefore, has all the problems associated with algorithms of this type. There is a great degree of uncertainty associated with using the algorithm for solving a given problem [24]. The algorithm may get stuck in local minima. It does not indicate when a local minimum is reached while training. Moreover, it does not specify how to get out of the local minimum, whenever one is reached. The training times obtained by applying the algorithm to various networks vary over several orders of magnitude. Even for the same network, training times may depend upon the initial weights assigned.

For consistency, we distinguish between a network and a configuration in this thesis. A *network* is a set of units with a defined interconnection. Weights on the interconnects need not be specified. In contrast, a *configuration* (or a *network configuration*) is defined as a network with a specified set of weights. Thus, for using the BEP algorithm, we need to specify a configuration and the training parameter values.

1.5 The Objective of the Study

The objective of this study is to develop a systematic design method which is executed under limited computational resources for automating the design of artificial neural networks using the BEP algorithm. The design of a neural network requires specification of the layout of the network and the final set of trained weights. The method aims at maximizing a user-specified objective function, which could be a function of the stopping criterion, the training time of the network, and the network complexity (cost). The method proposed should take into account various resource constraints in its execution. These constraints may include the time limit, memory-space limit, and architectural features, such as the number of parallel processors.

1.6 Assumptions

In this thesis, we assume that the objective function of the design is provided by the users and that it is a linear function of the parameters. These parameters are the readily measurable parameters of the network, at the end of its training. This assumption is justified because the effectiveness of the method can be measured only when its objective is known. The objective function used in the thesis is

$$OBJ = A - B \times TT - C \times Cost, \tag{1.7}$$

where A, B and C are problem-dependent constants specified by the users, and TT and $Cost$ are, respectively, the training time and the complexity of the configuration. We assume that the stopping criterion is the same for all the networks being considered and therefore it is represented by the constant A in the above equation. We fix a ceiling on the amount of time (TT_{norm}) that would be spent on a

network in the worst case. We would later describe a method to compute TT_{norm}. A network would be declared to have converged to the final configuration if its TSSE is less than *ecrit* and TT is less than TT_{norm}. A network is declared not to have converged if it can not achieve TSSE less than *ecrit* even after TT_{norm} amount of training. For consistency, we assume that these non-converging networks actually converge after a large amount of time is spent in training them. Hence, the objective function values for networks that do not converge are very large negative numbers.

Currently, we have considered only linear objective functions of the form given by Equation (1.7). With a linear objective function, the method scales well across different problems. In our future research, we will study more general non-linear objective functions.

Finally, we assume that an unambiguous problem specification is supplied by the user. The user specifies all the parameters characterizing the input and the output spaces of the problem, namely, the number of inputs and outputs and the format of the inputs. We are studying only the classification problems, which have binary outputs. We further assume that a sample training set for training the networks is available. We make these simplifying assumptions in order to focus on the main problem of automating the design of neural networks.

1.7 Two Training Methods

There are two methods used in practice for training neural networks. One is the *ad hoc* method, as proposed by Rumelhart, Hinton, and Williams [20]. Here, one starts with a network, assigns random initial weights to it, chooses some values for the training parameters and trains the network by applying the BEP algorithm. The problem with this method is that if the chosen network does not converge, part or all of the computational time allowed may be expended. This implies that an unpromising configuration should be detected early in the training process and rejected immediately. A previously tested configuration or a new configuration should be trained in place.

The second method for training neural networks is the *Restart Method* proposed by Scott Fahlman [10]. This method overcomes the difficulty of getting stuck in a local minimum of the ad hoc method by

setting an upper limit on the amount of time (T_{max}) that one would spend on training a network. If the configuration does not get trained within T_{max}, then it is abandoned, and another configuration is selected for training. Obviously, the success of the method depends upon a wise choice of T_{max}, which is difficult to make. A high value of T_{max} would result in wasting computational resources on unpromising networks. A low value of T_{max}, on the other hand, would result in giving up promising networks before enough training has been performed on them. Further, no guidelines are provided for choosing networks, initial weights, and the values for the training parameters.

Both methods described above do not apply in situations where one has limited resources. In Chapter 2, a resource-constrained design method will be presented. An important feature of this design method is that it is always able to propose one or more promising network configurations at its termination. The method also guides the selection of appropriate networks for solving the problem. The choice of initial weights and assignment of the values to the training parameters are also intelligently made. The proposed method uses *learning–by–experimentation* and aims at discovering heuristic configurations based on modeling the design process as a search [18]. The method is divided into two sets of experiments: Type I and Type II. These experiments are discussed in detail in Chapters 3 and 4. In Chapter 5, experimental results on testing the method on a variety of benchmarks selected from the list maintained at Carnegie-Mellon University by Scott Fahlman [9] and the UCI repository of machine learning databases maintained at University of California, Irvine by David W. Aha [2] are described. Conclusions and future research directions are presented in Chapter 6.

CHAPTER 2

THE DESIGN METHOD

The design method, proposed in this thesis, starts with a user-supplied objective function for comparing networks. For any problem, there are infinitely many networks with infinitely many initial weights and values of the training parameters for solving it. However, not all of these (initial) configurations would lead to (final) configurations that *maximize* the objective function. Since there is no systematic method to guide the selection of promising configurations and to prune unpromising configurations, the solution obtained must necessarily be heuristic. Further, it is impossible to train even a small fraction of the possible configurations to completion with the resources available. This means that one needs to schedule the available resources efficiently to arrive at better configurations.

There are two approaches to allocating resources. One can either allocate all the resources to one configuration and, in the end, inform the user whether the network converges or not (as done in the ad hoc method). Alternatively, one can divide up the resources into several equal parts, dynamically allocate these parts to various configurations which are possible candidates for solving the problem and, in the end, pick the best configuration. Obviously, the second approach is intuitively more appealing. In the design method, we divide up the available time resource into equal parts, where each part is known as a *time quantum*.

The block diagram of the design method is shown in Figure 2.1. The method is divided into two sets of experiments: Type I and Type II. Type I experiments are used to find promising configurations, while Type II experiments are used to refine the evaluation criterion used in Type I experiments. These experiments are discussed briefly in the following discussion and in [23].

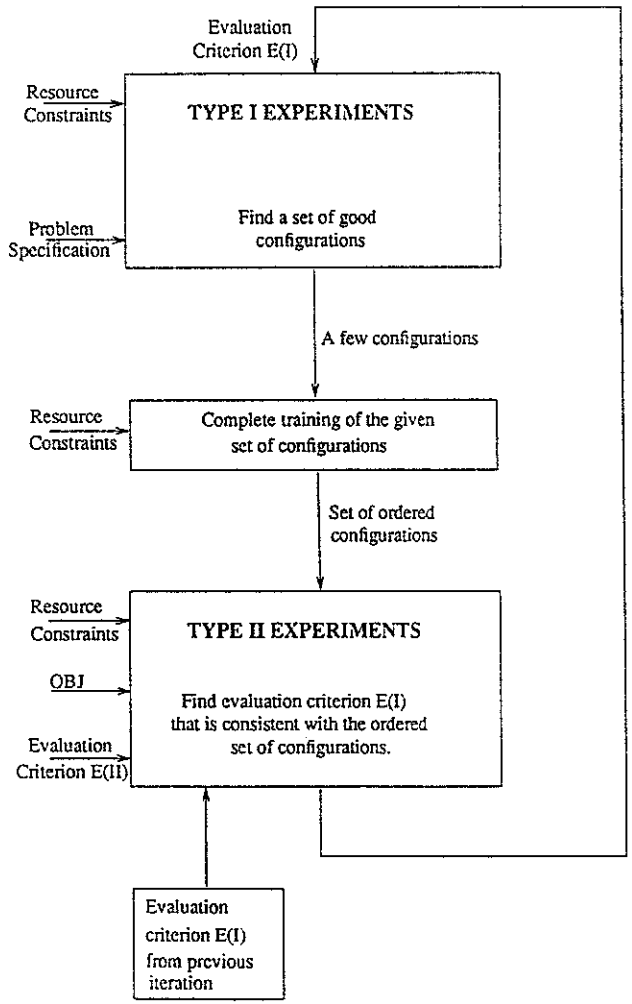

Figure 2.1: Block diagram of the design method.

2.1 Type I Experiments

Type I experiments try to find a set of good configurations, given the problem specifications, the resource constraints and an evaluation criterion, E(I), for evaluating alternative configurations. The result of Type I experiments is a set of promising configurations, selected on the basis of E(I). These experiments are in the form of a search, as depicted in Figure 2.2. The total computational time is divided into quanta. In each quantum, limited experiments are performed on a chosen configuration. Evaluation metrics for the configuration are computed at the end of the quantum based upon the experimental results obtained for all the quanta of time spent on it. Depending upon the results of this evaluation, the search continues with either the configuration with the best result or a new configuration.

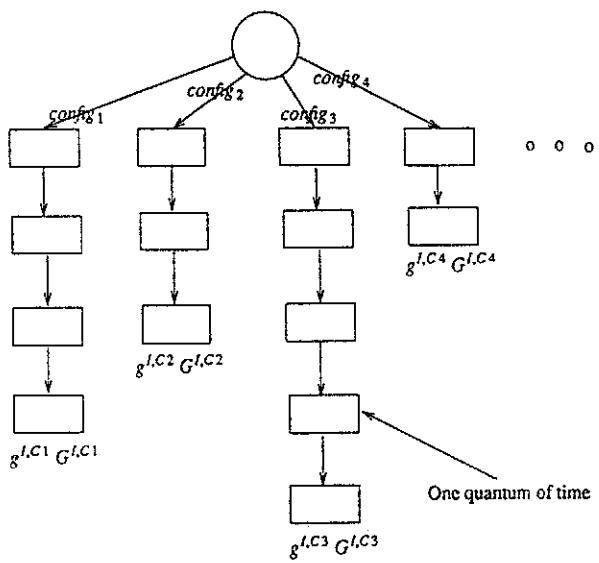

Figure 2.2: Organization of Type I experiments.

We use two functions in the evaluation criterion E(I): *goodness function*, g^I, and *guidance function*, G^I. The *goodness function* tries to order the configurations based upon their individual performances. Thus, the goodness function provides the order of *goodnesses* or merits of various configurations. It tries to select the best configuration or a set of best configurations at the end of Type I experiments, based upon the results of the experiments. However, since limited experiments have been carried out on each configuration, and the goodness function is heuristic in nature, the best configuration may not be precisely identified.

The *guidance function* selects the best configuration for experimentation, if Type I experiments were to be continued. It is a function of the goodness value of the configuration and the resources available. The reason for the guidance function to be different from the goodness function is to prevent a promising configuration from dominating all the resources in the system. Goodness and guidance functions are described in detail in Chapter 3.

Since the evaluation criterion supplied to Type I experiments may be wrong, it must be further refined. This is done in Type II learning experiments. To establish an absolute ordering on the configurations generated in Type I experiments, before Type II experiments can be carried out, we must train these configurations completely on the given problem. This is necessary because at the end of Type I experiments, not all configurations are trained to completion and therefore we need to train them completely before feeding to Type II. These exhaustive training experiments are carried out separately under specified resource constraints.

2.2 Type II Experiments

Type II experiments try to find an evaluation criterion E(I) consistent with the exhaustive evaluations already performed on the selected set of configurations from Type I experiments. The Type II learning process utilizes feedback information on whether the E(I) proposed last has improved the quality of the configurations generated. The process modifies its strategy (the evaluation criterion for Type II, E(II)) for generating E(I) accordingly.

Type II experiments can be implemented in a way similar to Type I experiments (see Figure 2.2). Here, again, time is divided into quanta, and in each quantum, limited experiments are performed on alternative E(I)s using a chosen evaluation criterion E(II). The problem with this approach is that E(II) for guiding the search in Type II experiments is heuristic and therefore must be verified and corrected by performing Type III experiments, which leads to E(III) and Type IV experiments, and so on. This recursive refinement of evaluation criteria renders the method useless because there is no way to define a standard evaluation criterion.

Fortunately, by re-examining the Type II learning process, we find that it is not necessary to carry out higher-order experiments. We do not need to divide time into intervals and perform overhead-intensive experiments on various E(I)s in Type II experiments. Quanta are necessary in Type I experiments because no experiments on various configurations have yet been performed, and we have to schedule the limited resources available. We note that, before Type II experiments are carried out, exhaustive experiments on various configurations proposed by Type I learning have already been performed. Thus, the traces of the individual experiments can be saved and therefore need not be duplicated in Type II experiments. As a result, we do not need E(II) to guide the scheduling of resources for E(I)s in Type II experiments (as required in Type I experiments).

The way various E(I)s are compared in Type II experiments is as follows. Let us suppose, at any time instant, two configurations are being compared, as shown in Figure 2.3. We assume that complete

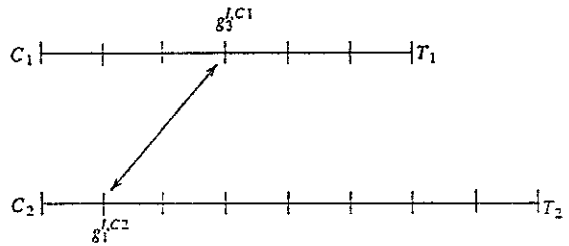

Figure 2.3: A step in evaluating two configurations in Type II.

training of the two configurations, C_1 and C_2, has been performed and that C_1 has a higher objective function value compared to C_2. As an illustration, if $g_3^{I,C1}$ (the goodness value using g^I computed at time 3 for C_1) is worse than $g_1^{I,C2}$, then g^I computed at these two times is not *consistent* with the final evaluation results on C_1 and C_2. A penalty is given to g^I. By repeating the above process over all possible pairs of quanta in the training and all pairs of configurations, we try to find an evaluation criterion E(I) that has the lowest total penalties associated with it.

In the next two chapters, we discuss the two types of experiments in detail.

CHAPTER 3

TYPE I EXPERIMENTS

The main components of Type I experiments are identified in Figure 3.1. Each of these components is discussed in detail below.

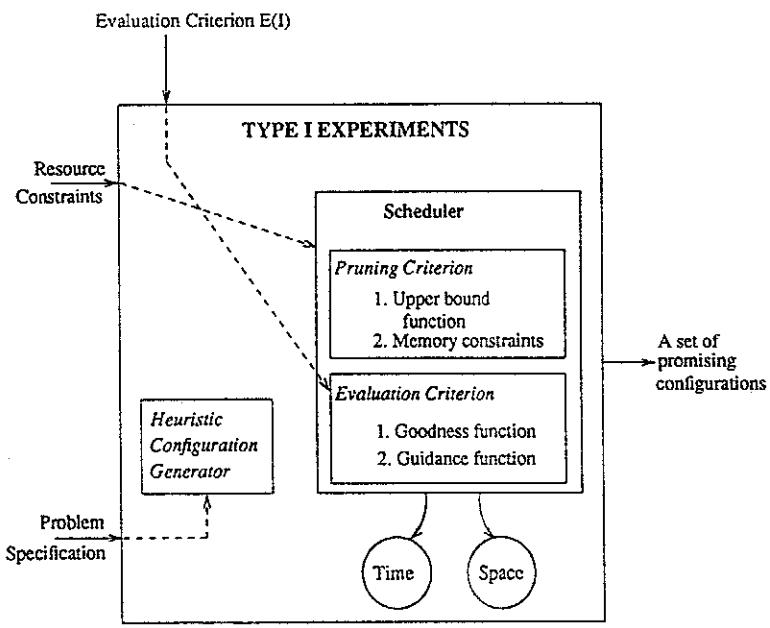

Figure 3.1: Main components of Type I experiments.

3.1 Heuristic Configuration Generator

The Heuristic Configuration Generator (HCG) is a demand-driven expert system for generating new network configurations. Configurations are generated in the beginning and during experiments in Type I learning. A new configuration may be generated by applying one of the following primitive *pattern operations* to an existing network:

(a) Add or remove a unit from one of the hidden layers;

(b) Add or remove a hidden layer from the network;

(c) Increase or decrease one of the learning algorithm parameter values;

(d) Generate an alternative initial weight configuration.

A Figure-of-Merit (FOM) can be used to monitor the performance of the configurations generated so far. One possible FOM could be the guidance function, which measures how well a configuration has been performing in a particular session. However, when very many configurations have converged in Type I experiments, average objective function value could be another such FOM that could be used.

Since the performance of various configurations coming from the same network could vary by orders of magnitude, we alternate between the following two meta-rules in generating new configurations when the HCG is invoked.

(1) Based upon the FOM of the configurations generated so far, apply one of the above pattern operations to the best network.

(2) Select a network randomly from the set of networks considered so far and generate a new set of initial weights.

We are in the process of developing the program for the HCG. Progress of this project will be reported in later publications. Currently, we have developed and tested a simplified version of the program, which asks the user to specify the possible choices for the networks and the values of the learning algorithm parameters. The program searches through the initial weight space and finds a good

set of weights to solve the problem. In short, in the current implementation, the user specifies a pool of possible candidate networks and the values of the training parameters to solve the problem.

The HCG is invoked to generate a new network configuration whenever one of the following two conditions is true:

(a) Whenever a trained network exits Type I experiments.

(b) If there has not been much improvement in the performance of any of the current configurations (such as guidance value improvement $\leq 10\%$) in the last T_{add} quanta.

The number of configurations generated by the HCG is an important parameter in the space-time trade-off of scheduling the available time for Type I experiments. If very many configurations are generated, then the amount of time spent on each may not be enough to obtain stable and meaningful performance results. If, on the other hand, too few configurations are generated, then an insufficient number of configurations are tested, and the resulting configurations reported by Type I experiments may not represent the best choice. Another consideration on the number of configurations to be generated is the memory-space constraint. An exceedingly large number of active configurations in (real or virtual) memory may reduce the processor efficiency to as little as 10%. As a result, the available time for Type I experiments is reduced significantly. In our current experiments, we view Type I experiments as a *window* which configurations are constantly added to or removed from. The number of configurations present at any time in the window is called the *window size*. The window size is set dynamically so that a sufficient number of configurations are either experimented or pruned.

3.2 The Scheduler

The Scheduler in Type I experiments schedules the available time and space so that a sufficiently large number of configurations can be examined and adequate training can be performed on each. It avoids wasting expensive resources on configurations which are not promising and prevents a very promising configuration from dominating all the available resources. It makes use of the *Evaluation criterion*, E(1), in making scheduling decisions. The Evaluation criterion is used to compare the

performance of configurations at the end as well as during experiments. It consists of two parts, the goodness function and the guidance function.

3.2.1 The goodness function

The goodness function selects the best configuration or a set of the best configurations based upon the results obtained at the termination of Type I experiments. If a configuration has been trained to completion, then its goodness function value is the same as its objective function value. Otherwise, the goodness function value represents an estimate of the objective function value based upon the training information available so far. If a configuration has not been trained completely, then it is necessary to estimate its training time, in order to predict its objective function value.

In our experiments, we set a ceiling (TT_{norm}) on the maximum training time that the Scheduler would spend on any configuration. This time limit (TT_{norm}) is a practical representation of infinite time in our experiments, i.e., if a configuration does not converge in TT_{norm} training time then we declare that the configuration is stuck in a local minimum and it would require a very large amount of training time to converge. Assume that t ($\leq TT_{norm}$) units of time have been spent on a configuration when Type I experiments terminate. If TSSE of the configuration is below the preset threshold *ecrit*, then the configuration has converged. Otherwise, we need to estimate its training time, which is equivalent to predicting the convergence point in the learning curve. The learning curve for a configuration is a plot of the TSSE as a function of training time. It shows how the error in the configuration varies as the training proceeds. A typical learning curve for the 4 bit encode-decode problem is shown in Figure 3.2. In the next chapter, we describe a method for setting TT_{norm} and an algorithm for distinguishing promising configurations from unpromising ones.

Some of the parameters that might be helpful in predicting the training time of a configuration include

(a) The time spent in training the configuration thus far;

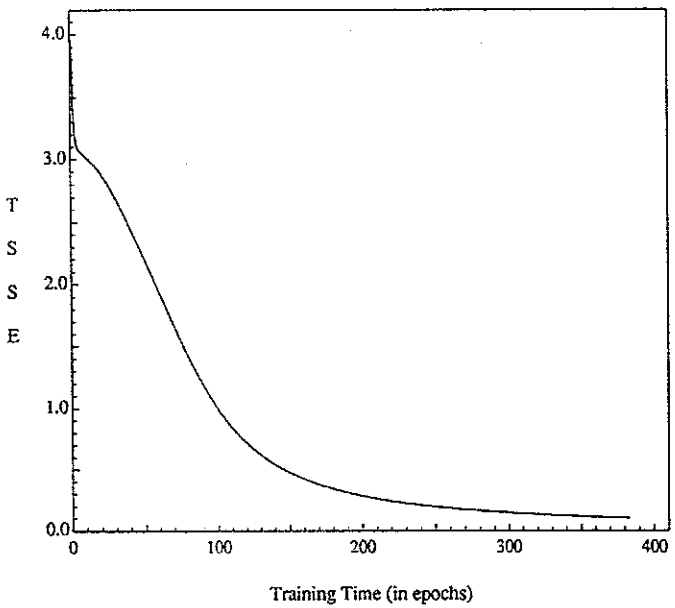

Figure 3.2: A typical learning curve for the 4-bit encode-decode problem. (4-2-4 network, $\eta = 0.5$, $\alpha = 0.5$, *ecrit* = 0.1)

(b) Its current TSSE value;

(c) Its current slope and higher-order derivatives of the learning curve; and

(d) The cost (or complexity) of the network.

The cost of a network is measured by either the memory and computational requirements, if the network is simulated on a digital computer, or the cost of a VLSI implementation, if it is implemented in hardware. The cost of a network is a function of parameters such as the number of units, the number of

layers, and the layout of the network. In this thesis, we heuristically express cost of a network by the following expression.

$$Cost = Number\ of\ units + 0.1 \times \sqrt{Number\ of\ weights} \tag{3.1}$$

3.2.2 The guidance function

The guidance function selects the best configuration for experimentation, if Type I experiments were to be continued. The guidance function tries to ensure that the remaining computational resources should not be spent on unpromising networks or be dominated by a few very promising ones. It heuristically measures the weighted performance of a configuration as a function of its goodness value, the amount of resources spent on this configuration, and the amount of resources remaining. By assigning a penalty to the goodness value of a configuration for the amount of time spent on it, the guidance function ensures that a configuration which is very promising in the beginning does not dominate all the resources. The reason for this is twofold. First, the goodness function is heuristic in nature and may be fallible. Consequently, a configuration that looks very promising in the beginning may get stuck in a local minimum in the end. Second, even if a configuration is very promising, we would still like to examine alternative configurations in the time available. This allows a better decision to be made when Type I experiments terminate.

The guidance function is set up so as to maximize some utility to cost ratio [22]. Goodness value (the expected objective function value) could be one candidate for the utility function and the cost of training the configuration to completion could be a candidate for the "cost" function.

3.3 The Pruning Criterion

The Pruning Criterion uses the upper-bound and the memory-constraint functions to prune the unpromising configurations in the experiments.

3.3.1 The Upper-Bound Function

The Upper-Bound Function (UBF) measures the maximum amount of time that the Scheduler should spend in training a configuration in order to obtain better objective function values. Initially, when no existing solution for the problem is known, the UBF values for all the configurations are set equal to TT_{norm}. As configurations are trained to completion, the UBF values of all the configurations are modified as follows: The value of UBF for a configuration is chosen so that the objective function value of this configuration is no worse than the objective function value of the best configuration obtained so far. Using the objective function defined in Eq. (1.7), the following equality must be satisfied.

$$A - B \times TT_{best} - C \times Cost_{best} = A - B \times UBF_{config} - C \times Cost_{config} \tag{3.2}$$

where TT_{best} and $Cost_{best}$ are, respectively, the training time and the cost of the best configuration known, and UBF_{config} and $Cost_{config}$ are, respectively, the upper bound and the cost of the configuration under consideration. Rewriting Eq. (3.2) yields

$$UBF_{config} = TT_{best} + \frac{C \times (Cost_{best} - Cost_{config})}{B} \tag{3.3}$$

3.3.2 The memory-constraints function

The memory-constraint function is applied in the course of Type I experiments to prune configurations with the lowest goodness values in order to reduce the memory usage to below a specified limit. This effectively improves the processing efficiency of the processors because it eliminates the extensive swapping of pages when memory usage is large.

CHAPTER 4

TYPE II EXPERIMENTS

As pointed out earlier, the evaluation criterion used in Type I experiments, which consists of the goodness and the guidance functions, could be wrong and, therefore, it must be refined in subsequent experiments. This is done in Type II experiments. The goal of Type II experiments is to produce goodness and guidance functions that are capable of identifying configurations with better objective function values in Type I experiments. Since the guidance function value for a configuration is a function of its goodness function value, we will focus our attention on the task of learning the goodness function first.

The goodness function values are used to compare configurations at the end of Type I experiments. Therefore, a good goodness function is one which can arrange configurations in the order of their objective function values, whenever the experiments are stopped. If it is not possible to achieve such a goodness function, we would like to arrive at a function which minimizes the number of places where such an ordering is violated. Hence, a good goodness function is one which encounters the least number of ordering inconsistencies. A few possible places where such inconsistencies might occur are shown in Figure 4.1.

4.1 Normalization of Parameters

Before the goodness function, and hence the guidance function, can be learned, it is necessary to normalize the parameters that these functions depend upon, so that the results obtained from one set of experiments become applicable across different experiments. In this thesis, we measure the goodness of a configuration by three independent parameters, viz., TSSE, cost and training time. Other parameters,

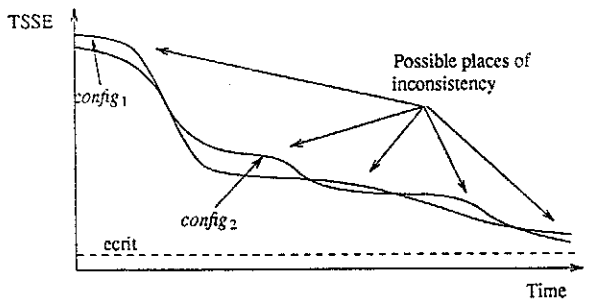

Figure 4.1: Possible places of inconsistency in two learning curves.

such as the slope and the higher-order derivatives of the learning curve, depend upon the three independent parameters. Therefore, we need to normalize the independent parameters only.

TSSE: Due to the random choice of the initial weights and the sigmoidal non-linearity (Eq. 1.3), we can claim that the average output value of any output unit in a back-prop network is equal to 0.5 at the start of the training. Thus, the average value of TSSE in a network at the beginning of the training is given by the following formula.

$$TSSE_{av} = No\ of\ patterns \times No\ of\ output\ units \times 0.25 \tag{4.1}$$

It has been experimentally verified for all the problems studied in this thesis that the maximum value of TSSE remains within 20% of the above value in most cases. Therefore, we use the following equation for normalizing TSSE.

$$TSSE_{norm} = 1.5 \times TSSE_{av} \tag{4.2}$$

If in case a value of TSSE higher than $TSSE_{norm}$ is observed during Type I experiments, the system makes a note of it and prompts the user to refine the above equation at the end of the experiments.

Cost: C_{norm} should be a multiple of the average cost of the networks considered for solving the problem. In our experiments, the user supplies a guess for C_{norm} at the start of the experiments. As more

and more experiments are conducted, the system refines the value of C_{norm}. For our current implementation, where the user specifies a pool of networks, this parameter is set equal to some multiple (= 3) of the average cost of all the networks in PON.

Training Time: As with cost, TT_{norm} should also be a multiple of the average training time of the networks being considered. In our current implementation, the user supplies some guess for this parameter.

4.2 Learning Goodness Function

The goodness function value of a configuration is the same as its objective function value when the training is completed. Otherwise, it is an estimate of the objective function value. Accordingly, we can set the goodness function as the objective function in its derivation.

$$\begin{aligned}
\begin{matrix} Goodness \\ Function \end{matrix} &= Objective\ Function \\
&= A - B \times TT - C \times Cost \\
&= A - B \times (TT - t + t) - C \times Cost, \quad where\ t \le TT \\
&= [A - B \times (TT - t)] - [B \times t + C \times Cost] \\
&= \left[A - B \times TT_{norm} \times \frac{(TT - t)}{TT_{norm}} \right] - \left[B \times TT_{norm} \times \frac{t}{TT_{norm}} + C \times Cost \right]
\end{aligned} \tag{4.3}$$

Since the goodness function is used only for comparing configurations, one could divide the function by a constant without affecting its role in the comparison process. Dividing the above equation by $B \times TT_{norm}$,

$$Goodness\ Function = \left[A' - \frac{(TT - t)}{TT_{norm}} \right] - \left[\frac{t}{TT_{norm}} + C' \times Cost \right] \tag{4.4}$$

The above equation represents the goodness of a configuration which has been trained for time t. Let us call the first term in the above equation the *expected goodness function* and the second term the *penalty function*. That is,

$$\textit{Expected Goodness Function} = g^{EG} = \left[A' - \frac{(TT - t)}{TT_{norm}} \right] \; ; \; A' = \frac{A}{B \times TT_{norm}} \tag{4.5}$$

$$\textit{Penalty Function} = -\left[\frac{t}{TT_{norm}} + C' \times Cost \right] \quad \textit{where } C' = \frac{C}{B \times TT_{norm}} \tag{4.6}$$

4.2.1 The penalty function

The penalty function is the cost of training the configuration up to time t. It can be readily computed given the amount of time spent in training the configuration so far and the complexity of the configuration. Note that the parameters B, C and TT_{norm} are known at the start of Type I experiments.

4.2.2 The expected goodness function

The expected goodness function is a measure of the proximity of the present configuration from the final (completely trained) configuration. Its computation requires an estimate of the training time. Obviously, when $t = TT$, i.e., when training is completed, g^{EG} is a constant independent of the specific parameters of the configurations. We further note that g^{EG} is a monotonically increasing function of the time already spent on a configuration and a monotonically decreasing function of the remaining training time. Since the training time for a configuration is not known until training has completed, Eq. (4.5) has to be approximated heuristically. One such possible heuristic for measuring g^{EG} is as follows.

$$\hat{g}^{EG} = \left[1.0 - \frac{TSSE}{TSSE_{norm}} \right] \tag{4.7}$$

This heuristic satisfies all the necessary properties of g^{EG}. However, it can be further improved by including other information, such as the slope of the learning curve and its higher-order derivatives. This improvement is done in Type II experiments. These experiments start with $\hat{g}^{EG}$ defined in Eq. (4.7), or some other initial guess, and refine the heuristic further until the specified resources are exhausted. The final heuristic, found at the end of these experiments, is outputted as the best heuristic found in the available computational resources.

An essential step in Type II experiments is to be able to compare alternative expected goodness functions and to reject choices that are inferior. Assume two configurations $config_1$ and $config_2$ with training times T_1 and T_2, respectively, and $T_1 < T_2$. Suppose $config_1$ has been trained for t_1 time units and $config_2$ for t_2 time units, where $(T_1 - t_1) < (T_2 - t_2)$. We would expect then that

$$\hat{g}^{EG}_{config\,1} \geq \hat{g}^{EG}_{config\,2} \tag{4.8}$$

since $config_1$ is closer (as far as training goes) to the final state as compared to $config_2$. Thus, if the proposed heuristic $\hat{g}^{EG}$ does not satisfy Eq. (4.8), we say that an inconsistency has occurred. A refined goal of Type II experiments is, therefore, to propose a heuristic for the expected goodness function which minimizes all possible inconsistencies for all pairs of configurations (supplied to Type II experiments) at all intermediate stages of training. More precisely, we would like to minimize the following *Inconsistency Sum* (IS), where

$$IS = X \times SMI + Y \times SIC + Z \times SID \tag{4.9}$$

where X, Y and Z are some constants and SMI, SIC and SID are respectively the sum of monotonic inconsistencies, sum of inconsistency counts and the sum of inconsistency differences. Wherever the proposed heuristic function violates the monotonicity constraint, we add the difference, weighted by its objective function value, to SMI for all the training traces. Wherever Eq. (4.8) is violated, the SIC count is incremented by an amount equal to the difference in training times of the two configurations. This is done for all pairwise combinations of configurations at all steps (quanta) of their training traces. Please note that for configurations which don't converge, we have assumed that their training times are TT_{norm}. Sum of inconsistency differences is similar to SIC, except that we sum the difference of the two proposed heuristic function values, weighted by the difference in their training times.

4.3 An Expert System for the Expected Goodness Function

We have developed an expert system in LISP and C programming languages to try out various functions and finally to propose a good heuristic for the expected goodness function. The schematic

diagram of the expert system is shown in Figure 4.2. The system consists of the following two building blocks: the *Function Generator* and the *Performance Measuring Element*.

4.3.1 The function generator

The function generator starts with an initial guess for the heuristic function. It successively applies a set of pattern operations to it to improve upon the initial guess. The set of pattern operations implemented includes:

(a) Increase or decrease the value of one of the coefficients;

(b) Change the sign of one of the terms; and

(c) Add a new term to the function.

The system is supplied with a library of possible candidate terms for the heuristic function. Sequencing of the pattern operations is performance driven and is decided by the set of meta rules supplied to the system. The set of meta rules implemented in our system is:

(1) Measure the sensitivity of applying the above pattern operations to the function, one at a time.

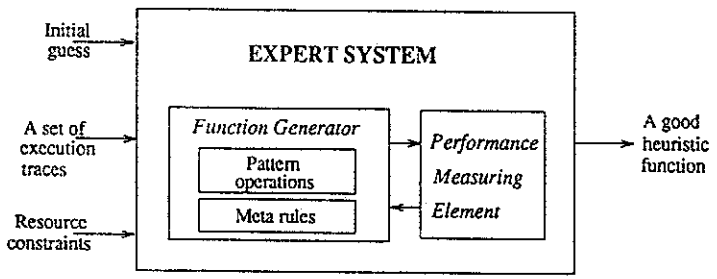

Figure 4.2: Schematic diagram of the expert system for learning Expected Goodness Function.

(2) Select a pattern operation (a) or (b), whichever performs better.

(3) Once a pattern operation is selected, repeatedly apply it to the function until it does not lead to any further improvement in performance.

(4) At this stage, again select either (a) or (b), and repeat the above process (step 3), until the pattern operation does not lead to any further improvement in the performance of the function. The function obtained at the end of the above greedy strategy is the best function obtained, with the given set of terms in the function.

(5) Add one more term to the function by applying the pattern operation (c) to it.

(6) Again, repeat the above greedy (steps 1 through 4) search for the best set of coefficients for this new function. The best set of coefficients obtained this time would not be the same as the one obtained last time.

(7) This process of successively adding new terms and refining the coefficients of the function (steps 1 through 6) is continued until the specified time resource is exhausted.

(8) At the end of the experiment, a function which performed the best during the experiments is outputted to the user.

The reason why we want to add terms to the function, one at a time and not all at once, is that we want to be able to stop the experiments at any stage and still be able to output a good heuristic function. If we start with a function which has all the library terms in it and then try to find the best set of coefficients for it, we may run short of resources. Any intermediate result outputted at this stage may lead to very poor performance.

4.3.2 The performance measuring element

The performance measuring element is a program written in C which takes a function from the function generator and outputs the inconsistency sum (Eq. 4.9) for this function (calculated from a set of execution traces already supplied to the expert system) to the function generator.

4.4 The Guidance Function

The guidance function is a function of the goodness function value for a configuration, the amount of resources spent on the configuration, and the amount of resources remaining. A good guidance function should be able to distinguish unpromising networks from promising ones at an early stage in Type I experiments. Thus, a guidance function learned in Type II experiments can result in a better performance in Type I experiments. Currently, the learning of new guidance functions is under implementation. Instead of learning new functions, we provide a heuristic guidance function to Type I experiments before they begin.

CHAPTER 5

EXPERIMENTAL RESULTS

In this chapter, we report the results of applying the proposed method to some of the benchmark problems collected from the connectionist mailing list being maintained at Carnegie Mellon University [9] by Scott Fahlman and from UCI repository of machine learning databases being maintained at University of California, Irvine [2] by David W. Aha. Details on these benchmark problems are summarized in Appendix A. Most of the experiments in this research were performed on Sun 3/60 workstations and a Stardent 4-processor computer. We have experimented extensively with the encode-decode problem with a varying number of bits. In order to study the scalability of the design method across different problems from the same family, we have studied the effect of using various functions obtained from a smaller problem in a larger one. The second benchmark we have chosen for our study is the two-spiral benchmark. This benchmark is known to take training times in hundreds of thousands of epochs, and so it is suitable for demonstrating that our method is applicable to large problems as well. An *epoch* is defined as a single presentation of each of the patterns in the training set, either in a fixed or a random order. Other benchmarks studied include the space shuttle autolanding domain, iris plants database and the lymphography domain.

In Figure 5.1, we show how Type I experiments proceed as a function of time for the 4-bit encode-decode problem. The initial window size is 3. At the start of the experiments, three configurations are selected from the pool of configurations and trained for one time quantum each. At the beginning of the fourth quantum, a configuration with the best guidance value ($config_3$) is selected. This configuration dominates other configurations until it is trained to completion in the ninth quantum. At the end of the ninth quantum, $config_3$ exits the window, and another configuration, $config_4$, is included in the active list

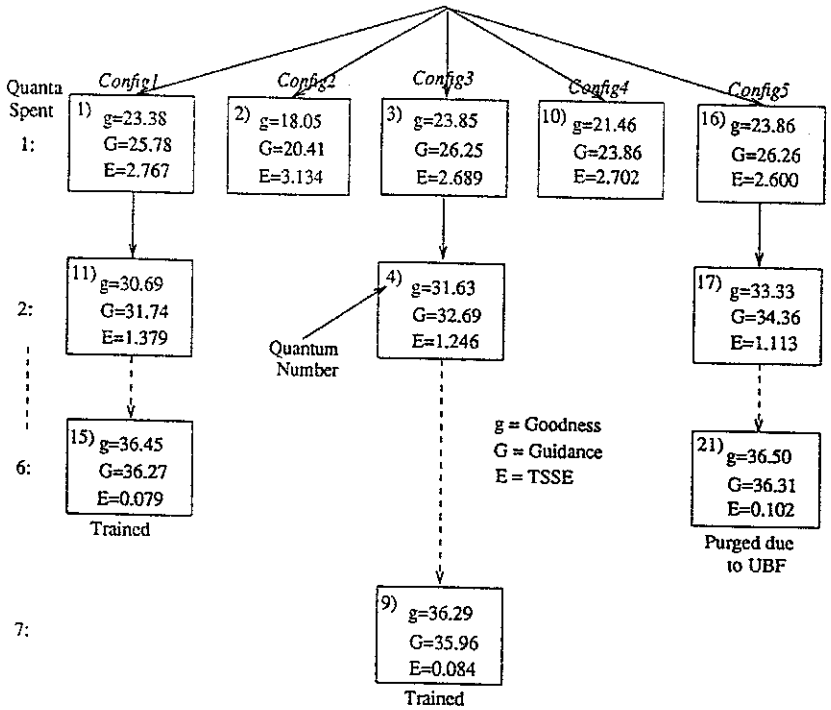

Figure 5.1: Trace of Type I experiments for the 4-bit encode-decode problem.

of configurations. This configuration is trained for one quantum (tenth). Again, at the start of the eleventh quantum, a configuration with the best guidance value is selected. This time, $config_1$ has the highest guidance value and dominates other configurations until it is trained to completion. This process continues until the specified time limit is reached.

All the configurations in the experiment shown in Figure 5.1 were chosen to have the same cost. Thus, the moment a configuration is trained to completion, UBF of all the configurations is set to this

training time value. Consequently, at the end of the twenty-first time quantum, $config_5$ is purged from the window, since six time quanta have been spent on it and its TSSE still does not satisfy the stopping criterion. (*ecrit* was set to 0.1 for all the networks).

In Figure 5.2, we have plotted the average objective function values (averaged over 10 experiments) for the 4-bit encode-decode problem against various durations of Type I experiments. This curve shows that, on an average, the resulting objective function value obtained improves as Type I experiments are

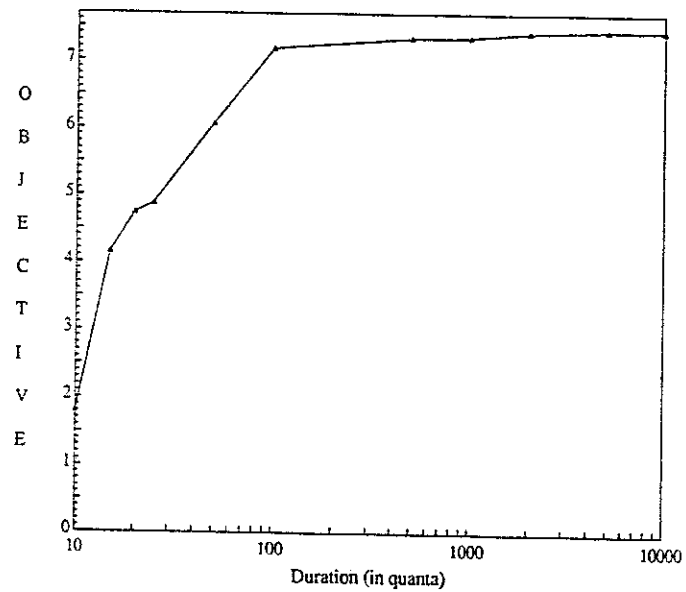

Figure 5.2: Average objective function values versus durations in Type I experiments for 4-bit encode-decode problems. (1 quantum = 10 epochs)

allowed to run for longer durations. We further note that beyond a certain duration, allowing Type I experiments to run longer does not improve the average objective function values significantly and a leveling-off effect is observed. In this figure (and those that follow), we have replaced the objective function values of those networks which do not converge by zero. In Figure 5.3, we plot the same curve, as seen in Figure 5.2 for the 8-bit encode-decode problem. We have used the same goodness and guidance functions as used in the 4-bit case. From the figure, it can be seen that a behavior similar to the 4-bit case

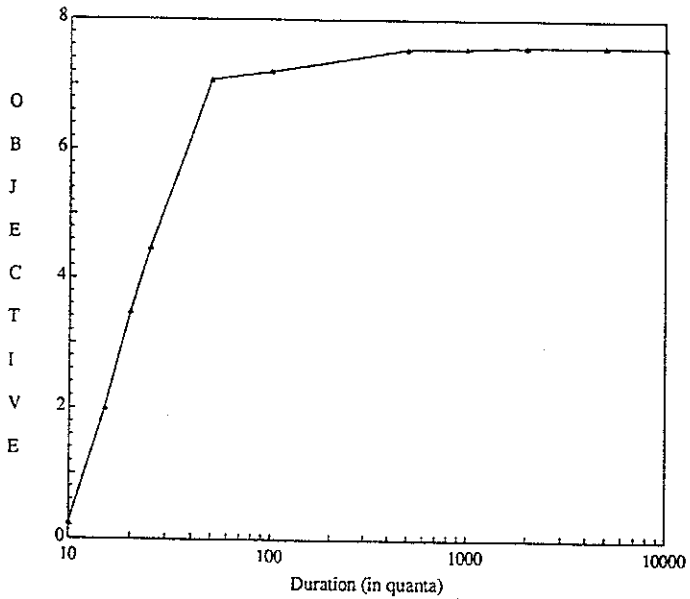

Figure 5.3: Average objective function values versus durations in Type I experiments for 8-bit encode-decode problems. (1 quantum = 10 epochs)

is observed. This shows that our method scales well across problems of different sizes. As a result, the goodness and guidance functions learned in solving the 4-bit encode-decode problems do not have to be relearned for solving the 8-bit encode-decode problems. In Figures 5.4, 5.5, 5.6 and 5.7, we show similar plots for the two-spiral, the space shuttle autolanding domain, the iris plants database and the lymphography domain benchmark problems, respectively.

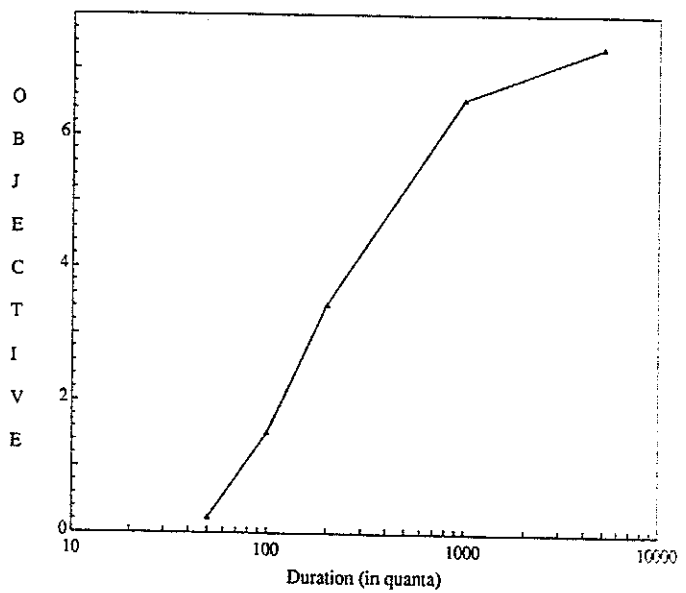

Figure 5.4: Average objective function values versus durations in Type I experiments for two-spiral problem. (1 quantum = 100 epochs)

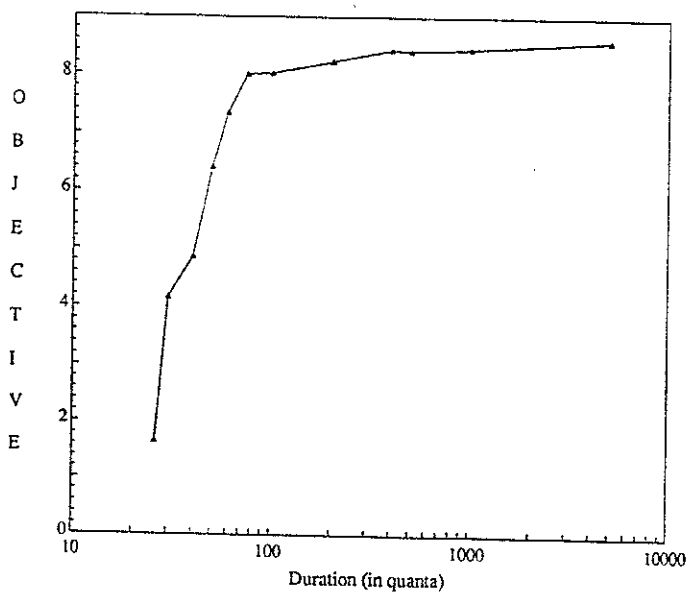

Figure 5.5: Average objective function values versus durations in Type I experiments for space shuttle autolanding domain problem. (1 quantum = 10 epochs)

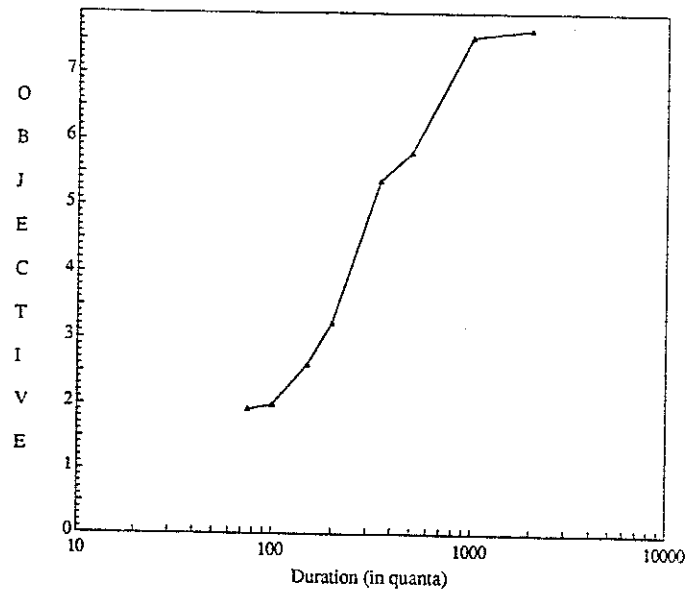

Figure 5.6: Average objective function values versus durations in Type I experiments for iris plants database problem. (1 quantum = 100 epochs)

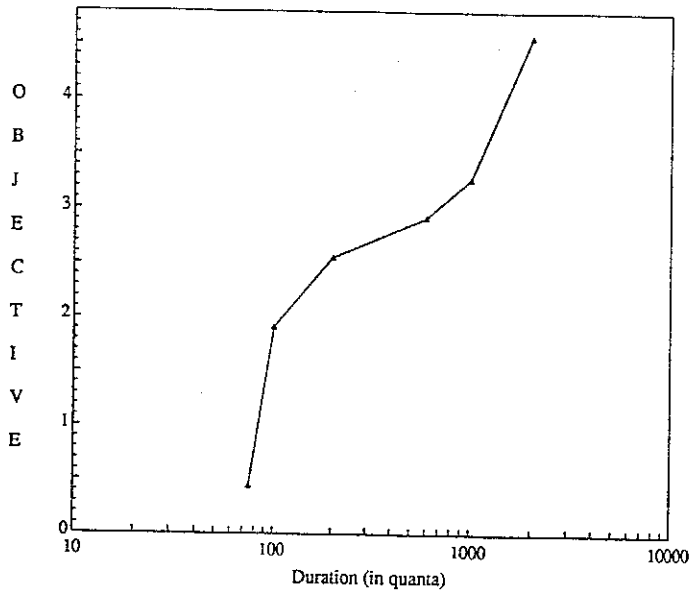

Figure 5.7: Average objective function values versus durations in Type I experiments for lymphography domain problem. (1 quantum = 10 epochs)

In Figure 5.8, we have plotted a typical trace of Type II experiments for the 4-bit encode-decode problem. A set of configurations was selected and trained either to completion or until the time spent on them exceeded a fixed ceiling. These traces were fed to Type II experiments, which tried out various heuristic functions for the expected goodness function. The initial guess for the expected goodness function was the same as given in Eq. (4.7). Figure 5.8 shows a plot of the inconsistency sum (IS) for each of the functions tried out versus the step number at which the particular function was generated. Each Δ in the figure marks the place where a new term was added to the heuristic function. The addition of a new term required an adjustment in the values of all the coefficients, which explains why the

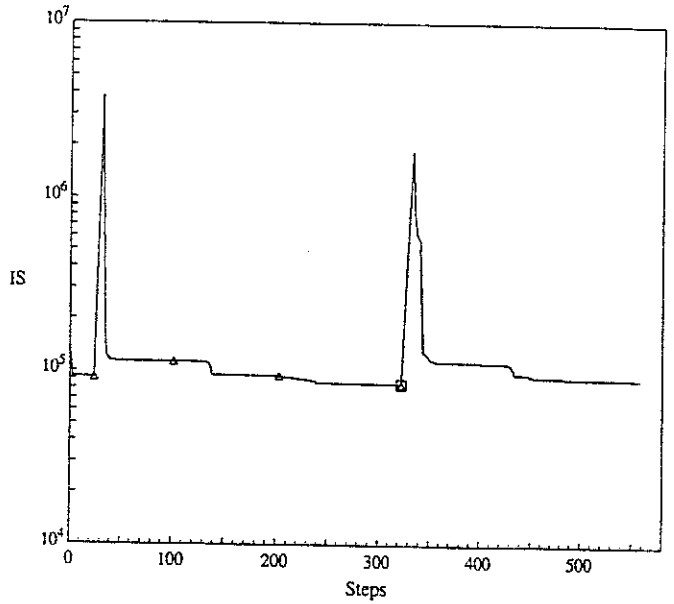

Figure 5.8: Inconsistency sum versus steps in Type II experiments. (1 step ≈ 55 sec)

performance of the function, obtained after adding a new term, worsened at times. After each step, the performance of the current function was compared with that of the best function obtained so far. If the current function was better, it replaced the best function. After exploring 560 functions, the best function was outputted as the best possible heuristic found in the time and resources available. This best function outputted is marked by □ in the figure.

CHAPTER 6

CONCLUSIONS

In this thesis, we have identified some of the major problems associated with training an artificial neural network using the back propagation algorithm. In an attempt to overcome these problems, we have presented a systematic design method for automating the design of neural networks using a fixed amount of computational time and space. By modeling the process of finding a suitable neural network for solving the given problem as a search process with an infinite search space, we need to find heuristic functions to guide and prune nodes in the search. We developed a learning method for acquiring problem-independent and scalable parameters for these heuristic functions. Our experimental results on the encode-decode, two-spiral space shuttle autolanding domain, iris plants database and lymphography domain problems illustrate that the design method can be applied across different problems and that it scales well for problems of different sizes.

The development of the system is not completed at this point. We are currently conducting extensive experiments with other benchmark problems. These will improve the heuristics and the rules that we have used in our current prototype. A more systematic development of the expert system for generating new networks is also being carried out at this time. This expert system will incorporate existing knowledge and experience on neural-network research in proposing promising networks to be tested in the experiments.

APPENDIX A

BENCHMARK PROBLEMS

A.1 Two-Spiral Problem

NAME: Two Spiral

SUMMARY: The task is to learn to discriminate between two sets of training points which lie on two distinct spirals in the x-y plane. These spirals coil three times around the origin and around one another. This appears to be a very difficult task for back-propagation networks and their relatives. Problems like this one, whose inputs are points on the 2-D plane, are interesting because we can display the 2-D "receptive field" of any unit in the network.

SOURCE: Posted to "connectionists" mailing list by Alexis Wieland of MITRE Corporation. Anonymous ftp from pt.cs.cmu.edu from /afs/cs/project/connect/bench directory.

MAINTAINER: Scott E. Fahlman, CMU

PROBLEM DESCRIPTION: The following fragment of C code, supplied by Wieland, generates the two sets of points, each with 97 members (three complete revolutions at 32 points per revolution, plus endpoints). Each point is represented by two floating-point numbers, the x and y coordinates, plus an output value of 0.0 for one set, 1.0 for the other set. (Some algorithms may require different output values, such as +1.0 and -1.0 or 0.2 and 0.8.)

```
main()
{
 int i;
 double x, y, angle, radius;

 /* write spiral of data */
```

```
for (i=0; i<=96; i++) {
    angle = i * M_PI / 16.0;
    radius = 6.5 * (104 - i) / 104.0;
    x = radius * sin(angle);
    y = radius * cos(angle);
    printf("((%8.5f %8.5f)  (%3.1f))\n", x, y, 1.0);
    printf("((%8.5f %8.5f)  (%3.1f))\n", -x, -y, 0.0);
}
}
```

METHODOLOGY: The task is to train on the 194 I/O pairs until the learning system can produce the correct output for all of the inputs. The time required for learning is then recorded. For back-propagation systems, the time is typically recorded in "epochs." Each epoch includes one training presentation for each of the 194 points in the training set.

The choice of output values for the two sets is up to the experimenter. For uniformity in reporting results, we suggest that the 40-20-40 criterion be used: an output is considered to be a logical zero if it is in the lower 40% of the output range, a one if it is in the upper 40%, and indeterminate (and therefore incorrect) if it is in the middle 20% of the range.

This task can obviously be solved quickly by methods that essentially do table-lookup, recording the desired result for each point; on the other hand, the problem appears hard for back-propagation networks and is impossible for a linear separator. The interesting question is how learning time varies with the algorithm chosen and the total number of weights (or equivalent) in the network.

VARIATIONS: It is possible to vary the density of points along the spiral by incrementing the angle by a smaller amount, while still making three complete revolutions in each set. Lang and Witbrock [14] report some results using the 4X version of the problem, with 770 points in all.

RESULTS: Lang and Witbrock [14] report results obtained on this problem using a 2-5-5-5-1 back-propagation network with short-cut connections: each unit is connected to every unit in all earlier layers, not just to units in the previous layer. Counting the unit thresholds, there are a total of 138 trainable weights in the network.

Three trials were run using standard back propagation on this network, with a uniform distribution of random initial weights in the range -0.1 to +0.1. Learning rate was initially 0.001 and momentum was initially 0.5; these values were "gradually" increased over 1000 epochs to final values of 0.002 and 0.95. The learning times on the three runs were 18,900 epochs, 22,300 epochs, and 19,000 epochs. (Average = 20,000 epochs.)

Using the same network and starting values, but with a nonlinear "cross entropy" error function, learning times were 16,200 epochs, 8600 epochs, and 7600 epochs, respectively. (Average = 11,000 epochs.)

With the same network and starting values, but using Fahlman's quickprop algorithm and hyperbolic arctangent error [10], the times were 4500 epochs, 12,300 epochs, and 6800 epochs. (Average = 7900 epochs.) For this test, epsilon was 0.002, mu was 1.5, and the weight decay factor was 0.001.

Stephen A. Frostrom of SAIC reports the following unpublished result, obtained by Dennis Walker of SAIC: Standard back-propagation network, 2-20-10-1 with no short-cut connections (281 weights in all). Learning rate 0.1, momentum 0.7, unit activation running from -0.5 to 0.5, error was set to 0.0 if the output was within 0.15 of target. The task was learned (to within 0.3 tolerance) in 13,900 epochs. With a tighter 2-16-8-1 network, the task was learned, but it required 300,000 epochs.

Lang and Witbrock [14] report that the variation with 4X density was learned in 64,000 epochs using the same standard back-propagation setup that they used for the single density case.

COMMENTS: None.

A.2 Encode - Decode Problem

NAME: Encode - Decode Problem

SUMMARY: The task is to train a network to learn a problem in which a set of orthogonal input patterns is mapped to a set of orthogonal output patterns through a small set of hidden units. In such cases, the internal representations of the patterns on the hidden units must be rather efficient.

SOURCE: The problem was first posed by Ackley, Hinton and Sejnowski [1]. It was studied by David E. Rumelhart and James L. McClelland in [20], and Scott E. Fahlmann in [10]. This summary was written by the author.

PROBLEM DESCRIPTION and METHODOLOGY: The task is to map a set of N orthogonal input patterns to a set of N orthogonal output patterns through a small set of M hidden units. A network with N input units, M hidden units and N output units is commonly referred to as "N-M-N" network. There are connections from every input unit to every hidden unit and from every hidden unit to every output unit. Normally M is smaller than N, so that the network has a bottleneck through which information must flow. The network is presented with N distinct input patterns, each of which has exactly one bit set to 1, the rest are all 0. The task is to duplicate the input pattern at the output units. An interesting case occurs when $N = \log_2 N$. In this case, we expect that the system will learn to use the hidden units to form a binary code with a distinct binary pattern for each of N patterns. Such networks are referred to as *tight* networks. However, a real-world classification task is best represented by networks which have a few redundant hidden units. A set of input output patterns for the N = 4 case is presented below:

1.0 0.0 0.0 0.0 → 1.0 0.0 0.0 0.0

0.0 1.0 0.0 0.0 → 0.0 1.0 0.0 0.0

0.0 0.0 1.0 0.0 → 0.0 0.0 1.0 0.0

0.0 0.0 0.0 1.0 → 0.0 0.0 0.0 1.0

VARIATIONS: The problem can be tried for different values of N, which are integral powers of 2, i.e., N = 2, 4, 8 etc. Networks with a varying number of hidden units can be tried and the effect of hidden units on training time can be studied. Networks with M = N are not very interesting because that allows the network to memorize the patterns rather than extracting the general features of the problem, which improves its generalization capabilities. Complement encoder problems could also be studied in which each of the 1's is replaced by 0's and 0's by 1's.

RESULTS: Some of the intersesting results are summarized in [10] and the interested readers are encouraged to look at the paper.

COMMENTS: None.

A.3 Space Shuttle Autolanding Domain Problem

TITLE: Space Shuttle Autolanding Domain

SOURCES: Obtained by anonymous ftp from ics.uci.edu from pub/machine-learning-databases directory. The UCI repository of machine learning databases is maintained by the site librarian, David W. Aha.

(a) Original source: unknown

-- NASA: Mr. Roger Burke's autolander design team

(b) Donor: Bojan Cestnik

Jozef Stefan Institute

Jamova 39

61000 Ljubljana

Yugoslavia (tel.: (38)(+61) 214-399 ext.287)

(c) Date: November 1988

PAST USAGE: Several, it appears, such as [16].

RELEVANT INFORMATION: This is a tiny database. Michie reports that Burke's group used RULEMASTER to generate comprehensible rules for determining the conditions under which an autolanding would be preferable to manual control of the spacecraft.

NUMBER OF INSTANCES: 15

NUMBER OF ATTRIBUTES: 7 (including the class attribute)

ATTRIBUTE INFORMATION:

1. Class: noauto, auto

-- that is, advise using manual/automatic control

2. STABILITY: stab, xstab

3. ERROR: XL, LX, MM, SS

4. SIGN: pp, nn

5. WIND: head, tail

6. MAGNITUDE: Low, Medium, Strong, OutOfRange

7. VISIBILITY: yes, no

MISSING ATTRIBUTE VALUES:

--none

--but several "don't care" values: (denoted by "*")

Attribute Number:	Number of Don't Care Values:
2:	2
3:	3
4:	8
5:	8
6:	5
7:	0

CLASS DISTRIBUTION:

1. Use noauto control: 6

2. Use automatic control: 9

MODIFICATION: The author did the following changes to the database in order for it to be applicable for the neural network training. The "don't care" values were enumerated over all possible values, resulting in 248 patterns. After this, the pattern sequencing was randomized. This random set of patterns was fed to the learning system for training in the patterns file. The inputs to the network were the stability, error, sign, wind, magnitude and visibility attributes and the network tried to learn the class attribute (the output produced).

A.4 Iris Plants Database Problem

TITLE: Iris Plants Database

SOURCES: Obtained by anonymous ftp from ics.uci.edu from pub/machine-learning-databases directory. The UCI repository of machine learning databases is maintained by the site librarian, David W. Aha.

(a) Creator: R.A. Fisher

(b) Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)

(c) Date: July, 1988

PAST USAGE: Publications: Too many to mention. Here are a few: Fisher [11] in contributions to Mathematical Statistics, the pattern classification and scene analysis book by Duda and Hart [7]. Dasarathy [6] reports very low misclassification rates (0% for setosa class.) Gates also reports very low misclassification rates in [12]. Cheeseman et al. AUTOCLASS II conceptual clustering system [4] finds 3 classes in the data.

RELEVANT INFORMATION: This is perhaps the best known database to be found in the pattern recognition literature. Fisher's paper [11] is a classic in the field and is referenced frequently to this day. See Duda and Hart, [7] for example. The data set contains 3 classes of 50 instances each, where each class refers to a type of the iris plant. One class is linearly separable from the other 2; the latter are NOT linearly separable from each other. Predicted attribute: class of iris plant. This is an exceedingly simple domain.

NUMBER OF INSTANCES: 150 (50 in each of three classes)

NUMBER OF ATTRIBUTES: 4 numeric, predictive attributes and the class

ATTRIBUTE INFORMATION:

1. sepal length in cm
2. sepal width in cm

3. petal length in cm
4. petal width in cm
5. class:
   -- Iris Setosa
   -- Iris Versicolour
   -- Iris Virginica

MISSING ATTRIBUTE VALUES: None.

Summary Statistics					
	Min	Max	Mean	SD	Class Correlation
sepal length	4.3	7.9	5.84	0.83	0.7826
sepal width	2.0	4.4	3.05	0.43	-0.4194
petal length	1.0	6.9	3.76	1.76	0.9490
petal width	0.1	2.5	1.20	0.76	0.9565

The class correlation values for the petal length and petal width are rather high !

CLASS DISTRIBUTION: 33.3% for each of 3 classes.

A.5 Lymphography Domain Problem

TITLE: Lymphography Domain

SOURCES: Obtained by anonymous ftp from ics.uci.edu from pub/machine-learning-databases directory. The UCI repository of machine learning databases is maintained by the site librarian, David W. Aha.

(a) See below.

(b) Donors: Igor Kononenko,

University E.Kardelj

Faculty for electrical engineering

Trzaska 25

61000 Ljubljana (tel.: (38)(+61) 265-161

Bojan Cestnik

Jozef Stefan Institute

Jamova 39

61000 Ljubljana

Yugoslavia (tel.: (38)(+61) 214-399 ext.287)

(c) Date: November 1988

CITATION REQUEST: This lymphography domain was obtained from the University Medical Center, Institute of Oncology, Ljubljana, Yugoslavia. Thanks go to M. Zwitter and M. Soklic for providing the data. Please include this citation if you plan to use this database.

PAST USAGE: Several: Cestnik et al. report 76% accuracy using Assistant-86 in [3]. Clark et al. [5] report 83% accuracy using simple Bayesian learning. They also report 82% accuracy using CN2 (99% threshold.) Michalski et al. report 85% estimated accuracy in [15].

RELEVANT INFORMATION: This is one of three domains provided by the Oncology Institute that has repeatedly appeared in the machine learning literature. See also breast-cancer and primary-tumor available from the site.

NUMBER OF INSTANCES: 148

NUMBER OF ATTRIBUTES: 19 including the class attribute

ATTRIBUTE INFORMATION: All attribute values in the database have been entered as numeric values corresponding to their index in the list of attribute values for that attribute domain as given below.

1. class: normal find, metastases, malign lymph, fibrosis
2. lymphatics: normal, arched, deformed, displaced
3. block of affere: no, yes
4. bl. of lymph. c: no, yes
5. bl. of lymph. s: no, yes
6. by pass: no, yes
7. extravasates: no, yes
8. regeneration of: no, yes
9. early uptake in: no, yes
10. lym.nodes dimin: 0-3
11. lym.nodes enlar: 1-4
12. changes in lym.: bean, oval, round
13. defect in node: no, lacunar, lac. marginal, lac. central
14. changes in node: no, lacunar, lac. margin, lac. central
15. changes in stru: no, grainy, drop-like, coarse, diluted, reticular, stripped, faint
16. special forms: no, chalices, vesicles
17. dislocation of: no, yes

18. exclusion of no: no, yes

19. no. of nodes in: 0-9, 10-19, 20-29, 30-39, 40-49, 50-59, 60-69, >=70

MISSING ATTRIBUTE VALUES: None

CLASS DISTRIBUTION:

Class:	Number of Instances:
normal find:	2
metastases:	81
malign lymph:	61
fibrosis:	4

APPENDIX B

PROGRAMS AND SAMPLE DATA FILES LISTING

B.1 Program Listing

In the appendix, we list all the programs that were used in this project. In Figure B.1, we provide a flow chart of the various programs and their interconnection. Here is a copy of the manual page for these programs, converted to a format acceptable to the graduate college.

NAME

overall – Execute resource constrained design experiments for training Artificial Neural Networks.

DESCRIPTION

This set of programs is used to execute Type I, Order and Type II experiments for training Artificial Neural Networks (ANN) under pre-specified resource constraints. Details on these experiments for training ANN can be found in my M.S. Thesis. The lisp function (overall) defined in overall.lsp is the function which executes the programs. The function type1() is the main routine for implementing Type I experiments. order() is the routine for ordering configurations produced by Type I experiments. Finally function.i is the lisp code for Type II experiments. These functions should be executed strictly in this order, since the results produced by one are used by others. Before the programs can be compiled and executed, we need to specify a set of parameters, a set of functions and a set of input files. Details on setting up the parameters can be found in PARAMETERS. Details on setting up various functions and input files can be found in FUNCTIONS and INPUT FILES, respectively. The programs can be compiled and executed following the steps given in STEPS. These programs can be tested on the sample

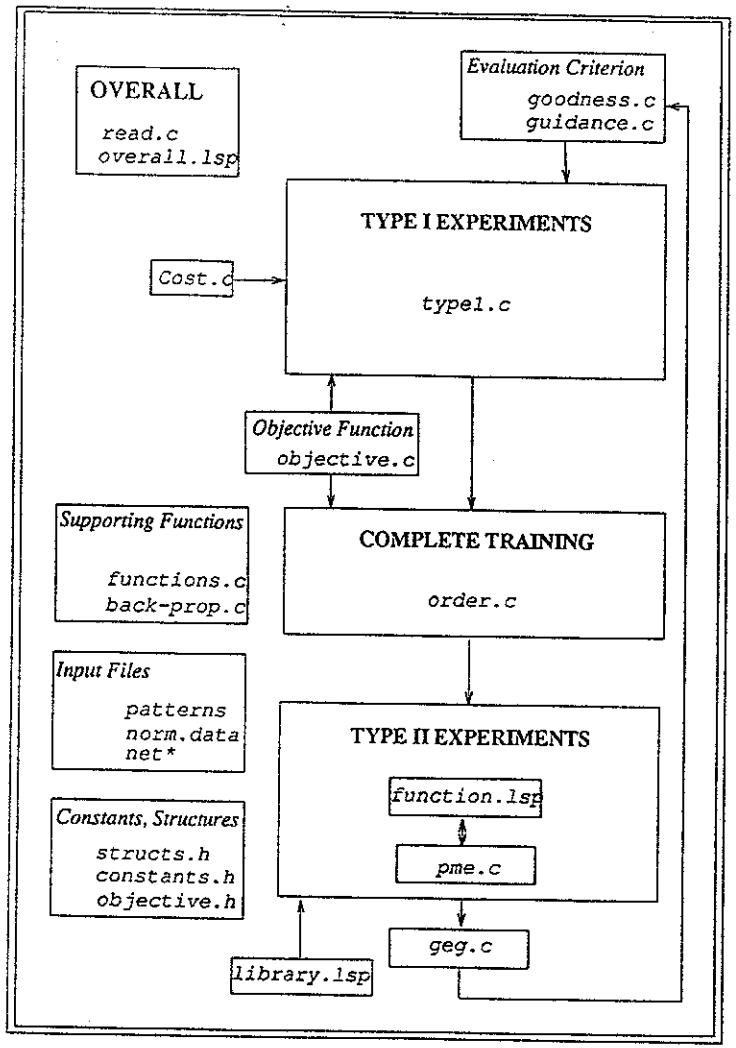

Figure B.1: Flow chart of various programs

benchmark problems described in EXAMPLE RUNS. To interpret the results obtained, please see OUTPUT FILES.

STEPS

Follow these steps to execute the supplied programs for a given application:

(1) Specify various parameters of the system in constants.h and objective.h files in /home/rhea/software/kriplani/type1 directory. (see PARAMETERS for details on how to do it.)

(2) Specify the expected goodness (geg.c), cost (functions.c) and guidance functions (guidance.c) in /home/rhea/software/kriplani/type1. (see FUNCTIONS for details.)

(3) cd /home/rhea/software/kriplani.

(4) Compile the programs by typing "make compile" at the console.

(5) cd overall.

(6) cd into encod4 or twosp to execute the programs on the benchmark problems or set up your own directory for a new problem. Specify various input files, as explained in INPUT FILES.

(7) Following command will run the program and produce appropriate outputs.

nice +20 cl <../overall.i >& debug &

(8) See OUTPUT FILES for interpreting the results.

PARAMETERS

One needs to specify various system level parameters before the programs can be compiled. These system level parameters are specified in constants.h file. Parameters used in the objective function are specified in objective.h file. The system level parameters specified in constants.h file are:

max_layers: (Used for allocating space in data structures) Maximum number of layers in any network. Pick up a network with the maximum number of layers from the library of networks and set this parameter equal to the number of layers in it.

max_weights: (Used for allocating space in data structures) Maximum number of weights and biases in any

network. Value of this parameter should be

$$\geq \sum_{i=0}^{(no\_layers-1)} (no\_units[i]+1) \times no\_units[i+1], \qquad \text{for } \textit{all the networks considered.}$$

max_units: (Used for allcoating space in data structures) Maximum number of units in any network.

max_patterns: (Used for allocating space in data structures) Maximum number of input-output training patterns.

max_pat_length: (Used for allocating space in data structures) Maximum length of any input-output training pattern. Note that all the patterns have the same lengths.

quantum: Quantum is a fixed number of epochs, to be spent on a configuration before updating its goodness and guidance function values. Value of quantum should be such that the goodness and guidance function values change significantly; not much time is wasted on unpromising configurations before the decision to stop training it is made. (Recommended range: quantum should be one-fifth to one-hundredth of the average training time.)

max_quanta: Should be greater than or equal to the norm_time (the normalization factor for time.). Used for allocation of space in data structures.

window_tsse: TSSE is averaged over this moving window. Should be smaller than quantum. For the problems tested, (when momentum factor was small) 5 was a good value found (by looking at the first derivative curves).

window_slope: First derivative (slope) of the learning curve is averaged over these many past values, to smoothen it out. A good guess = 2 or 3.

window_sec_der: Again, as before, the second derivative of the learning curve is smoothened over these many past values. A good guess = 2 or 3. Please note that (window_tsse + window_slope + window_sec_der) < quantum.

network_window_size: Number of networks to try at any given instant of time. Window size represents a trade-off between the memory space and time. Typically should be between 2-100. Represents number of alternatives one would like to keep at any instant in Type I training.

NO_NETS_IN_PON: Total number of library networks.

ORDER_MAX_NETS: In Type I experiments, we save the first ORDER_MAX_NETS and pass them on to the order function, for complete training. Greater the number, better the accuracy of the function generator. One could try 10 or more networks for complete training, depending upon their training times and the resources specified.

MIN_OBJECTIVE: Replace the objective function values of all the networks which don't converge by this value. Should be equal to the objective function value of the most complex network (maximum cost) which converges in norm_time.

worst_cost: worst_cost is an upper bound on the cost of networks considered. Any loose bound will do the job, since it is used only to compute UBF for networks in the beginning, when none of the networks have converged.

debug: This bit when set to 1, forces the programs to save intermediate results about each experiment. In particular, following files are created: type1, net_stat, conv, no_conv, perfm. See OUTPUT FILES for interpreting the results of these files.

function_debug: When this bit is set to 1, the following file is created: evaluation.

order_debug: When this bit is set to 1, training traces for all the configurations supplied to order program are saved in "trace(network_no)" files.

initial_amp_att: (Used in function generator program, function.lsp) It is the initial amplification or attenuation (amp_att) coefficient. Coefficients of expected goodness function are amplified or attenuated by (1 + amp_att) at any time. Typically around 0.9.

reduction_factor: (Used in function generator program, function.lsp) It is the factor by which the amp_att (amplification or attenuation coefficient) is reduced, after a local minimum for the current function is reached. Typically around 1.5.

add_term_after_no_reductions: (Used in function generator program, function.lsp) Add a new term to the current function after amp_att has been reduced add_term_after_no_reductions number of times. Typically around 8.

X, Y, Z: X, Y, Z are the three coefficient of the IS. (see Equation (4.9) in the thesis.) Typically X=10.0, Y=1.0, Z=1.0.

no_cycles: A step through "type1 - order - type2" experiments is considered as one cycle. Specify the number of times one would like to repeat this cycle.

type1_duration_in_quanta: The duration of Type I experiment in number of quanta.

no_of_functions_to_try: (Used in function generator program, function.lsp) Number of functions that should be generated and tested in Type II experiment before the experiment is terminated.

initial_no_of_terms_in_goodness: Specifies the (starting) expected goodness function in Type II experiments. Expected goodness function = fixed goodness + a fixed number of terms from the library of

terms. (library of terms is specified in overall/library.lsp). The number of library terms to include is specified by this parameter. Fixed goodness is defined in function.lsp and is equal to 10.0 * (1.0 - tsse).

objective.h file contains various objective function parameters. These are user specified parameters. Sample values are provided in objective.h file. Any objective function can be supplied by the user. However, in the thesis, for uniformity of all the figures, I tried to choose an objective function which produced values between 0 and 10, for most of the configurations considered. The values of the constants were chosen as follows:

A: was set to 10.0

B: was set to 8.0 / norm_time. norm_time is the normalization factor for time, as specified in the input file norm.data. See INPUT FILES.

C: was set to 2.0 / norm_cost. norm_cost is the normalization factor for cost, as specified in the input file norm.data. See INPUT FILES.

FUNCTIONS

Here, we describe how and where to specify various functions such as guidance function, cost function and the expected goodness function.

Guidance function: This function is supplied in guidance.c file in /home/rhea/software/kriplani/type1 directory. The function is heuristically decided and is not learned by the learning experiments, at this point.

Cost function: This is the heuristic cost function specified in functions.c file in /home/rhea/software/kriplani/type1 directory.

Expected goodness function: This function is specified in /home/rhea/software/kriplani/type1/geg.c file.

This function is learned by the Type II learning experiment. An initial guess for this function is required to initiate Type I experiments. (1.0 - tsse) is a smart initial guess for this function.

INPUT FILES

Various input files which need to be set up at the start of the experiments are: norm.data, patterns, and various library networks specified in "net0", "net1", "net2" etc files. Formats of these files are as follows:

norm.data : Contains various normalization constants.

Normalization factor for the cost: Typically 3 * Average cost.

Normalization factor for TSSE: Typically 1.5 * No of patterns * No of output units * 0.25.

Normalization factor for the Training Time: Typically 3 * Average training time.

Network Specification:

Number of layers (excluding input layer).

Number of units in each layer (including input layer).

ecrit

Number of quanta to be spent on the network before updating its goodness and guidance values. (=1)

learning rate constant

momentum factor

wrange (range of initial weights)

newstart (=1 for a network)

patterns: Input-output training patterns.

Number of patterns

Length of each pattern

Input-Output Patterns

OUTPUT FILES

Output produced by the type1 program is explained below. The best objective function value obtained at the end of Type I experiment is saved in "final" file. The format of this file is:

Type I Duration Best Objective Library Network No.

Whenever debug bit is set to 1, the following additional files are saved: The execution trace of the experiments is saved in "type1" in the following format:

File No TSSE Quanta Spent Goodness Guidance Library Net No

Final information about each and every configuration considered in the experiments is saved in "net_stat" file, in the following format:

Lib Net No Net No Goodness Guidance Quanta TSSE Objective

The output of the order() function is as follows: No output is produced if the debug and order_debug bits are set to 0. Whenever debug = 1, the objective function values and the training times for all the networks which converged along with their network numbers are saved in "conv" file. This file is sorted based upon the training times. The same information about networks which didn't converge is saved in "no_conv" file. The format of these files are:

Network Number Objective Function Value Training Time

Whenever order_debug = 1, individual traces about each and every configuration considered are saved in files "trace0", "trace1", "trace2" etc. The format of these files is:

Average Tsse Average Slope Average Second Derivative Time

The output produced by pme() function is IS, the inconsistency sum, as defined by the equation 4.9 in the thesis. This value is returned by this function whenever it is executed. Whenever debug = 1, an additional file "perfm" is saved. The format of this file is:

IS

Sum of monotonic inconsistencies

Sum of inconsistency counts

Sum of inconsistency differences

The output produced by the function generator is the expected goodness function. An addition file "evaluation" is generated whenever function_debug = 1. This file contains the execution trace of the function generator. It saves the performance (IS) of the various functions generated as follows:

function_no inconsistency_sum

Additional information about the execution like when a new term is added to the expected goodness function ("New term added."), and when a better expected goodness function (better than the existing one) is found ("Better expected goodness function found.") is also saved.

EXAMPLE RUNS

Sample files are provided in /home/rhea/software/kriplani/overall. Examples include two-spiral benchmark problem (twosp) and the encode-decode problem with 4 bits (encod4). See the Appendix A (Benchmark problems) of the thesis for more detail on these benchmarks. The directories in which sample files for these two benchmarks are provided are twosp and encod4 respectively. Follow the steps given in STEPS to run the programs.

FILES

/home/rhea/software/kriplani/type1/back-prop.c	The back propagation algorithm.
/home/rhea/software/kriplani/type1/constants.h	Various constants of the system.
/home/rhea/software/kriplani/type1/cost.c	A program to find out the cost of a network.
/home/rhea/software/kriplani/type1/functions.h	Supplementary functions needed by type1.c and order.c.
/home/rhea/software/kriplani/type1/geg.c	The expected goodness function.
/home/rhea/software/kriplani/type1/goodness.c	The goodness function.
/home/rhea/software/kriplani/type1/guidance.c	The guidance function.
/home/rhea/software/kriplani/type1/structs.h	Various type definitions.
/home/rhea/software/kriplani/type1/type1.c	The Type I routine.
/home/rhea/software/kriplani/type2/objective.c	The objective function.
/home/rhea/software/kriplani/type2/objective.c	Parameters of the objective function.
/home/rhea/software/kriplani/type2/order.c	Main routine to order configurations.

/home/rhea/software/kriplani/type2/pme.c	The performance measuring element.
/home/rhea/software/kriplani/type2/lisp/function.lsp	The function generator.
/home/rhea/software/kriplani/overall/library.lsp	File containing library of terms.
/home/rhea/software/kriplani/overall/overall.lsp	The overall program for the system.
/home/rhea/software/kriplani/overall/read.c	Functions for reading in data about the whole system.

SEE ALSO

Various files mentioned in FILES. Another file /home/rhea/software/kriplani/overall/test.c might be helpful.

AUTHOR

Harish Kriplani

BUGS

NONE TO DATE. Please report any bugs to kriplani@aquinas.csl.uiuc.edu.

type1.c

```
/* **********  type1.c  ********** */

/* This program implements TYPE I experiments */

/* Function type1 simulates the type1 experiments. Various parameters taken
  by this function are explained below:

  total_time: total duration of type1 experiments.
  pat: structure containing input output patterns.
  norm: structure containing normalization constants.
  pon: structure containing specifications about the pool of networks.
  iwts: structure in which various initial weight configurations are returned
        for order.c program.

  The output of the program consists of the following:
   The best objective function value obtained at the end of the experiments
   is saved in "final" file. The format of the file is:
      Type1_duration  Best_objective  Library_net_no
  Whenever debug bit is set to 1, the following additional information is
  saved:
   The execution trace of the experiments, in the format: in "type1"
     File No  TSSE  Quanta_spent  Goodness  Guidance  Lib_net_no

   Final information about each and every configuration considered in the
   experiment is saved in "net_stat" file, in the following format:
    Lib_Net_no  Net_no  Goodness  Guidance  quanta  tsse  objective
*/


#include <stdio.h>
#include <math.h>
#include <sys/types.h>
#include <sys/timeb.h>
#include "constants.h"
#include "structs.h"
#include "../type2/objective.h"

#define  worst_objective      -1000000.0

void  type1();
void  BP();
struct goodness_list_struct *insert_guide();
```



```
struct goodness_list_struct *insert_good();
void  Print();
void  print_good();
int   random_net_no();
void  print_stat();
int   UBF();


extern float  Cost();
extern float  av_tsse();
extern void   bp();
extern void   copy_bp_data();
extern float  goodness();
extern float  guidance();
extern float  objective();
extern float  random_no();
extern void   read_bp_data();
extern float  second_derivative();
extern float  slope();
extern void   write_bp_data();

extern long   seed;



void type1(total_time, pat, norm, pon, iwts)
int     total_time;
  /* Total duration of the Type I experiment, in number of quanta */
struct  pat_struct    *pat;
struct  norm_struct   *norm;
struct  pon_struct    *pon;
struct  configs_tried *iwts;
{
  int   npatterns = (pat -> npatterns);
  float best_objective = worst_objective; /* obj. of the best network */
  int   best_quanta = max_quanta ; /* quanta spent in training it ! */
  float best_cost = worst_cost;    /* cost of this network ! */
  int   best_lib_no = 0;   /* lib net no. of this net */
  float net_cost;       /* Actual cost of a network. */
  int   spent_nquanta;  /* Quanta spent on a network. */
  float net_objective;  /* objective function value of a network. */
  int   lib_net_no;     /* Library network number */
  int   Next_file_no = 0; /* Next file no to be assigned */
  int   add_new_net = 0;    /* 0 = don't add, 1 = add */
  struct bp_data_struct bp_data[network_window_size + 1], *junk_bp_data ;
  struct goodness_list_struct *root, *node;
  struct goodness_list_struct *trained_root;
       /* Maintain a list of nets which are trained to completion */
  struct goodness_list_struct *abort_root; /* circuits aborted since
```

```
                they didn't converge in specified max time */
int   net_no ;  /* network number */
int   best_net ; /* best network number */
int   i, j ;
int   no_inputs_outputs ;  /* Number of inputs and outputs */
FILE  *fptrace, *fpop, *fopen();
int   spent_time = 0 ;
float norm_cost = norm -> norm_cost ;
float norm_tsse = norm -> norm_tsse ;
int   norm_time = norm -> norm_time ;

junk_bp_data = (struct bp_data_struct *) malloc(sizeof(*junk_bp_data)) ;
seed = time(0) ;
srand48(seed);
  /* *** FILL IN THE WINDOW WITH CIRCUITS TO BE TRAINED *** */
for (net_no = 0 ; net_no < network_window_size ; net_no++) {
  lib_net_no = random_net_no((pon -> no_nets_in_pon) -1) ;
  copy_bp_data(&(pon -> nets[lib_net_no]), &bp_data[net_no]) ;
}
if (debug == 1) {
  fptrace = fopen("type1", "w") ;
  fprintf(fptrace,
   "File No  TSSE  Quanta spent  Goodness  Guidance  Lib net no \n") ;
}
  /* *** TRAIN ALL CKTS FOR ONE TIME QUANTUM EACH *** */
root = NULL ;
for (net_no =0 ; net_no  < network_window_size ; net_no++) {
       /* *** Save initial weights for ORDER_MAX_NETS configs only *** */
 if ((iwts -> no_nets) >= ORDER_MAX_NETS) {
   BP(&bp_data[net_no ], npatterns, junk_bp_data, (pat -> patterns),
                              norm_cost, norm_tsse, norm_time) ;
 }
 else {
   BP(&bp_data[net_no ], npatterns, &(iwts -> nets[iwts -> no_nets]),
                 (pat -> patterns), norm_cost, norm_tsse, norm_time) ;
   (iwts -> no_nets) = 1 + (iwts -> no_nets) ;
 }
 node = (struct goodness_list_struct *) malloc(sizeof(*node)) ;
 node -> net_no = net_no  ;
 node -> file_no = Next_file_no ;
 ++Next_file_no ;
 node -> guidance = bp_data[net_no ].guidance ;
 node -> goodness = bp_data[net_no ].goodness ;
 node -> tsse = bp_data[net_no].tsse ;
 node -> spent_nquanta = bp_data[net_no].spent_nquanta ;
 node -> lib_net_no = bp_data[net_no].lib_net_no ;
 root = Insert_guide(root, node) ;
 print_stat(node, fptrace) ;
 ++spent_time ;
```



```
}   /* All circuits have been trained for one time quantum each and
          their Goodness and Guidance values have been calculated */
abort_root = NULL ;
trained_root = NULL ;
while (spent_time < total_time) {
 best_net = root -> net_no ;
 node = root ;
 root = root -> next ;
 ++spent_time ;
 BP(&bp_data[best_net], npatterns, junk_bp_data, (pat -> patterns),
                              norm_cost, norm_tsse, norm_time) ;
 node -> guidance = bp_data[best_net].guidance ;
 node -> goodness = bp_data[best_net].goodness ;
 node -> tsse = bp_data[best_net].tsse ;
 node -> spent_nquanta = bp_data[best_net].spent_nquanta ;
 node -> lib_net_no = bp_data[best_net].lib_net_no ;
 print_stat(node, fptrace) ;
 if ((bp_data[best_net].spent_nquanta >= bp_data[best_net].ubf) &&
       (bp_data[best_net].tsse > bp_data[best_net].ecrit)) {
       abort_root = Insert_good(abort_root, node) ;
       add_new_net = 1 ;
 }
 else {
  if (bp_data[best_net].tsse > bp_data[best_net].ecrit) {
    root = Insert_guide(root, node) ;
  }
  else {
        trained_root = Insert_good(trained_root, node) ;
        add_new_net = 1 ;
         /* A circuit has been trained */
        net_cost = Cost(&bp_data[best_net]) ;  /* actual cost */
        spent_nquanta = bp_data[best_net].spent_nquanta ;
        net_objective = objective(spent_nquanta, net_cost) ;
        node -> objective = net_objective ;
        if (net_objective > best_objective) {
          best_objective = net_objective ;
          best_quanta = bp_data[best_net].spent_nquanta ;
          best_cost = Cost(&bp_data[best_net]) ;
          best_lib_no = bp_data[best_net].lib_net_no ;
          /* Compute ubf of all the nets in the window */
          for (i = 0 ; i < network_window_size ; i++) {
                net_cost = Cost(&bp_data[i]) ;
                j = UBF(best_quanta, best_cost, net_cost) ;
                if (j < norm_time) bp_data[i].ubf = j ;
          }
        }
  }
}
  /* *** ADD ANOTHER CIRCUIT TO THE WINDOW *** */
```

```
  if (add_new_net == 1) {
    add_new_net = 0 ;
    lib_net_no = random_net_no((pon -> no_nets_in_pon) -1) ;
    copy_bp_data(&(pon -> nets[lib_net_no]), &bp_data[best_net]) ;
    ++spent_time ;
          /* *** Save initial weights for ORDER_MAX_NETS configs only *** */
    if ((iwts -> no_nets) >= ORDER_MAX_NETS) {
      BP(&bp_data[best_net], npatterns, junk_bp_data,
        (pat -> patterns), norm_cost, norm_tsse, norm_time) ;
    }
    else {
      BP(&bp_data[best_net], npatterns, &(iwts -> nets[iwts -> no_nets]),
        (pat -> patterns), norm_cost, norm_tsse, norm_time) ;
      (iwts -> no_nets) = 1 + (iwts -> no_nets) ;
    }

    node = (struct goodness_list_struct *) malloc(sizeof(*node)) ;
         node -> net_no = best_net ;
         node -> file_no = Next_file_no ;
         ++Next_file_no ;
    node -> guidance = bp_data[best_net].guidance ;
    node -> goodness = bp_data[best_net].goodness ;
    node -> tsse = bp_data[best_net].tsse ;
    node -> spent_nquanta = bp_data[best_net].spent_nquanta ;
    node -> lib_net_no = bp_data[best_net].lib_net_no ;
    root = Insert_guide(root, node) ;
         print_stat(node, fptrace) ;
           /* compute its ubf */
    net_cost = Cost(&bp_data[best_net]) ;
         j = UBF(best_quanta, best_cost, net_cost) ;
         if (j < norm_time) bp_data[best_net].ubf = j ;
  }
}
  /* *** SAVE STATISTICS ABOUT THE BEST NETWORK FOUND. *** */
fpop = fopen("final", "w") ;
if (best_objective != worst_objective) {
  fprintf(fpop, "Type1_duration Best_objective Library_net_no \n") ;
  fprintf(fpop, "%d   %f   %d \n", spent_time, best_objective, best_lib_no) ;
}
else
  fprintf(fpop, "No network converged in %d quanta. \n", spent_time) ;
fclose(fpop) ;
if (debug == 1) {
  fpop = fopen("net_stat", "w") ;
  fprintf(fpop,
"Lib_Net_no  Net_no  Goodness  Guidance  quanta  tsse  objective \n");
  if (trained_root != NULL)
         fprintf(fpop, "\nFOLLOWING NETWORKS CONVERGED: \n") ;
  while (trained_root != NULL) {
```



```
    fprintf(fpop, "%d   %d   %f   %f   %d   %f   %f \n",
                    (trained_root -> lib_net_no),
                    (trained_root -> file_no),
                    (trained_root -> goodness),
                    (trained_root -> guidance),
                    (trained_root -> spent_nquanta),
                    (trained_root -> tsse),
                    (trained_root -> objective)) ;
    trained_root = trained_root -> next ;
  }
  fprintf(fpop, "\n") ;
  if (root != NULL)
         fprintf(fpop,
           "FOLLOWING NETWORKS COULD NOT BE TRAINED TO COMPLETION: \n") ;
  while (root != NULL) {
    fprintf(fpop, "%d   %d   %f   %f   %d   %f   - \n",
                    (root -> lib_net_no),
                    (root -> file_no),
                    (root -> goodness),
                    (root -> guidance),
                    (root -> spent_nquanta),
                    (root -> tsse)) ;
    root = root -> next ;
  }
  fprintf(fpop, "\n") ;
  if (abort_root != NULL)
         fprintf(fpop, "FOLLOWING NETWORKS WERE ABORTED: \n") ;
  while (abort_root != NULL) {
    fprintf(fpop, "%d   %d   %f   %f   %d   %f   - \n",
                    (abort_root -> lib_net_no),
                    (abort_root -> file_no),
                    (abort_root -> goodness),
                    (abort_root -> guidance),
                    (abort_root -> spent_nquanta),
                    (abort_root -> tsse)) ;
    abort_root = abort_root -> next ;
  }
  fclose(fpop) ;
 }
 if (debug == 1)
  fclose(fptrace) ;
}



/* THE BACK-PROPAGATION ALGORITHM SUPPLEMENTARY BLOCK. */

void BP(bp_data, npatterns, iwts_data, patterns, norm_cost, norm_tsse, norm_time)
struct bp_data_struct *bp_data, *iwts_data ;
```

```
int    npatterns ;
float  patterns[max_patterns][max_pat_length] ;
float  norm_cost, norm_tsse ;
int    norm_time ;
{
  float  trace[quantum] ;
  int    spent_time ;
  float  cost1 ;
  float  av_tsse1 ;
  float  slope1 ;
  float  sec_der ;
  int    i ;
  float  ecrit = (bp_data -> ecrit) ;

  bp(bp_data, npatterns, patterns, trace, iwls_data) ;
        /* Now, calculate goodness and guidance values */
  cost1 = Cost(bp_data) ;
  av_tsse1 = av_tsse((quantum-1), trace) ;
    /* av_tsse value at the end of the time quantum (wherever we stop) */
  slope1 = slope((quantum - 1), trace) ;
  sec_der = second_derivative((quantum - 1), trace) ;
  spent_time = (bp_data -> spent_nquanta) ;
  if ((bp_data -> tsse) < ecrit) {
   av_tsse1 = ecrit ;
   slope1 = 0.0 ;
   sec_der = 0.0 ;
        /* After convergence, don't want these factors to bias
          my decision */
  }
  bp_data -> goodness = goodness(cost1, av_tsse1, slope1, sec_der, spent_time,
                                 norm_time, norm_tsse, ecrit) ;
  bp_data ->guidance = guidance((bp_data ->goodness), spent_time, norm_time) ;
}


struct goodness_list_struct *Insert_good(root, node)
struct goodness_list_struct *root, *node ;
{
  struct  goodness_list_struct  *ptr, *prev_ptr ;
  if ((root == NULL) || ((node -> goodness) > (root -> goodness))) {
   node -> next = root ;    /* Insert in the beginning of the list */
   root = node ;
  }
  else {
   ptr = root ;
   while ((ptr -> goodness) >= (node -> goodness)) {
        prev_ptr = ptr ;
        ptr = (ptr -> next) ;
        if (ptr == NULL)  break ;
```



```
   }
   (node -> next) = (prev_ptr -> next) ;
   (prev_ptr -> next) = node ;
  }
  return(root) ;
}


struct goodness_list_struct *Insert_guide(root, node)
struct goodness_list_struct *root, *node ;
{
  struct  goodness_list_struct  *ptr, *prev_ptr ;
  if ((root == NULL) || ((node -> guidance) > (root -> guidance))) {
   node -> next = root ;    /* Insert in the beginning of the list */
   root = node ;
  }
  else {
   ptr = root ;
   while ((ptr -> guidance) >= (node -> guidance)) {
        prev_ptr = ptr ;
        ptr = (ptr -> next) ;
        if (ptr == NULL)  break ;
   }
   (node -> next) = (prev_ptr -> next) ;
   (prev_ptr -> next) = node ;
  }
  return(root) ;
}


void Print(root)
struct goodness_list_struct *root ;
{
  struct goodness_list_struct *ptr ;
  ptr = root ;
  while (ptr != NULL) {
   printf("%d  %f  \n", (ptr -> net_no), (ptr -> goodness)) ;
   ptr = (ptr -> next) ;
  }
}


void  print_good(root)
struct goodness_list_struct *root ;
{
  while (root != NULL) {
   printf("                %d %f \n",
                                  root -> net_no, root -> goodness) ;
   root = root -> next ;
```

```
    }
}


int random_net_no(range)  /* Produce a random number between 0 and range */
int range ;
{
   float temp1, temp2 ;
   int   number ;
   ++range ;
   temp1 = (float) range ;
   temp2 = random_no(temp1) ;
   if (temp2 < 0) temp2 = - temp2 ;
   number = (int) temp2 ;
   return(number) ;
}



void print_stat(NODE, fpop)
struct goodness_list_struct *NODE ;
FILE  *fpop ;
{
  if (debug == 1) {
    fprintf(fpop, "  %d   %f     %d       %f %f    %d\n",
                                (NODE -> file_no),
                                (NODE -> tsse),
                                (NODE -> spent_nquanta),
                                (NODE -> goodness),
                                (NODE -> guidance),
                                (NODE -> lib_net_no)) ;
  }
}


int UBF(best_quanta, best_cost, current_cost)
int    best_quanta ;
float  best_cost, current_cost ;
{
 int  temp ;
 temp = best_quanta + (int) (C * (best_cost - current_cost)) / (B) ;
 return(temp) ;
}
```


structs.h

```
/* **********  structs.h  ********** */

/* Header file for type definitions. */


/* Contains important information about each configuration generated. */

struct goodness_list_struct {
  int    net_no ;       /* Network number in the window */
  int    file_no ;      /* file no, in which data about the net are saved */
  float  guidance ;
  float  goodness ;
  float  objective ;
  float  tsse ;
  int    spent_nquanta ;    /* time spent on the net */
  int    lib_net_no ;       /* parent library network, its a config of */
  struct goodness_list_struct *next ;
} ;

/* Data Structure for network configurations. */

struct bp_data_struct {
  int    nlayers ;   /* Number of layers */
  int    nquanta ;   /* next time train for 'nquanta' quanta */
  float  tsse ;      /* tsse value at the end of the last training */
  float  ecrit ;     /* stop training when tsse < ecrit */
  float  lrate ;     /* learning rate constant */
  float  momentum ;  /* momentum constant */
  float  wrange ;    /* choose weights between -wrange and +wrange */
  int    no_units[max_layers] ;  /* no of units per layer */
  int    newstart ;  /* =1 means weights are not specified. */
  float  activation[max_units] ; /* Save activations of the units, before
                                    stopping this training session. */
  float  weights[max_weights] ;  /* weights and biases */
  int    spent_nquanta ;        /* amount of quanta previously spent */
  float  goodness ;  /* goodness value of net at the end of spent_nquanta */
  float  guidance ;  /* guidance value of net at the end of spent_nquanta */
  int    ubf ;   /* no of quanta to be spent on the net in the worst case */
  int    lib_net_no ;  /* Network number, from pon */
} ;
```

```
/* Data Structure for arranging configurations. */

struct order_obj_struct {
  int    net_no ;
  float  objective ;
  int    traintime ;   /* Training time */
  struct order_obj_struct *next ;
} ;

/* Data structure for reading in training patterns. */

struct pat_struct {
  int   npatterns ;
  int   no_inputs_outputs ;
  float patterns[max_patterns][max_pat_length] ;
} ;

/* Data structure for normalization constants. */

struct norm_struct {
  float   norm_cost ;   /* Normalization constant for cost */
  float   norm_tsse ;   /* Normalization constant for tsse */
  int     norm_time ;   /* Normalization constant for time */
} ;

/* Data structure for reading in the library networks. */

struct pon_struct {
  int   no_nets_in_pon ;
  struct bp_data_struct nets[NO_NETS_IN_PON] ;
} ;


/* configs_tried stores complete information about all the configurations
   tried during type1 experiment. Information includes their weights, costs,
   training traces, goodness values at the end of each time quantum. Also,
   there are two linked list included in this structure, which informs which
   networks converged in norm_time (converged) and which didn't
   (not_converged). */

struct configs_tried {
  int    no_nets ;  /* Total number of networks tried */
  struct bp_data_struct nets[ORDER_MAX_NETS] ;  /* Configurations */
  float  cost[ORDER_MAX_NETS] ;   /* associated costs */
  float  tsse[ORDER_MAX_NETS][max_quanta] ;  /* training traces */
  float  slope[ORDER_MAX_NETS][max_quanta] ;
  float  sec_der[ORDER_MAX_NETS][max_quanta] ;
  int    int_time[ORDER_MAX_NETS][max_quanta] ; /* intermediate time */
  float  goodness[ORDER_MAX_NETS][max_quanta] ; /* goodness values */
```



```
  struct order_obj_struct *converged ;  /* linked list for nets which
                                           converged */
  struct order_obj_struct *not_converged ;  /* linked list for nets which
                                               didn't converge */
} ;
```

TYPE I makefile

```
BIN=back-prop geg goodness guidance functions cost type1

all:${BIN}

FLAGS= -O2

back-prop: back-prop.c constants.h structs.h
	cc ${FLAGS} -c back-prop.c -lm

geg: geg.c
	cc ${FLAGS} -c geg.c -lm

goodness: goodness.c
	cc ${FLAGS} -c goodness.c -lm

guidance: guidance.c
	cc ${FLAGS} -c guidance.c -lm

functions: functions.c constants.h structs.h
	cc ${FLAGS} -c functions.c -lm

cost: cost.c functions.c constants.h structs.h
	cc ${FLAGS} -o cost functions.o cost.c -lm

type1: type1.c constants.h structs.h ../type2/objective.h
	cc ${FLAGS} -c type1.c -lm

clean:
	rm -f *.o core ${BIN}
```


guidance.c

```
/* **********  guidance.c  ********** */

/* This program implements the heuristic for the Guidance function.
   Parameters for this function are:

   goodness: the goodness value of the configuration.
   time: no of quanta spent in training so far.
   norm_time: normalization constant for time   */


#include <stdio.h>
#include <math.h>

float guidance(goodness, time, norm_time)
float goodness ;
int  time ;
int  norm_time ;
{
   float  temp ;
   float  ntime ; /* normalized time */
   ntime = ((float) time) / ((float) norm_time) ;

         /* Guidance function heuristic */
   temp = goodness - 5.0 * ntime + 0.05 / ntime ;
   return(temp) ;
}
```

goodness.c

```
/* **********  goodness.c  ********** */

/* This Program implements the goodness function. It takes the following
   parameters to return the goodness function value for a configuration:

   tsse: total sum of square of error, at the end of training.
   slope: value of the slope of the learning curve.
   second derivative: value of the second derivative of the learning curve.
   cost: cost of the configuration.
   quanta spent: number of quanta spent in training the configuration.
   norm_time: normalization constant for time.
   norm_tsse: normalization constant for tsse.
   ecrit: critical error value. Network trained when tsse < ecrit.

   Goodness function =  alpha * geg function - penalty function.
   Since goodnes function is used only for comparison in type I experiments,
   multiplying it by a constant does not affect its role. */


#include <stdio.h>
#include "../type2/objective.h"

extern float geg() ;

float goodness(cost, tsse, slope, sec_der, t, norm_time, norm_tsse, ecrit)
float   cost, tsse, slope, sec_der ;
int     t, norm_time ;
float   norm_tsse ;
float   ecrit ;
{
   float gdness ;
   float alpha ;
   float Aprime ;
   float Cprime ;
   float Geg ;
   float penalty ;
   float ntsse, nslope, nsec_der ; /* normalized parameters */

   Aprime = (A / (B * norm_time)) ;
   ecrit = ecrit / norm_tsse ;  /* normalize ecrit */
   alpha = Aprime / geg(ecrit, 0.0, 0.0) ;
```



```
   ntsse = tsse / norm_tsse ;
   nslope = slope * norm_time / norm_tsse ;
   nsec_der = slope * norm_time * norm_time / (norm_tsse * norm_tsse) ;
   Geg = geg(ntsse, nslope, nsec_der) ;
   Cprime = (C / (B * norm_time)) ;
   penalty = (t/norm_time) + Cprime * cost ;
   gdness = alpha * Geg - penalty ;
     /* to make gdness = obj, after convergence, multiply by a const */
   gdness = gdness * B * norm_time ;
   return(gdness) ;
}
```

geg.c

```
/* SD -> second derivative */

#include <stdio.h>

float geg(TSSE, SLOPE, SD)
float TSSE, SLOPE, SD ;
{
  float gdness ;
  gdness = (10.0 * (1.0 - TSSE) - 6.1322184 * SLOPE) ;
  return(gdness) ;
}
```


functions.c

```
/* **********  functions.c  ********** */

/* Supplementary Functions File: Contains some of the common functions required
  by type1.c, order.c and other programs. */


#include <stdio.h>
#include <math.h>
#include "constants.h"
#include "structs.h"


float av_tsse() ;
float slope() ;
float second_derivative() ;
float Cost() ;


float av_tsse(time, trace)
int  time ;
float trace[] ;
{
  int  i ;
  float av_tsse1 = 0.0 ;

  for (i=0 ; i <= (window_tsse - 1) ; i++) {
    av_tsse1 += trace[time - i] ;
  }
  av_tsse1 = av_tsse1 / ((float) window_tsse) ;
  return(av_tsse1) ;
}

float slope(time, trace)
int  time ;
float trace[] ;
{
  float slope1 ;

  slope1 = (av_tsse(time, trace) -
            av_tsse((time - window_slope), trace)) / ((float) window_slope) ;
  return(slope1) ;
}
```

```
float second_derivative(time, trace)
int  time ;
float trace[] ;
{
  float sec_der ;

  sec_der = (slope(time, trace) - slope((time - window_sec_der), trace))
            / ((float) window_sec_der) ;
  return(sec_der) ;
}


float  Cost(bp_data)
struct  bp_data_struct *bp_data ;
{
   float cost ;
   int  nunits = (bp_data -> no_units[0]) ;  /* Number of input units */
   int  nweights = 0 ;
   float temp ;
   int  i ;

     /* Find the number of units and weights in the configuration */
   for (i=1; i <= (bp_data -> nlayers) ; i++) {
     nunits += (bp_data -> no_units[i]) ;
     nweights += (bp_data -> no_units[i]) * (bp_data -> no_units[i-1]) ;
   }
   temp = (float) nweights ;
       /* ***** HEURISTIC COST FUNCTION ***** */
   cost = nunits + 0.1 * sqrt(temp) ;
   return(cost) ;
}
```


cost.c

```
/* **********  cost.c  ********** */

/* This program returns cost of a configuration. The configuration is supplied
   as an input file to the program. */

#include <stdio.h>
#include <math.h>
#include "constants.h"
#include "structs.h"

extern float  Cost() ;
void err() ;

main(argc, argv)
int  argc;
char  *argv[];
{
  struct bp_data_struct bp_data ;
  int  i ;
  FILE  *fpip, *fopen() ;
  if (argc < 2)
    err("Usage: cost bp_data_file") ;
  fpip = fopen(*++argv, "r") ;
    /* Just read number of layers and the number of units in each layer */
  fscanf(fpip, "%d \n", &(bp_data.nlayers)) ;
  for (i=0; i <= (bp_data.nlayers) ; i++) {
    fscanf(fpip, "%d", &(bp_data.no_units[i]) ) ;
  }
  fclose(fpip) ;
  printf("%f \n", Cost(&bp_data)) ;
}


void err(mesg)
char *mesg;
{
   fprintf(stderr, "%s \n", mesg);
   exit(0);
}
```

constants.h

```
/* **********  constants.h  ********** */

/* Header file containing constants for the type1-order-type2 system */


/* Network Specification constants. */

#define  max_layers                3
#define  max_weights               50
#define  max_units                 15
#define  max_patterns              10
#define  max_pat_length            15

/* Overall system constants. */

#define  quantum                   10
#define  max_quanta                50
#define  max_epochs          max_quanta * quantum

/* window_tsse, window_slope and window_sec_der are used for moving window
   averaging operations. E.g., Window_tsse -> Average tsse value over past
   window_tsse values. */

#define  window_tsse               5
#define  window_slope              2
#define  window_sec_der            2
#define  network_window_size       3

/* Number of networks in the pool of networks. */

#define  NO_NETS_IN_PON            4

/* Constant for ordering the networks. Save first ORDER_MAX_NETS networks in
   type1 experiments and order them by order.c */

#define  ORDER_MAX_NETS            5

/* Networks which do not converge in time <= norm_time, replace their objective
   function value by MIN_OBJECTIVE. */

#define  MIN_OBJECTIVE             0.0
```



```
/* worst_cost is some upper bound on the cost of networks considered. Any loose
   bound will do. */

#define  worst_cost                100.0

/* Various debug bits for generating intermediate information about each of
   the experiments. bit = 0 -> don't save information, = 1 -> debug */

#define  debug                     1
#define  function_debug            1
#define  order_debug               1

/* Constants for the function generator. (function.lsp) */

#define  initial_amp_att           0.9
#define  reduction_factor          1.5
#define  add_term_after_no_reductions    8

/* Constants for the performance measuring element (pme.c) */

#define  X                         10.0
#define  Y                         1.0
#define  Z                         1.0

/* Constants for the overall system. */

#define  no_cycles                 2
#define  type1_duration_in_quanta        10

/* Number of functions to try in type2 experiments. */

#define  no_of_functions_to_try    16
#define  initial_no_of_terms_in_goodness    1
```

back-prop.c

```
/* ********** back-prop.c ********** */

/* The general back propagation algorithm. The algorithm is activated by the
   procedure bp(). Various parameters of the procedure are explained below:
   (Note: Before using bp(), please initialize the seed.)

   bp_data: network specification structure.
   npatterns: number of training patterns.
   patterns: the training patterns.
   trace: an array for saving the tsse values for the last quantum of training.
   iwts_data: returns the initial weight configuration in this, if the procedure
              is being called on a network for which initial weights are not
              specified. (i.e. iwts_data returns the initial weights if newstart
              bit = 1 in bp_data structure).

Various other supporting procedures provided in this file are:

copy_bp_data(): Copy network specification structure from one to another.
                Parameters:
                bp_data_from: structure containing specification.
                bp_data_to: copy specification into this structure.
random_no(): Generate a random number between -wrange and +wrange. wrange is
             taken as an input parameter.
read_bp_data(): Procedure for reading in network specification from a file.
                Parameters:
                bp_data: return the network specification into this structure.
                fpip: read the input data from this file. Pointer to the file
                   is provided.
write_bp_data(): Write network specification into a file. Parameters:
                bp_data: structure containing network specification.
                newstart: newstart = 1 -> don't save weights. else do if
                          weights are included in specification.
                fpop: save the specification into this file. Pointer to the
                   file is provided.

Comments: (General) About the bp() procedure:
          Weight from j to i = weight[N] : layer containing j has l units
          N = N' + i * l + j ; N' = total weights in previous layers
*/


#include <stdio.h>
```



```
#include <math.h>
#include <sys/types.h>
#include <sys/timeb.h>
#include "constants.h"
#include "structs.h"

#define max_rand 2147483648.0

void  bp();
void  copy_bp_data();
float random_no();
void  read_bp_data();
float sigmoidal();
void  write_bp_data();

extern void write_bp_data();

long seed;



/* THE BACK-PROPAGATION ALGORITHM */

void bp(bp_data, npatterns, pattern, trace, iwts_data)
struct bp_data_struct *bp_data, *iwts_data;
int   npatterns;
float pattern[max_patterns][max_pat_length];
float trace[quantum];
{
  int   nlayers = (bp_data -> nlayers);
  int   nquanta = (bp_data -> nquanta);
  float tsse;
  float ecrit = (bp_data -> ecrit);
  float lrate = (bp_data -> lrate);
  float momentum = (bp_data -> momentum);
  float wrange = (bp_data -> wrange);
  int   newstart = (bp_data -> newstart);
  float *activation;
  float *weights;
  int   *no_units;
    /* Copy pointers to minimize typing in the rest of the program */
  int   pattern_no, unit_no, i, j, k, l;
  int   temp_wt_no, layer_no, weight_no;
  int   j_unit_no;      /* index of weight from j to unit_no */
  int   j_k;            /* index of weight from j to k */
  float weighted_sum, bias, Delta, temp;
  int   iteration;
  int   current_quantum = 0;
```

```
int   ninputs, noutputs ;
int   nunits ;     /* number of units in the network */
int   nweights ;   /* number of weights in the network */
int   dimension ;  /* number of weights and biases in the network */
float delta[max_units] ;
float prev_weights[max_weights] ;
int   units_so_far[max_layers] ;
      /* units_so_far[i] = total no of units in all
                 the previous layers including ith */
int   weights_so_far[max_layers] ;
          /* weights_so_far[i] = total no of wts in all
                 the previous layers including ith */

activation = (bp_data -> activation) ;
weights = (bp_data -> weights) ;
no_units = (bp_data -> no_units) ;
ninputs = no_units[0] ;
noutputs = no_units[nlayers] ;
units_so_far[0] = ninputs ;
weights_so_far[0] = 0 ;
for (i = 1 ; i <= nlayers ; i++) {
 units_so_far[i] = units_so_far[i-1] + no_units[i] ;
 weights_so_far[i] = weights_so_far[i-1]
           + no_units[i-1] * no_units[i] ;
}
nunits = units_so_far[nlayers] ;
nweights = weights_so_far[nlayers] ;
dimension = weights_so_far[nlayers] + units_so_far[nlayers] - ninputs ;
if (newstart == 1) {
 for (i = 0 ; i < dimension ; i++) {
  weights[i] = random_no(wrange) ;
 }
}
for (i = 0 ; i < dimension ; i++) {
 prev_weights[i] = weights[i] ;
}
if (newstart == 1) {
          /* Save the initial configuration for type2 expts */
 (bp_data -> newstart) = 0 ;
 copy_bp_data(bp_data, iwts_data) ;
}
tsse = 2.0 * ecrit ; /* Go through the while loop at least once */
while ((current_quantum < nquanta) && (tsse > ecrit)) {
 for (iteration = 0 ; iteration < quantum ; iteration++) {
  tsse = 0.0 ;
  for (pattern_no = 0 ; pattern_no < npatterns ; pattern_no++) {
          /* apply inputs */
   for (unit_no=0 ; unit_no < ninputs ; unit_no++) {
    activation[unit_no] = (pattern[pattern_no][unit_no]) ;
```



```
   }
   for (unit_no=ninputs ; unit_no < nunits ; unit_no++) {
                                   /* calculate activations */
    layer_no = 0 ;  /* find the layer no, the unit belongs to */
    while ((unit_no + 1) > units_so_far[layer_no]) {
        ++layer_no ;
    }       /* layer_no is the layer, the unit is in */
    weighted_sum = 0.0 ;
    for (j = (units_so_far[layer_no - 1] -
                                   no_units[layer_no - 1]) ;
         j < units_so_far[layer_no - 1] ; j++) {
        j_unit_no = weights_so_far[layer_no -1] +
                 (unit_no - units_so_far[layer_no - 1]) *
                    no_units[layer_no -1] +
                    (j - units_so_far[layer_no -1] +
                     no_units[layer_no -1]);
     weighted_sum += activation[j] * weights[j_unit_no] ;
    }
     bias = weights[nweights + unit_no - ninputs] ;
     activation[unit_no] = sigmoidal(weighted_sum, bias) ;
   }
   for (j = (nunits - noutputs) ; j < nunits ; j++) {
                                   /* Calculate error */
    temp = activation[j] - (pattern[pattern_no][
                          j - nunits + noutputs + ninputs]) ;
/* *** CORRECTION FOR TWO-SPIRAL PROBLEM *** */
    /*
    if ((temp <= 0.4) && (temp >= - 0.4)) temp = 0.0 ;
    */
    tsse += temp * temp ;
   }
    /* Calculate deltas */
for (j= (nunits - noutputs) ; j < nunits ; j++) {
                          /* deltas for the output layer */
 delta[j] = activation[j] * (1.0 - activation[j]) *
          ((pattern[pattern_no][
                          j - nunits + noutputs + ninputs]) -
           activation[j]) ;
}
for (j = (units_so_far[nlayers - 1] - 1) ; j >= ninputs ; j--) {
                    /* Calculate deltas for other layers */
 layer_no = 0 ;  /* Find the layer j belongs to */
 while ((j + 1) > units_so_far[layer_no]) {
      ++layer_no ;
 }    /* layer_no is the layer, j is in */
 Delta = 0.0 ;
 for (k=units_so_far[layer_no] ;
                          k < units_so_far[layer_no + 1] ; k++) {
     j_k = weights_so_far[layer_no]
```

```
                + (k - units_so_far[layer_no])
                        * no_units[layer_no]
                        + (j - units_so_far[layer_no - 1]) ;
    Delta += delta[k] * weights[j_k] ;
  }
  delta[j] = Delta * activation[j] * (1.0 - activation[j]) ;
}
        /* Update weights */
for (weight_no = 0 ; weight_no < nweights ; weight_no++) {
      /* Find the unit indices the weight is between and
                then use the formula. */
      /* First find the layer_no, where weight is between
                layer_no and (layer_no + 1) */
  layer_no = 0 ;
  while ((weight_no + 1) > weights_so_far[layer_no]) {
  ++layer_no ;
  }
       --layer_no ;
  temp_wt_no = weight_no - weights_so_far[layer_no] ;
                                                  /* = i * l + j */
  l = no_units[layer_no] ;
  i = temp_wt_no / l ;     /* weight is from j to i */
  j = temp_wt_no % l ;
                        /* i and j are local indices wrt their layers */
       i = i + units_so_far[layer_no] ;
       j = j + units_so_far[layer_no] - no_units[layer_no] ;
                /* i and j are now global indices wrt the network */
       temp = weights[weight_no] ;
  weights[weight_no] = weights[weight_no] + lrate * delta[i]
                                * activation[j] + momentum
                                * (weights[weight_no]
                                - prev_weights[weight_no]) ;
          /* Include weight decay term here ! */
       prev_weights[weight_no] = temp ;
  }
  for (weight_no = nweights ;
                        weight_no < (nweights + nunits - ninputs) ;
                                                weight_no++) {
                                                        /* set biases */
       temp = weights[weight_no] ;
  weights[weight_no] = weights[weight_no]
                                + lrate * delta[weight_no - nweights
                                + ninputs] + momentum
                                * (weights[weight_no]
                                - prev_weights[weight_no]) ;
       prev_weights[weight_no] = temp ;
  }
}
trace[iteration] = tsse ;
```



```
        /* Save training trace for the last time quantum. */
    }
    ++current_quantum ;
  }
  (bp_data -> spent_nquanta) += nquanta ;
  (bp_data -> tsse) = tsse ;
}


float sigmoidal(weighted_sum, bias)
float weighted_sum, bias;
{
  float temp ;
  temp = exp(- weighted_sum - bias) ;
  temp += 1.0 ;
  temp = 1.0 / temp ;
  return(temp) ;
}


float random_no(wrange)
float wrange;
{
  float temp, number ;
  temp = (float) mrand48(seed) ;
  number = temp / max_rand ; /* number is rand bet [-1, 1] */
  number = number * wrange ;
  return(number) ;
}



void copy_bp_data(bp_data_from, bp_data_to)
struct bp_data_struct  *bp_data_from, *bp_data_to ;
{
  int  i, j ;
  int  nunits = 0 ;  /* number of units */
  int  nwts_bs = 0 ;

  (bp_data_to -> nlayers) = (bp_data_from -> nlayers) ;
  for (i=0; i <= (bp_data_from -> nlayers) ; i++) {
   (bp_data_to -> no_units[i]) = (bp_data_from -> no_units[i]) ;
   nunits += (bp_data_to -> no_units[i]) ;
  }
  (bp_data_to -> ecrit) = (bp_data_from -> ecrit) ;
  (bp_data_to -> nquanta) = (bp_data_from -> nquanta) ;
  (bp_data_to -> lrate) = (bp_data_from -> lrate) ;
```

```
  (bp_data_to -> momentum) = (bp_data_from -> momentum) ;
  (bp_data_to -> wrange) = (bp_data_from -> wrange) ;
  (bp_data_to -> spent_nquanta) = (bp_data_from -> spent_nquanta) ;
  (bp_data_to -> ubf) = (bp_data_from -> ubf) ;
  (bp_data_to -> lib_net_no) = (bp_data_from -> lib_net_no) ;
  (bp_data_to -> newstart) = (bp_data_from -> newstart) ;
  (bp_data_to -> tsse) = (bp_data_from -> tsse) ;
  (bp_data_to -> goodness) = (bp_data_from -> goodness) ;
  (bp_data_to -> guidance) = (bp_data_from -> guidance) ;
  for (j=0 ; j < nunits ; j++) {   /* copy activation array */
    (bp_data_to -> activation[j]) = (bp_data_from -> activation[j]) ;
  }
          /* Calculate no of weights and biases */
  for (i=0 ; i < (bp_data_from -> nlayers) ; i++) {
    nwts_bs += ((bp_data_from -> no_units[i]) + 1)
                    * (bp_data_from -> no_units[i+1]) ;
  }
  for (i=0 ; i < nwts_bs ; i++) {
    (bp_data_to -> weights[i]) = (bp_data_from -> weights[i]) ;
  }
}



void  read_bp_data(bp_data, fpip)
struct bp_data_struct   *bp_data ;
FILE   *fpip ;
{
  int  i, j ;
  int  nwts_bs ;
          /* total number of weights in the net including biases */
  int  nunits = 0 ;  /* number of units */

  fscanf((fpip, "%d \n", &(bp_data -> nlayers)) ;
  for (i=0; i <= (bp_data -> nlayers) ; i++) {
   fscanf((fpip, "%d", &(bp_data -> no_units[i]) ) ;
   nunits += (bp_data -> no_units[i]) ;
  }
  fscanf((fpip, "\n%f \n", &(bp_data -> ecrit) ) ;
  fscanf((fpip, "%d \n", &(bp_data -> nquanta) ) ;
  fscanf((fpip, "%f \n", &(bp_data -> lrate) ) ;
  fscanf((fpip, "%f \n", &(bp_data -> momentum) ) ;
  fscanf((fpip, "%f \n", &(bp_data -> wrange) ) ;
  fscanf((fpip, "%d \n", &(bp_data -> newstart) ) ;
  if ((bp_data -> newstart) == 0 ) {
   fscanf((fpip, "%d \n", &(bp_data -> spent_nquanta) ) ;
   fscanf((fpip, "%d \n", &nwts_bs) ;
   for (i = 0 ; i < nwts_bs ; i++)
          fscanf((fpip, "%f \n", &(bp_data -> weights[i]) ) ;
```



```
   fscanf((fpip, "%d \n", &nunits) ;
   for (j=0 ; j < nunits ; j++)
          fscanf((fpip, "%f \n", &(bp_data -> activation[j])) ;
   fscanf((fpip, "%f \n", &(bp_data -> tsse)) ;
   fscanf((fpip, "%f \n", &(bp_data -> goodness)) ;
   fscanf((fpip, "%f \n", &(bp_data -> guidance)) ;
   fscanf((fpip, "%d \n", &(bp_data -> ubf)) ;
  }
  else {
   (bp_data -> spent_nquanta) = 0 ;
   for (j=0 ; j < nunits ; j++)   /* initialize activation array */
          (bp_data -> activation[j]) = 0.0 ;
  }
}



void  write_bp_data(bp_data, newstart, fpop)
struct bp_data_struct   *bp_data ;
int   newstart ;  /* newstart = 0 means weights should be saved */
FILE  *fpop ;
{
  int  i, j ;
  int  nunits ;
  int  nwts_bs ;
          /* total number of weights in the net including biases */

  fprintf((fpop, "%d \n", (bp_data -> nlayers)) ;
  for (i=0; i <= (bp_data -> nlayers) ; i++) {
   fprintf((fpop, "%d  ", (bp_data -> no_units[i]) ) ;
  }
  fprintf((fpop, "\n%f \n", (bp_data -> ecrit) ) ;
  fprintf((fpop, "%d \n", (bp_data -> nquanta) ) ;
  fprintf((fpop, "%f \n", (bp_data -> lrate) ) ;
  fprintf((fpop, "%f \n", (bp_data -> momentum) ) ;
  fprintf((fpop, "%f \n", (bp_data -> wrange) ) ;
  fprintf((fpop, "%d \n", newstart) ;
  if ((newstart == 0 ) && (bp_data -> newstart == 0)) {
   fprintf((fpop, "%d \n", (bp_data -> spent_nquanta) ) ;
   nwts_bs = 0 ;        /* calculate no of weights + biases */
   for (i = 0 ; i < (bp_data -> nlayers) ; i++) {
          nwts_bs += ((bp_data -> no_units[i]) + 1)
                    * (bp_data -> no_units[i+1]) ;
   }
   fprintf((fpop, "%d \n", nwts_bs) ;
   for (i = 0 ; i < nwts_bs ; i++) {
          fprintf((fpop, "%f \n", (bp_data -> weights[i]) ) ;
   }
   nunits = 0 ;     /* calculate no. of units */
```

```
  for (i=0 ; i <=(bp_data -> nlayers) ; i++)
    nunits += (bp_data -> no_units[i]) ;
  fprintf(fpop, "%d \n", nunits) ;
  for (i=0 ; i < nunits ; i++)
    fprintf(fpop, "%f \n", (bp_data -> activation[i])) ;
  fprintf(fpop, "%f \n", (bp_data -> tsse)) ;
  fprintf(fpop, "%f \n", (bp_data -> goodness)) ;
  fprintf(fpop, "%f \n", (bp_data -> guidance)) ;
  fprintf(fpop, "%d \n", (bp_data -> ubf)) ;
 }
}
```


pme.c

```
/* ********** pme.c ********** */

/* The function pme() implements the performance measuring element part
  of Type II experiments. The input parameters of this function are:

  iwts: structure containing training traces of various configurations
        generated by the order program.
  norm: structure containing various normalization constants.

  The program returns Inconsistency Sum for all the configurations
  considered.
  Whenever debug bit is set to 1, an additional file "perfm" is created which
  contains some more information about the performance of the function. The
  information contained in this file is:
  IS
  Sum of monotonic inconsistencies
  Sum of inconsistency counts
  Sum of inconsistency differences
*/


#include <stdio.h>
#include <math.h>
#include "./type1/constants.h"
#include "./type1/structs.h"


float monotonic_inconsistency() ;
void  pair_wise_comparison_converged() ;
void  pair_wise_comparison_not_converged() ;

extern float geg() ;

float pme(iwts, norm)
struct configs_tried *iwts ;
struct norm_struct   *norm ;
{
  struct  order_obj_struct *root_obj1, *root_obj2 ;
  int     total_count = 0 ;
          /* Total no of places where pair_wise_comparison was made. */
  float   mod_count = 0.0 ;  /* no_of_inconsistent_places * (obj1 - obj2) */
  float   mod_inconsis = 0.0 ;
```

```
float  total_monotonic_inconsis = 0.0 ;
float  final_inconsis ;
int    quantum_no ;
int    net_no ;
float  ntsse, nslope, nsec_der ;  /* normalized values */
FILE   *fpop, *fopen() ;
int    norm_time = norm -> norm_time ;
float  norm_tsse = norm -> norm_tsse ;

   /* *** CALCULATE ALL GOODNESS FUNCTION VALUES *** */
for (net_no=0 ; net_no < (iwts -> no_nets) ; net_no++) {
  for (quantum_no=1 ; quantum_no <= (iwts -> nets[net_no]).spent_nquanta ;
                          quantum_no++) {
            /* CALCULATE NORMALIZED PARAMETERS */
        ntsse = iwts -> tsse[net_no][quantum_no] ;
        ntsse = ntsse / norm_tsse ;
        nslope = iwts -> slope[net_no][quantum_no] ;
        nslope = nslope * norm_time / norm_tsse ;
        nsec_der = iwts -> sec_der[net_no][quantum_no] ;
        nsec_der= nsec_der * norm_time * norm_time / (norm_tsse * norm_tsse) ;
    (iwts -> goodness[net_no][quantum_no])= geg(ntsse, nslope, nsec_der) ;
  }
}
      /* *** MONOTONIC INCONSISTENCY *** */
root_obj1 = (iwts -> converged) ;
total_monotonic_inconsis = monotonic_inconsistency(iwts, root_obj1) ;
root_obj1 = (iwts -> not_converged) ;
total_monotonic_inconsis += monotonic_inconsistency(iwts, root_obj1) ;
          /* *** PAIR WISE COMPARISON *** */
      /* ***** DO THE ANALYSIS FOR CIRCUITS THAT CONVERGED ***** */
root_obj1 = (iwts -> converged) ;
if (root_obj1 != NULL) {
  root_obj2 = root_obj1 -> next ;
  pair_wise_comparison_converged(root_obj1, root_obj2, iwts, &total_count,
                                    &mod_count, &mod_inconsis) ;
}
    /* *** CIRCUITS THAT DIDN'T CONVERGE *** */
root_obj1 = (iwts -> converged) ;
root_obj2 = (iwts -> not_converged) ;
pair_wise_comparison_not_converged(root_obj1, root_obj2, iwts,
                                    &total_count, &mod_count, &mod_inconsis) ;
final_inconsis = X * total_monotonic_inconsis + Y * mod_count
                 + Z * mod_inconsis ;
if (debug == 1) {
  fpop = fopen("perfm", "w") ;
  fprintf(fpop, "IS = %10.1f\n", final_inconsis) ;
  fprintf(fpop, "Sum of monotonic inconsistencies = %f\n",
                    total_monotonic_inconsis) ;
  fprintf(fpop, "Sum of inconsistency counts = %f\n", mod_count) ;
```



```
  fprintf(fpop, "Sum of inconsistency differences =% f\n", mod_inconsis) ;
  fclose(fpop) ;
 }
 return(final_inconsis) ;
}


float monotonic_inconsistency(iwts, root)
struct configs_tried   *iwts ;
struct order_obj_struct *root ;
{
  float goodness1, prev_goodness ;
  int  net_no, quantum_no ;
  float monotonic_inconsis ;
  float total_monotonic_inconsis = 0.0 ;
  while (root != NULL) {
    net_no = root -> net_no ;
    prev_goodness = (iwts -> goodness[net_no][1]) ;
    monotonic_inconsis = 0.0 ;
    for (quantum_no =1; quantum_no <= (iwts -> nets[net_no]).spent_nquanta ;
                        quantum_no++) {
      goodness1 = (iwts -> goodness[net_no][quantum_no]) ;
          if (goodness1 < prev_goodness)
            monotonic_inconsis += (prev_goodness - goodness1) ;
          prev_goodness = goodness1 ;
    }
    total_monotonic_inconsis += monotonic_inconsis * (root -> objective) ;
    root = root -> next ;
  }
  return(total_monotonic_inconsis) ;
}


void pair_wise_comparison_converged(root_obj1, root_obj2, iwts, total_count,
                                    mod_count, mod_inconsis)
struct order_obj_struct  *root_obj1, *root_obj2 ;
struct configs_tried     *iwts ;
int   *total_count ;
float *mod_count, *mod_inconsis ;
{
  struct order_obj_struct *root2 ;
  int   net_no1, net_no2, quantum_no1, quantum_no2, traintimediff ;
  int   count ;
  float inconsis, goodness1, goodness2 ;

  while ((root_obj1 != NULL) && (root_obj2 != NULL)) {
    net_no1 = root_obj1 -> net_no ;
    root2 = root_obj2 ;
    while (root2 != NULL) {
```

```
    net_no2 = root2 -> net_no ;
        count = 0 ;
        inconsis = 0.0 ;
        for (quantum_no1=1 ; quantum_no1 <= (iwts -> nets[net_no1]).spent_nquanta ;
                              quantum_no1++) {
          traintimediff = (iwts -> nets[net_no2]).spent_nquanta -
                              (iwts -> nets[net_no1]).spent_nquanta ;
          for (quantum_no2=1 ; quantum_no2 <= (quantum_no1 + traintimediff) ;
                              quantum_no2++) {
        goodness1 = (iwts -> goodness[net_no1][quantum_no1]) ;
        goodness2 = (iwts -> goodness[net_no2][quantum_no2]) ;
                ++(*total_count) ;
                if (goodness1 < goodness2) {
                  inconsis += (goodness2 - goodness1) ;
                  ++count ;
                }
      }
        }
        *mod_count += count * ((iwts -> nets[net_no2]).spent_nquanta -
                              (iwts -> nets[net_no1]).spent_nquanta) ;
        *mod_inconsis += inconsis * ((iwts -> nets[net_no2]).spent_nquanta -
                              (iwts -> nets[net_no1]).spent_nquanta) ;
        root2 = root2 -> next ;
    }
    root_obj1 = root_obj1 -> next ;
    root_obj2 = root_obj2 -> next ;
  }
}

void pair_wise_comparison_not_converged(root_obj1, root_obj2, iwts,
                                        total_count, mod_count, mod_inconsis)
struct order_obj_struct  *root_obj1, *root_obj2 ;
struct configs_tried     *iwts ;
int   *total_count ;
float  *mod_count, *mod_inconsis ;
{
  struct order_obj_struct *root2 ;
  int   net_no1, net_no2, quantum_no1, quantum_no2, traintimediff ;
  int   count ;
  float inconsis, goodness1, goodness2 ;

  while ((root_obj1 != NULL) && (root_obj2 != NULL)) {
    net_no1 = root_obj1 -> net_no ;
    root2 = root_obj2 ;
    while (root2 != NULL) {
        net_no2 = root2 -> net_no ;
        count = 0 ;
        inconsis = 0.0 ;
        for (quantum_no1=1 ; quantum_no1 <= (iwts -> nets[net_no1]).spent_nquanta ;
```



```
                              quantum_no1++) {
          traintimediff = (iwts -> nets[net_no2]).spent_nquanta -
                              (iwts -> nets[net_no1]).spent_nquanta ;
          for (quantum_no2=1 ; quantum_no2 <= (quantum_no1 + traintimediff) ;
                              quantum_no2++) {
        goodness1 = (iwts -> goodness[net_no1][quantum_no1]) ;
        goodness2 = (iwts -> goodness[net_no2][quantum_no2]) ;
                ++(*total_count) ;
                if (goodness1 < goodness2) {
                  inconsis += (goodness2 - goodness1) ;
                  ++count ;
                }
      }
        }
        *mod_count += count * ((iwts -> nets[net_no2]).spent_nquanta -
                              (iwts -> nets[net_no1]).spent_nquanta) ;
        *mod_inconsis += inconsis * ((iwts -> nets[net_no2]).spent_nquanta -
                              (iwts -> nets[net_no1]).spent_nquanta) ;
        root2 = root2 -> next ;
    }
    root_obj1 = root_obj1 -> next ;
  }
}
```

order.c

```
/* **********  order.c  ********** */

/* The function order is used to train a set of configurations to completion
   and save their execution traces. Complete training => train each
   configuration until its tsse < ecrit OR time < norm_time. If after
   norm_time, tsse does not fall below ecrit, then the configuration is said
   not to have converged. Parameters of the order function are explained below:

   pat: structure containing input output patterns.
   norm: structure containing normalization constants.
   iwts: structure containing initial weight configurations. Training traces
         of configurations are attached to this structure. (Output of the
         program.)

   Following additional information about order is saved, if various debug
   bits are set in constants.h file:
   debug = 1:
   The objective function values and the training times for all the networks
   which converged along with their network numbers are saved in "conv" file.
   This file is sorted based upon the training times. The same information
   about networks which didn't converge is saved in "no_conv" file. The
   format of these files is as follows:
      Net_no    Objective_function_value   Training_time

   order_debug = 1:
    Individual traces about each and every network considered are saved in
    "trace(network_no)" files. The format of these files is:
         average_tsse  average_slope  average_second_derivative   time
*/


#include <stdio.h>
#include <math.h>
#include <sys/types.h>
#include <sys/timeb.h>
#include "../type1/constants.h"
#include "../type1/structs.h"


void  order() ;
void  BP1();
float  goodness_values() ;
```



```
struct order_obj_struct *Insert_T() ;
struct order_obj_struct *Insert() ;
void  PRINT() ;


extern float  Cost() ;    /* returns cost of the neural network */
extern float  av_tsse() ;
extern void  bp() ;
extern float  objective() ;
extern void  read_bp_data() ;
extern float  second_derivative() ;
extern float  slope() ;
extern void  write_bp_data() ;

extern long  seed ;


void order(pat, norm, iwts)
struct  pat_struct    *pat ;
struct  norm_struct   *norm ;
struct  configs_tried  *iwts ;
{
  struct order_obj_struct *node ;
  float  obj ;   /* objective function value for the net */
  float  whole_trace[max_epochs] ;
  FILE   *fpop, *fopen();
  int    trainingtime ;
  int    network_no ;    /* Current network number */
  char   name[20] ;

  seed = time(0) ;
  srand48(seed);
  (iwts -> converged) = NULL ;
  (iwts -> not_converged) = NULL ;
  for (network_no=0 ; network_no < (iwts -> no_nets) ; network_no++) {
    BP1(&(iwts -> nets[network_no]), (norm -> norm_time),
          (pat -> npatterns), (pat -> patterns), whole_trace) ;
    if (order_debug == 1) {
      sprintf(name, "trace%d", network_no) ;
          fpop = fopen(name, "w") ;
    }
    else
          fpop = stderr ;
    obj = goodness_values(iwts, network_no, norm,
                                    &trainingtime, whole_trace, fpop) ;
    if (order_debug == 1)
           fclose(fpop) ;
    node = (struct order_obj_struct *) malloc(sizeof(*node)) ;
    node -> net_no = network_no ;
```

```
    node -> objective = obj ;
    node -> traintime = trainingtime ;
    if ((iwts -> nets[network_no]).tsse <= (iwts -> nets[network_no]).ecrit)
          (iwts -> converged) = Insert_T((iwts -> converged), node) ;
    else {
          (node -> objective) = MIN_OBJECTIVE ;
          (iwts -> not_converged) = Insert((iwts -> not_converged), node) ;
    }
  } /* all the networks have been trained to completion */
  if (debug == 1) {
    fpop = fopen("conv", "w") ;
    PRINT((iwts -> converged), fpop) ;
    fclose(fpop) ;
    fpop = fopen("no_conv", "w") ;
    PRINT((iwts -> not_converged), fpop) ;
    fclose(fpop) ;
  }
}


void BP1(bp_data, norm_time, npatterns, patterns, whole_trace)
struct bp_data_struct *bp_data ;
int    norm_time ;
int    npatterns ;
float  patterns[max_patterns][max_pat_length] ;
float  whole_trace[] ;
{
  int   i ;
  float trace[quantum] ;
  struct bp_data_struct *junk_bp_data ;

  (bp_data -> tsse) = 2.0 * (bp_data -> ecrit) ;
            /* To go through the while loop at least once. */
  while (((bp_data -> tsse) > (bp_data -> ecrit)) &&
          ((bp_data -> spent_nquanta) < norm_time)) {
    bp(bp_data, npatterns, patterns, trace, junk_bp_data) ;
    for (i=0 ; i < quantum ; i++) {
          whole_trace[((bp_data -> spent_nquanta) - 1) * quantum + i] = trace[i] ;
    }
  }
}



float goodness_values(iwts, network_no, norm, trainingtime, whole_trace, fpop)
struct  configs_tried *iwts ;
struct  norm_struct  *norm ;
int     network_no ;
```



```
int     *trainingtime ;
float   whole_trace[] ;
FILE    *fpop ;
{
  int  i ;
  float cost1 ;
  int  time ;  /* current time */
  float obj ;  /* objective fn value for the net */
  float av_tsse1, slope1, sec_der ;
  float ecrit ;
  int  spent_nquanta = (iwts -> nets[network_no]).spent_nquanta ;

  cost1 = Cost(&(iwts -> nets[network_no])) ;
  (iwts -> cost[network_no]) = cost1 ;
  for (i=1 ; i <= spent_nquanta ; i++) {  /* do it for all the quanta */
    time = i * quantum - 1; /* since trace starts from "0" */
    av_tsse1 = av_tsse(time, whole_trace) ;
    (iwts -> tsse[network_no][i]) = av_tsse1 ;
    slope1 = slope(time, whole_trace) ;
    (iwts -> slope[network_no][i]) = slope1 ;
    sec_der = second_derivative(time, whole_trace) ;
    (iwts -> sec_der[network_no][i]) = sec_der ;
    (iwts -> int_time[network_no][i]) = i ;
    if (order_debug == 1)
          fprintf(fpop, "%f %f %f %d\n", av_tsse1, slope1, sec_der, i) ;
  }
  ecrit = (iwts -> nets[network_no]).ecrit ;
  obj = objective(spent_nquanta, cost1) ;
  *trainingtime = spent_nquanta ;
  return(obj) ;
}



struct order_obj_struct *Insert_T(root, node)
struct order_obj_struct *root, *node ;
{
  struct order_obj_struct *ptr, *prev_ptr ;
  if ((root == NULL) || ((node -> traintime) < (root -> traintime))) {
    node -> next = root ;
                        /* Insert node in the beginning of the list */
    root = node ;
  }
  else {
    ptr = root ;
    while ((ptr -> traintime) <= (node -> traintime)) {
          prev_ptr = ptr ;
          ptr = (ptr -> next) ;
```

```
    if (ptr == NULL) break ;
    }
    (node -> next) = (prev_ptr -> next) ;
    (prev_ptr -> next) = node ;
  }
  return(root) ;
}



struct order_obj_struct *Insert(root, node)
/* Insert the node in the beginning */
struct order_obj_struct *root, *node ;
{
  node -> next = root ;
  return(node) ;
}



void PRINT(root, fpop)
struct  order_obj_struct *root ;
FILE    *fpop ;
{
  struct order_obj_struct *ptr ;
  fprintf(fpop, "Net_No  Objective  Trainingtime \n") ;
  ptr = root ;
  while (ptr != NULL) {
    fprintf(fpop, "  %d     %f        %d \n", (ptr -> net_no),
                  (ptr -> objective), (ptr -> traintime)) ;
    ptr = (ptr -> next) ;
  }
}
```


objective.h

```
/* **********  objective.h  ********** */

/* This header file contains the coefficients of the objective function.
   See objective.c file for the exact objective function. */

#define  A   10.0
#define  B   (8.0 / 50.0)
#define  C   (2.0 / 12.0)
```

objective.c

```
/* **********  objective.c  ********** */

/* This program returns the objective function value of a configuration,
   given its training time in number of quanta and the cost. */

#include <stdio.h>
#include "objective.h"

float objective(TT, cost)
int    TT  ;   /* Training Time in quanta   */
float  cost ;  /* Cost of the network       */
{
  float obj ;
        /* ***** OBJECTIVE FUNCTION USED ***** */
     /* Various constants are specified in objective.h file */
  obj = A - B * TT - C * cost ;
  return(obj) ;
}
```


TYPE II makefile

```
BIN = objective order pme

all:${BIN}

FLAGS = -O2

objective: objective.c objective.h
        cc ${FLAGS} -c objective.c

order: order.c ../type1/constants.h ../type1/structs.h
        cc ${FLAGS} -c order.c -lm

pme: pme.c ../type1/constants.h ../type1/structs.h
        cc ${FLAGS} -c pme.c -lm

clean:
        rm -f *.o core ${BIN}
```

function.lsp

```
;;; ********** function.lsp **********

;;; The program in lisp implements the function generator part of type2
;;; experiments. The program outputs the expected goodness function generated
;;; in geg.c file. The program gets the performance information about a
;;; function generated by executing the pme() function.
;;; This program requires a library file which contains all possible terms
;;; which are candidates for the goodness function.
;;;
;;; Format of the *library.lsp file:
;;;     no_lib_terms
;;;     Repeat for each term : sign, coeff, function.

;;; Include the following file to get all the constants declared for the system.

#include "../../type1/constants.h"

;;; Some information about the program:
;;; In this program an array term_array is declared. This array contains the
;;; goodness function. This function is initialized to
;;; "initial_no_of_terms_in_goodness" number of terms of the library plus
;;; the fixed goodness (see below). no_lib_terms denote the number of terms
;;; in the library. The array index starts from zero. no_terms denotes only
;;; these many terms of the array are under consideration now. The array
;;; old_good_array with old_no_terms no of terms has the current best goodness
;;; function, performance kept in old_good_perfm.

;;; amp_att is the amplification or attenuation coefficient.
;;; amplification coeff = 1 + amp_att
;;; attenuation coeff = 1 / (1 + amp_att).
;;; Reduction factor is the factor by which amp_att is reduced, after a local
;;; minimum for the current function is reached.
;;; Add a new term to the current function after amp_att has been reduced
;;; add_term_after_no_reductions number of times.

(setf fixed_goodness '(10.0 * (1.0 - TSSE)))


(defstruct term
  number          ; term number
  prev_op         ; previous operation attenuate/amplify/sign.
  sensitivity     ; to amplify/attenuate pattern operations.
```



```
  sign            ; sign of the coefficient.
                  ; ++ means sign is + and fixed.
                  ; -- means sign is - and fixed.
  coeff           ; value of the coefficient.
  function        ; term = sign coeff * function.
)


(setf initial_sensitivity 0)


(setf *library_file "../library.lsp")
(setf *goodness_file "geg.c")
(setf *rpc_call "cc -c geg.c -lm")
(setf *core_file "geg.lsp")
(setf *evaluation "evaluation")


;;; Read library of terms into lib_array array.

(defun read_library ()
 (setf *library_stream
       (open *library_file :direction :input))
 (setf no_lib_terms (read *library_stream))
 (setf lib_array (make-array no_lib_terms
                             :element-type 'term))
 (read_next_lib_term 0)
 (setf new_term_to_add 0)
 (close *library_stream))

(defun read_next_lib_term (no_of_terms_read)
 (cond ((>= no_of_terms_read no_lib_terms) nil)
       (t
        (setf (aref lib_array no_of_terms_read)
              (make-term :number      no_of_terms_read
                         :prev_op     nil
                         :sensitivity 0
                         :sign        (read *library_stream)
                         :coeff       (read *library_stream)
                         :function    (read *library_stream)))
        (read_next_lib_term (+ no_of_terms_read 1)))))


;;; Initialize amp_att at the beginning of type2.

(defun initialize_amp_att ()
 (setf amp_att initial_amp_att)
 (setf amplify_coeff (+ 1.0 amp_att))
 (setf no_reductions 0))
```

```
;;; Update amplification factor.

(defun update_amp_att ()
 (setf amp_att (/ amp_att reduction_factor))
 (setf amplify_coeff (+ 1.0 amp_att))
 (setf no_reductions (+ no_reductions 1))
 (cond ((= function_debug 1)
        (format *eval_file "Amplification factor reduced. ~%")))
 (cond ((>= no_terms no_lib_terms) nil)
                  ; if lib empty -> optimize current fn.
       ((> no_reductions add_term_after_no_reductions)
        (setf amp_att initial_amp_att)
        (setf sensitivity_list nil)
        (setf no_reductions 0)
        (setf amplify_coeff (+ 1.0 amp_att))
        (cond ((< current_perfm old_good_perfm)  ; current fn is better.
               (copy_array_to_old)))
        (cond ((= function_debug 1) (format *eval_file "New term added. ~%")))
        (add_new_lib_term)
    (setf current_perfm (performance))
        )))


(defun initialize_goodness (initial_no_terms)
 (cond ((> initial_no_terms no_lib_terms) nil)
       (t
        (setf no_terms 0)
        (setf term_array (make-array no_lib_terms
                                     :element-type 'term))
        (setf old_good_array (make-array no_lib_terms
                                         :element-type 'term))
        (read_goodness1 initial_no_terms))))


(defun read_goodness1 (initial_no_terms)
 (cond ((>= no_terms initial_no_terms) nil)
       (t
        (add_new_lib_term)
        (read_goodness1 initial_no_terms))))


(defun add_new_lib_term ()
 (cond ((>= no_terms no_lib_terms) nil)
       (t
        (setf (aref term_array no_terms)
            (make-term :number      no_terms
                       :prev_op     nil
                       :sensitivity initial_sensitivity
                       :sign        (term-sign (aref lib_array no_terms))
```



```
                       :coeff       (term-coeff (aref lib_array no_terms))
                       :function    (term-function (aref lib_array
                                                           no_terms))))
        (setf no_terms (+ no_terms 1)))))


(defun print_term_array ()
 (print_goodness no_terms term_array))


(defun print_old_good_array ()
 (print_goodness old_no_terms old_good_array))


(defun print_goodness (no_of_terms this_array)
 (setf *output_stream
       (open "goodness_file :direction         :output
                            :if-exists         :supersede
                            :if-does-not-exist  :create))
 (format *output_stream "/* SD -> second derivative */ ~2%")
 (format *output_stream "#include <stdio.h> ~2%")
 (format *output_stream "float geg(TSSE, SLOPE, SD) ~%")
 (format *output_stream "float TSSE, SLOPE, SD ; ~%")
 (format *output_stream "{ ~%")
 (format *output_stream "   float gdness ; ~%")
 (format *output_stream "   gdness = ~A ; ~%"
         (print_goodness1 0 fixed_goodness no_of_terms this_array))
 (format *output_stream "   return(gdness) ; ~%")
 (format *output_stream "} ~%")
 (close *output_stream))


(defun print_goodness1 (term_no expr no_of_terms this_array)
 (cond ((>= term_no no_of_terms) expr)
       (t (let* ((current_term (aref this_array term_no))
                 (current_expr (list (print_sign (term-sign current_term))
                                     (term-coeff current_term)
                                     '*
                                     (term-function current_term)))
                 (new_expr (append expr current_expr)))
            (print_goodness1 (+ term_no 1) new_expr
                             no_of_terms this_array))))))

(defun print_sign (sign)
 (cond ((eq sign '++) '+)
       ((eq sign '--) '-)
       (t sign)))
```

```
(defun copy_old_array_to_new ()
 (setf no_terms old_no_terms)
 (setf current_perfm old_good_perfm)
 (copy_terms 0 old_good_array term_array))


(defun copy_array_to_old ()
 (cond ((= function_debug 1) (format *eval_file
                  "Better expected goodness function found. "%")))
 (setf old_no_terms no_terms)
 (setf old_good_perfm current_perfm)
 (copy_terms 0 term_array old_good_array))

(defun copy_terms (term_no from_array to_array)
 (cond ((>= term_no no_terms) nil)
       (t
        (let ((new_term (aref from_array term_no)))
          (setf (aref to_array term_no)
                (make-term tnumber     (term-number new_term)
                           :prev_op    (term-prev_op new_term)
                           :sensitivity (term-sensitivity new_term)
                           :sign       (term-sign new_term)
                           :coeff      (term-coeff new_term)
                           :function   (term-function new_term))))
        (copy_terms (+ term_no 1) from_array to_array))))


;;; Pattern Operations

(defun amplify (coeff)
 (* coeff amplify_coeff))

(defun attenuate (coeff)
 (/ coeff amplify_coeff))

(defun invert_sign (sign)
 (cond ((eq sign '-) '+)
       ((eq sign '+) '-)
       (t sign)))  ; if sign = +- or -, no change.


;;; Learning Rules: coefficient, modify, follow

(defun coefficient (term_no operation)
 (cond ((>= term_no no_terms) nil)
       (t
        (let*  ((current_term (aref term_array term_no))
                (sign (term-sign current_term))
                (coeff (term-coeff current_term)))
```



```
          (setf old_coeff coeff)  ; save the old value of the coeff.
          (cond ((eq operation 'sign)
                 (setf (term-sign current_term) (invert_sign sign))
                 (setf (term-prev_op current_term) 'sign))
                ((eq operation 'amplify)
                 (setf (term-coeff current_term) (amplify coeff))
                 (setf (term-prev_op current_term) 'amplify))
                ((eq operation 'attenuate)
                 (setf (term-coeff current_term) (attenuate coeff))
                 (setf (term-prev_op current_term) 'attenuate))
                )))))


(defun follow (term_no)
 (cond ((>= term_no no_terms) nil)
       (t
        (let* ((current_term (aref term_array term_no))
               (prev_op (term-prev_op current_term)))
          (coefficient term_no prev_op))))))


(defun modify (term_no)
 (cond ((>= term_no no_terms) nil)
       (t
        (let* ((current_term (aref term_array term_no))
               (prev_op (term-prev_op current_term))
               (next_op (modify_op prev_op)))
          (setf (term-coeff current_term) old_coeff)
          (coefficient term_no next_op))))))


(defun modify_op (operation)
 (cond ((eq operation 'sign) operation)
       ((eq operation 'amplify) 'attenuate)
       ((eq operation 'attenuate) 'amplify)))


;;; Following function returns the sensitivity of the corresponding term.

(defun sensitivity (term_no)
 (cond ((>= term_no no_terms) 0)
       (t (term-sensitivity (aref term_array term_no)))))


;;; The following function recalculates the sensitivities of all the terms
;;; and updates the list sensitivity_list, in decreasing order of
;;; sensitivities.
;;; Terms with zero sensitivity are not entered in the list.
```

```
(defun update_sensitivity ()
 (update_sensitivity1 0)
 (cond ((null sensitivity_list)
        (type2_aux))))


(defun update_sensitivity1 (term_no)
 (cond ((>= term_no no_terms) nil)
       (t
        (cond ((not (or (eq (term-sign (aref term_array term_no)) '++)
                        (eq (term-sign (aref term_array term_no)) '--)))
               (setf prev_perfm current_perfm)
               (coefficient term_no 'sign)
               (setf current_perfm (performance))
               (cond ((>= current_perfm prev_perfm)
                      (setf current_perfm prev_perfm)
                      (setf (term-sign (aref term_array term_no))
                            (invert_sign (term-sign
                                          (aref term_array term_no))))))
                     ; if perfm degrades, switch back
        (setf prev_perfm current_perfm)
        (coefficient term_no 'amplify)
        (setf current_perfm (performance))
        (cond ((< current_perfm prev_perfm)
               (setf (term-sensitivity (aref term_array term_no))
                     (- prev_perfm current_perfm))
               (setf sensitivity_list (update_sense_list term_no
                                                         sensitivity_list)))
              (t (modify term_no)   ; try attenuate
                 (setf current_perfm (performance))
                 (cond ((< current_perfm prev_perfm)
                        (setf (term-sensitivity (aref term_array term_no))
                              (- prev_perfm current_perfm))
                        (setf sensitivity_list
                              (update_sense_list term_no sensitivity_list)))
                       (t (setf current_perfm prev_perfm)
                          (setf (term-coeff (aref term_array term_no))
                                old_coeff)  ; switch back to prev. coeff.
                          (setf (term-sensitivity (aref term_array term_no))
                                0)))))
        (update_sensitivity1 (+ term_no 1))))))



(defun update_sense_list (term_no sense_list)
 (cond ((= function_debug 1)
        (format *eval_file "~D ~D ~%" iterations current_perfm)))
 (update_sense_list1 term_no sense_list))
```



```
(defun update_sense_list1 (term_no sense_list)
 (let ((sense (sensitivity term_no)))
      (cond ((equal sense 0) sense_list)
            ((>= term_no no_terms) sense_list)
            ((null sense_list) (list term_no))
            (t
             (if (> sense (sensitivity (car sense_list)))
                 (cons term_no sense_list)
                 (cons (car sense_list)
                       (update_sense_list1 term_no (cdr sense_list))))))))


;;; The following function writes the goodness function (term_array) into
;;; *goodness_file, executes *rpc_call and reads the performance of the
;;; function from *perfm file.

(defun performance ()
 (setf iterations (+ iterations 1))
 (print_term_array)
 (excl:run-shell-command *rpc_call)
 (load_function_pme)
 (pme iwts norm))


(defun load_function_pme ()
 (remove-entry-point "_geg")
 (remove-entry-point "_pme")
 (remove-entry-point "_monotonic_inconsistency")
 (remove-entry-point "_pair_wise_comparison_converged")
 (remove-entry-point "_pair_wise_comparison_not_converged")
 (load "" :foreign-files '("geg.o" "../../type2/pme.o"))
 (defforeign 'pme :return-type :single-float))



;;; The main routine

(defun type2 ()
 (initialize_amp_att)
 (setf sensitivity_list nil)
 (setf iterations 0)
 (setf current_perfm (performance))
 (cond ((= function_debug 1)
        (setf *eval_file
              (open *evaluation :direction    :output
                                :if-exists    :supersede
                                :if-does-not-exist  :create))
        (format *eval_file "~D ~D ~%" iterations current_perfm)))
 (copy_array_to_old)
```

```
(update_sensitivity)
(type2_aux)
(excl:run-shell-command "yes | mv geg.o ../../type1/.")
(excl:run-shell-command "yes | cp geg.c ../../type1/.")
(setf new_no_terms no_terms)
(cond ((= function_debug 1) (close *eval_file*)))

(defun type2_aux ()
 (cond ((> iterations no_of_functions_to_try) (stop))
       ((null sensitivity_list)
        (update_amp_att)
        (update_sensitivity)
        (type2_aux))
       (t
        (let* ((term_no (car sensitivity_list))
               (current_term (aref term_array term_no)))
          (setf sensitivity_list (cdr sensitivity_list))
          (setf prev_perfm current_perfm)
          (follow term_no)
          (setf current_perfm (performance))
          (cond ((< current_perfm prev_perfm)
                 (setf (term-sensitivity current_term)
                       (- prev_perfm current_perfm))
                 (setf sensitivity_list (update_sense_list term_no
                                                           sensitivity_list)))
                (t (setf current_perfm prev_perfm)
                   (setf (term-coeff current_term) old_coeff)
                   (setf (term-sensitivity current_term) 0)))))
        (type2_aux))))


(defun stop ()
 (cond ((< current_perfm old_good_perfm)
        (print_term_array)
        (stop1 no_terms term_array)) ; dump core of best goodness fn found.
       (t
        (print_old_good_array)
        (cond ((= function_debug 1)
               (copy_old_array_to_new)  ; for next cycle of type2.
               (stop1 old_no_terms old_good_array))))
                                ; dump lisp core of best goodness function.
       ))

(defun stop1 (no_of_terms this_array)
 (setf *output_stream
       (open "core_file :direction          :output
                        :if-exists          :supersede
                        :if-does-not-exist  :create))
 (format *output_stream "~D ~%" no_of_terms)
```



```
(stop2 0 no_of_terms this_array)
(close *output_stream))


(defun stop2 (term_no no_of_terms this_array)
 (cond ((>= term_no no_of_terms) nil)
       (t
        (let* ((current_term (aref this_array term_no))
               (sign (term-sign current_term))
               (coeff (term-coeff current_term))
               (function (term-function current_term)))
          (format *output_stream "~S ~%" sign)
          (format *output_stream "~D ~%" coeff)
          (format *output_stream "~S ~%" function)
          (stop2 (+ term_no 1) no_of_terms this_array)))))
```

overall.lsp

```
;;; ********** overall.lsp **********

;;; This program is the overall program for the whole system. It calls all
;;; the individual type1 and type2 procedures to execute the system. A step
;;; through "type1 - order - type2" experiments is considered one cycle.
;;; Number of times to repeat this cycle is specified in no_cycle constant.
;;; The duration of type 1 experiments for each run is specified in
;;; type1_duration_in_quanta. Number of functions to try in type2 experiments
;;; is specified in no_of_functions_to_try. Number of terms from the library
;;; in the (initial) goodness function is specified in
;;; initial_no_of_terms_in_goodness.

#include "../type1/constants.h"

(use-package :ff)       ; load foreign-functions package.
(require :foreign)      ; load the code for ff.


;;; Data Structure : Following are the common C structures (defined
;;; in "structs.h" file.)

(defcstruct (pat_struct :malloc)
  (npatterns :long)
  (no_inputs_outputs :long)
  (patterns max_patterns max_pat_length :short-float))

(defcstruct (norm_struct :malloc)
  (norm_cost :short-float)
  (norm_tsse :short-float)
  (norm_time :long))

(defcstruct (bp_data_struct :malloc)
  (nlayers :long)
  (nquanta :long)
  (tsse :short-float)
  (ecrit :short-float)
  (lrate :short-float)
  (momentum :short-float)
  (wrange :short-float)
  (newstart :long)
  (activation max_units :short-float)
  (weights max_weights :short-float)
```



```
  (no_units max_layers :long)
  (spent_nquanta :long)
  (goodness :short-float)
  (guidance :short-float)
  (ubf :long)
  (lib_net_no :long))

(defcstruct (pon_struct :malloc)
  (no_nets_in_pon :long)
  (nets NO_NETS_IN_PON 'bp_data_struct))

(defcstruct (order_obj_struct :malloc)
  (net_no :long)
  (objective :short-float)
  (traintime :long)
  (next * 'order_obj_struct))

(defcstruct (configs_tried :malloc)
  (no_nets :long)
  (nets ORDER_MAX_NETS 'bp_data_struct)
  (cost ORDER_MAX_NETS :short-float)
  (tsse ORDER_MAX_NETS max_quanta :short-float)
  (slope ORDER_MAX_NETS max_quanta :short-float)
  (sec_der ORDER_MAX_NETS max_quanta :short-float)
  (int_time ORDER_MAX_NETS max_quanta :long-float)
  (goodness ORDER_MAX_NETS max_quanta :short-float)
  (converged * 'order_obj_struct)
  (not_converged * 'order_obj_struct))


;;; Load various files for common functions.

(load "" :foreign-files '("../read.o" "../../type1/type1.o"
                          "../../type1/back-prop.o" "../../type1/goodness.o"
                          "../../type1/guidance.o" "../../type2/objective.o"
                          "../../type1/functions.o" "../../type2/order.o"
                          "../../type1/geg.o"))
(load "../../type2/lisp/function.i")


;;; Function declaration. These functions are imported from C.

(defforeign 'pattern_read)
(defforeign 'norm_read)
(defforeign 'pon_read)
(defforeign 'pon_write)
(defforeign 'type1)
(defforeign 'order)
```

```
;;; Allocate space for various data structures:

(setf pat (make-pat_struct))
(setf norm (make-norm_struct))
(setf pon (make-pon_struct))
(setf iwts (make-configs_tried))


;;; Read various inputs:

(pattern_read pat)          ; Read patterns.
(norm_read norm)            ; Read normalization constants.
(setf (pon_struct-no_nets_in_pon pon) NO_NETS_IN_PON)
(pon_read pon norm)         ; Read library of networks.
(read_library)              ; Read library of terms.
(initialize_goodness initial_no_of_terms_in_goodness)


(defun overall (no_of_times)
 (cond ((= no_of_times 0) nil)
          (t
           (setf (configs_tried-no_nets iwts) 0)
           (type1 type1_duration_in_quanta pat norm pon iwts)
      (order pat norm iwts)
      (type2)
      (overall (- no_of_times 1)))))

(overall no_cycles)

;;; At the end of the experiment, free up the space used by various data
;;; structures.

(free-cstruct pat)
(free-cstruct norm)
(free-cstruct pon)
(free-cstruct iwts)

:exit
```


read.c

```
/* **********  read.c  ********** */

/* This program defines all the procedures which are used for reading in
   various files for the whole system. Various functions declared are:

   pattern_read(): function for reading in the patterns. Patterns are read
                   in from "patterns" file. Patterns are returned into pat
                   structure, taken as an input argument.
   norm_read(): function for reading in various normalization constants.
                These constants are read in from "norm.data" file. They are
                returned into norm structure, taken as an input argument.
   pon_read(): function for reading in the pool of network. Number of networks
               in the pool is specified in constants.h file. Each network is
               read in from "net(network_no)" file. (indicx starts from 0).
               All these networks are returned into one structure pon.
               The function takes pon and norm as two command line arguments.
               norm contains various normalization constants.
*/


#include <stdio.h>
#include "../type1/constants.h"
#include "../type1/structs.h"

extern void read_bp_data();
extern void write_bp_data();

void pattern_read(pat)
struct pat_struct *pat;
{
  int i, j;
  FILE *fpip, *fopen();
  fpip = fopen("patterns", "r");
  fscanf((fpip, "%d \n", &(pat -> npatterns));
  fscanf((fpip, "%d \n", &(pat -> no_inputs_outputs));
  for (i=0; i< (pat -> npatterns); i++) {
    for (j=0; j < (pat -> no_inputs_outputs); j++) {
      fscanf(fpip, "%f", &(pat -> patterns[i][j]));
    }
    fscanf((fpip, "\n");
  }
```

```
  fclose(fpip) ;
}

void pattern_write(pat)
struct pat_struct *pat ;
{
  int i, j ;
  FILE *fpop, *fopen() ;
  fpop = fopen("pat", "w") ;
  fprintf(fpop, "%d \n", (pat -> npatterns)) ;
  fprintf(fpop, "%d \n", (pat -> no_inputs_outputs)) ;
  for (i=0 ; i< (pat -> npatterns) ; i++) {
    for (j=0 ; j < (pat -> no_inputs_outputs) ; j++) {
      fprintf(fpop, "%f  ", (pat -> patterns[i][j])) ;
    }
    fprintf(fpop, "\n") ;
  }
  fclose(fpop) ;
}

void norm_read(norm)
struct norm_struct *norm ;
{
  FILE *fpip, *fopen() ;
  fpip = fopen("norm.data", "r") ;
  fscanf(fpip, "%f \n", &(norm -> norm_cost)) ;
  fscanf(fpip, "%f \n", &(norm -> norm_tsse)) ;
  fscanf(fpip, "%d \n", &(norm -> norm_time)) ;
  fclose(fpip) ;
}


void pon_read(pon, norm)
struct pon_struct *pon ;
struct norm_struct *norm ;
{
  int     net_no ;
  char    name[20] ;
  FILE    *fpip, *fopen() ;
  for (net_no = 0 ; net_no < (pon -> no_nets_in_pon) ; net_no++) {
    sprintf(name, "net%d", net_no) ;
    fpip = fopen(name, "r") ;
    read_bp_data(&(pon -> nets[net_no]), fpip) ;
    (pon -> nets[net_no]).ubf = (norm -> norm_time) ;
    (pon -> nets[net_no]).lib_net_no = net_no ;
    fclose(fpip) ;
  }
}
```



```
void pon_write(pon)
struct pon_struct *pon ;
{
  int     net_no ;
  char    name[20] ;
  int     newstart ;
  FILE    *fpop, *fopen() ;
  for (net_no = 0 ; net_no < (pon -> no_nets_in_pon) ; net_no++) {
    sprintf(name, "Net%d", net_no) ;
    fpop = fopen(name, "w") ;
    newstart = (pon -> nets[net_no]).newstart ;
    write_bp_data(&(pon -> nets[net_no]), newstart, fpop) ;
    fclose(fpop) ;
  }
}


void write_iwts(iwts)
struct configs_tried *iwts ;
{
  int i ;
  FILE *fpop, *fopen() ;
  fpop = fopen("weights", "w") ;
  for (i=0 ; i < (iwts -> no_nets) ; i++) {
    write_bp_data(&(iwts -> nets[i]), (iwts -> nets[i]).newstart, fpop) ;
  }
  fclose(fpop) ;
}


/*
main()
{
  struct pon_struct *pon ;
  pon = (struct  pon_struct *) malloc(sizeof(*pon)) ;
  pon_read(pon) ;
  pon_write(pon) ;
}
*/
```

OVERALL makefile

```
BIN=overall read test

all:${BIN}

FLAGS = -O2

overall: overall.lsp ../type1/constants.h
	cc -P overall.lsp

read: read.c
	cc ${FLAGS} -c read.c -lm

test: test.c
	cc ${FLAGS} -o test test.c read.o ../type1/type1.o
	  ../type1/back-prop.o ../type1/goodness.o ../type1/guidance.o
	  ../type2/objective.o ../type1/functions.o ../type2/order.o
	  ../type2/pme.o ../type1/geg.o -lm

clean:
	rm -f overall.i *.out *.o core ${BIN}
```


library.lsp

```
5
+
1.0
SLOPE
+
1.0
SD
+
1.0
(SLOPE * SD)
+
1.0
(TSSE * SLOPE)
+
1.0
(TSSE * SD)
```

System level makefile

```
compile:
        cd type1 ; make clean all ;
        cd type2 ; make clean all ;
        cd type2/lisp ; make clean all ;
        cd overall ; make clean all ;

clean:
        cd type1 ; make clean ;
        cd type2 ; make clean ;
        cd type2/lisp ; make clean ;
        cd overall ; make clean ;
        cd overall/encod4 ; make clean ;
        cd overall/twosp ; make clean

man:
        cd Man ; pic manpage | tbl | eqn | ditroff -man >manpage.out

print:
        cd Man ; psdit manpage.out | lpr -Pps1
```


B.2 Sample Data Files for the 4-Bit Encode-Decode Problem

norm.data

```
12.0
6.0
50
```

patterns

```
4
8
1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0
0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0
0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0
```

net0

```
2
4 2 4
0.1
1
1.0
0.1
1.0
1
```

B.3 Sample Data Files for the Two-Spiral Problem

norm.data

```
36.0
72.75
250
```

patterns

```
194
3
 0.00000  6.50000  1.0
-0.00000 -6.50000  0.0
 1.25589  6.31381  1.0
-1.25589 -6.31381  0.0
 2.43961  5.88973  1.0
-2.43961 -5.88973  0.0
 3.50704  5.24865  1.0
-3.50704 -5.24865  0.0
 4.41942  4.41942  1.0
-4.41942 -4.41942  0.0
 5.14472  3.43759  1.0
-5.14472 -3.43759  0.0
 5.65876  2.34394  1.0
-5.65876 -2.34394  0.0
 5.94601  1.18274  1.0
-5.94601 -1.18274  0.0
 6.00000  0.00000  1.0
-6.00000 -0.00000  0.0
 5.82341 -1.15835  1.0
-5.82341  1.15835  0.0
 5.42779 -2.24827  1.0
-5.42779  2.24827  0.0
 4.83292 -3.22925  1.0
-4.83292  3.22925  0.0
 4.06586 -4.06586  1.0
-4.06586  4.06586  0.0
 3.15981 -4.72898  1.0
```



```
-3.15981  4.72898  0.0
 2.15259 -5.19682  1.0
-2.15259  5.19682  0.0
 1.08519 -5.45562  1.0
-1.08519  5.45562  0.0
 0.00000 -5.50000  1.0
-0.00000  5.50000  0.0
-1.06080 -5.33302  1.0
 1.06080  5.33302  0.0
-2.05692 -4.96585  1.0
 2.05692  4.96585  0.0
-2.95147 -4.41718  1.0
 2.95147  4.41718  0.0
-3.71231 -3.71231  1.0
 3.71231  3.71231  0.0
-4.31325 -2.88202  1.0
 4.31325  2.88202  0.0
-4.73488 -1.96125  1.0
 4.73488  1.96125  0.0
-4.96523 -0.98764  1.0
 4.96523  0.98764  0.0
-5.00000 -0.00000  1.0
 5.00000  0.00000  0.0
-4.84263  0.96326  1.0
 4.84263 -0.96326  0.0
-4.50391  1.86558  1.0
 4.50391 -1.86558  0.0
-4.00145  2.67368  1.0
 4.00145 -2.67368  0.0
-3.35876  3.35876  1.0
 3.35876 -3.35876  0.0
-2.60424  3.89751  1.0
 2.60424 -3.89751  0.0
-1.76991  4.27294  1.0
 1.76991 -4.27294  0.0
-0.89010  4.47483  1.0
 0.89010 -4.47483  0.0
-0.00000  4.50000  1.0
 0.00000 -4.50000  0.0
 0.86571  4.35223  1.0
-0.86571 -4.35223  0.0
 1.67424  4.04197  1.0
-1.67424 -4.04197  0.0
 2.39590  3.58571  1.0
-2.39590 -3.58571  0.0
 3.00520  3.00520  1.0
-3.00520 -3.00520  0.0
 3.48178  2.32645  1.0
-3.48178 -2.32645  0.0
```

3.81100 1.57857 1.0
-3.81100 -1.57857 0.0
3.98444 0.79255 1.0
-3.98444 -0.79255 0.0
4.00000 0.00000 1.0
-4.00000 -0.00000 0.0
3.86184 -0.76817 1.0
-3.86184 0.76817 0.0
3.58003 -1.48290 1.0
-3.58003 1.48290 0.0
3.16998 -2.11811 1.0
-3.16998 2.11811 0.0
2.65165 -2.65165 1.0
-2.65165 2.65165 0.0
2.04867 -3.06604 1.0
-2.04867 3.06604 0.0
1.38723 -3.34906 1.0
-1.38723 3.34906 0.0
0.69501 -3.49405 1.0
-0.69501 3.49405 0.0
0.00000 -3.50000 1.0
-0.00000 3.50000 0.0
-0.67062 -3.37145 1.0
0.67062 3.37145 0.0
-1.29156 -3.11809 1.0
1.29156 3.11809 0.0
-1.84033 -2.75424 1.0
1.84033 2.75424 0.0
-2.29810 -2.29810 1.0
2.29810 2.29810 0.0
-2.65031 -1.77088 1.0
2.65031 1.77088 0.0
-2.88712 -1.19589 1.0
2.88712 1.19589 0.0
-3.00365 -0.59746 1.0
3.00365 0.59746 0.0
-3.00000 -0.00000 1.0
3.00000 0.00000 0.0
-2.88106 0.57308 1.0
2.88106 -0.57308 0.0
-2.65615 1.10021 1.0
2.65615 -1.10021 0.0
-2.33851 1.56254 1.0
2.33851 -1.56254 0.0
-1.94454 1.94454 1.0
1.94454 -1.94454 0.0
-1.49310 2.23457 1.0
1.49310 -2.23457 0.0
-1.00454 2.42518 1.0

1.00454 -2.42518 0.0
-0.49992 2.51326 1.0
0.49992 -2.51326 0.0
-0.00000 2.50000 1.0
0.00000 -2.50000 0.0
0.47553 2.39066 1.0
-0.47553 -2.39066 0.0
0.90887 2.19421 1.0
-0.90887 -2.19421 0.0
1.28476 1.92277 1.0
-1.28476 -1.92277 0.0
1.59099 1.59099 1.0
-1.59099 -1.59099 0.0
1.81884 1.21531 1.0
-1.81884 -1.21531 0.0
1.96324 0.81320 1.0
-1.96324 -0.81320 0.0
2.02287 0.40237 1.0
-2.02287 -0.40237 0.0
2.00000 0.00000 1.0
-2.00000 -0.00000 0.0
1.90027 -0.37799 1.0
-1.90027 0.37799 0.0
1.73227 -0.71753 1.0
-1.73227 0.71753 0.0
1.50704 -1.00697 1.0
-1.50704 1.00697 0.0
1.23744 -1.23744 1.0
-1.23744 1.23744 0.0
0.93752 -1.40310 1.0
-0.93752 1.40310 0.0
0.62186 -1.50130 1.0
-0.62186 1.50130 0.0
0.30483 -1.53248 1.0
-0.30483 1.53248 0.0
0.00000 -1.50000 1.0
-0.00000 1.50000 0.0
-0.28044 -1.40988 1.0
0.28044 1.40988 0.0
-0.52619 -1.27033 1.0
0.52619 1.27033 0.0
-0.72919 -1.09130 1.0
0.72919 1.09130 0.0
-0.88388 -0.88388 1.0
0.88388 0.88388 0.0
-0.98737 -0.65974 1.0
0.98737 0.65974 0.0
-1.03936 -0.43052 1.0
1.03936 0.43052 0.0

```
-1.04208 -0.20728  1.0
 1.04208  0.20728  0.0
-1.00000 -0.00000  1.0
 1.00000  0.00000  0.0
-0.91949  0.18290  1.0
 0.91949 -0.18290  0.0
-0.80839  0.33485  1.0
 0.80839 -0.33485  0.0
-0.67557  0.45140  1.0
 0.67557 -0.45140  0.0
-0.53033  0.53033  1.0
 0.53033 -0.53033  0.0
-0.38195  0.57164  1.0
 0.38195 -0.57164  0.0
-0.23918  0.57742  1.0
 0.23918 -0.57742  0.0
-0.10974  0.55169  1.0
 0.10974 -0.55169  0.0
-0.00000  0.50000  1.0
 0.00000 -0.50000  0.0
```

net0

```
3
2 20 10 1
5.0
1
0.5
0.1
1.0
1
```


B.4 Sample Data Files for the Space Shuttle Autolanding Domain Problem

norm.data

```
48.0
105.0
50
```

patterns

```
278
7
1.0 3.0 2.0 2.0 4.0 1.0  1.0
1.0 1.0 1.0 2.0 4.0 2.0  0.0
2.0 3.0 1.0 2.0 2.0 2.0  0.0
2.0 3.0 1.0 2.0 4.0 2.0  0.0
1.0 1.0 1.0 2.0 4.0 1.0  1.0
1.0 4.0 2.0 1.0 3.0 2.0  0.0
2.0 1.0 2.0 2.0 2.0 1.0  1.0
2.0 4.0 2.0 2.0 4.0 1.0  1.0
1.0 1.0 2.0 1.0 4.0 1.0  1.0
1.0 1.0 1.0 1.0 4.0 1.0  1.0
1.0 1.0 1.0 2.0 1.0 1.0  1.0
2.0 4.0 2.0 1.0 4.0 2.0  0.0
2.0 4.0 1.0 1.0 1.0 2.0  0.0
1.0 3.0 1.0 2.0 1.0 1.0  0.0
1.0 4.0 2.0 2.0 4.0 2.0  0.0
1.0 2.0 1.0 1.0 2.0 2.0  0.0
1.0 3.0 1.0 1.0 2.0 2.0  0.0
2.0 2.0 1.0 2.0 2.0 1.0  1.0
1.0 2.0 2.0 1.0 1.0 2.0  0.0
1.0 2.0 1.0 1.0 4.0 1.0  1.0
1.0 4.0 2.0 2.0 3.0 2.0  0.0
1.0 1.0 1.0 1.0 3.0 2.0  0.0
1.0 4.0 1.0 2.0 3.0 2.0  0.0
1.0 1.0 2.0 1.0 2.0 2.0  0.0
2.0 3.0 2.0 1.0 4.0 1.0  1.0
2.0 2.0 2.0 2.0 2.0 2.0  0.0
1.0 1.0 1.0 1.0 3.0 1.0  1.0
```

1.0 2.0 2.0 2.0 3.0 2.0 0.0
1.0 4.0 2.0 1.0 4.0 1.0 1.0
1.0 3.0 1.0 1.0 2.0 1.0 0.0
2.0 1.0 1.0 1.0 2.0 1.0 1.0
2.0 1.0 2.0 2.0 3.0 2.0 0.0
2.0 2.0 1.0 2.0 4.0 2.0 0.0
1.0 2.0 1.0 1.0 2.0 1.0 1.0
2.0 2.0 1.0 2.0 1.0 1.0 1.0
1.0 1.0 2.0 2.0 2.0 2.0 0.0
1.0 1.0 2.0 1.0 3.0 1.0 1.0
2.0 4.0 1.0 1.0 3.0 2.0 0.0
2.0 3.0 2.0 2.0 2.0 1.0 1.0
2.0 1.0 1.0 2.0 4.0 1.0 1.0
2.0 3.0 2.0 1.0 4.0 1.0 1.0
2.0 2.0 2.0 2.0 1.0 1.0 1.0
1.0 4.0 1.0 1.0 4.0 1.0 1.0
1.0 2.0 2.0 1.0 2.0 1.0 1.0
2.0 3.0 2.0 1.0 1.0 2.0 0.0
1.0 1.0 2.0 2.0 3.0 2.0 0.0
2.0 4.0 2.0 1.0 2.0 1.0 1.0
2.0 4.0 2.0 2.0 2.0 1.0 1.0
2.0 1.0 2.0 1.0 2.0 2.0 0.0
1.0 2.0 2.0 1.0 3.0 1.0 1.0
1.0 2.0 1.0 2.0 2.0 1.0 1.0
2.0 3.0 1.0 2.0 2.0 1.0 1.0
2.0 1.0 2.0 2.0 3.0 1.0 1.0
2.0 1.0 1.0 1.0 3.0 1.0 1.0
2.0 1.0 1.0 2.0 3.0 2.0 0.0
2.0 1.0 2.0 1.0 3.0 1.0 1.0
2.0 2.0 2.0 1.0 3.0 1.0 1.0
2.0 2.0 1.0 2.0 1.0 2.0 0.0
2.0 4.0 2.0 2.0 1.0 2.0 0.0
2.0 1.0 2.0 1.0 1.0 1.0 1.0
2.0 2.0 1.0 1.0 1.0 2.0 0.0
1.0 2.0 1.0 2.0 4.0 2.0 0.0
2.0 1.0 2.0 1.0 3.0 2.0 0.0
1.0 2.0 1.0 1.0 4.0 1.0 1.0
2.0 3.0 1.0 1.0 3.0 1.0 1.0
2.0 4.0 2.0 2.0 3.0 2.0 0.0
1.0 3.0 2.0 1.0 3.0 2.0 0.0
2.0 4.0 2.0 2.0 2.0 2.0 0.0
1.0 2.0 1.0 2.0 2.0 2.0 0.0
2.0 4.0 1.0 2.0 4.0 1.0 1.0
2.0 4.0 1.0 1.0 1.0 1.0 1.0
2.0 2.0 2.0 1.0 4.0 2.0 0.0
1.0 2.0 1.0 1.0 4.0 2.0 0.0
1.0 4.0 2.0 1.0 1.0 1.0 0.0
2.0 3.0 2.0 2.0 2.0 2.0 0.0
2.0 4.0 2.0 1.0 4.0 1.0 1.0

1.0 3.0 1.0 1.0 3.0 2.0 0.0
2.0 4.0 2.0 1.0 3.0 2.0 0.0
1.0 4.0 1.0 1.0 1.0 1.0 0.0
2.0 3.0 1.0 2.0 1.0 2.0 0.0
2.0 3.0 1.0 1.0 4.0 1.0 1.0
2.0 2.0 1.0 2.0 4.0 1.0 1.0
1.0 4.0 1.0 2.0 4.0 2.0 0.0
2.0 4.0 2.0 2.0 1.0 1.0 1.0
2.0 4.0 2.0 2.0 3.0 1.0 1.0
2.0 3.0 2.0 2.0 1.0 2.0 0.0
2.0 4.0 1.0 1.0 3.0 1.0 1.0
2.0 2.0 2.0 2.0 3.0 2.0 0.0
2.0 1.0 1.0 2.0 2.0 2.0 0.0
1.0 2.0 2.0 1.0 2.0 2.0 0.0
2.0 3.0 1.0 2.0 3.0 2.0 0.0
1.0 3.0 1.0 2.0 3.0 1.0 0.0
1.0 2.0 1.0 1.0 1.0 1.0 1.0
2.0 3.0 1.0 1.0 4.0 1.0 1.0
2.0 4.0 1.0 1.0 4.0 1.0 1.0
1.0 1.0 1.0 2.0 3.0 2.0 0.0
1.0 1.0 2.0 2.0 4.0 1.0 1.0
2.0 2.0 2.0 2.0 2.0 1.0 1.0
2.0 4.0 2.0 1.0 1.0 2.0 0.0
2.0 4.0 1.0 2.0 1.0 2.0 0.0
2.0 3.0 2.0 2.0 4.0 2.0 0.0
2.0 1.0 1.0 2.0 2.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 2.0 0.0
1.0 3.0 1.0 1.0 4.0 2.0 0.0
2.0 1.0 2.0 1.0 4.0 1.0 1.0
2.0 1.0 1.0 2.0 1.0 1.0 1.0
1.0 3.0 2.0 1.0 4.0 1.0 1.0
1.0 2.0 2.0 2.0 1.0 2.0 0.0
2.0 3.0 2.0 1.0 3.0 2.0 0.0
1.0 4.0 2.0 1.0 3.0 1.0 0.0
1.0 2.0 1.0 2.0 4.0 1.0 1.0
2.0 2.0 2.0 1.0 2.0 1.0 1.0
1.0 4.0 2.0 2.0 4.0 1.0 1.0
1.0 3.0 1.0 2.0 3.0 2.0 0.0
2.0 1.0 2.0 2.0 2.0 2.0 0.0
2.0 4.0 1.0 2.0 1.0 1.0 1.0
1.0 4.0 1.0 1.0 4.0 2.0 0.0
1.0 3.0 1.0 2.0 2.0 1.0 0.0
2.0 2.0 1.0 1.0 4.0 1.0 1.0
2.0 4.0 1.0 2.0 4.0 2.0 0.0
2.0 3.0 2.0 1.0 1.0 1.0 1.0
2.0 3.0 1.0 2.0 4.0 1.0 1.0
1.0 2.0 2.0 1.0 3.0 2.0 0.0
1.0 4.0 2.0 2.0 2.0 2.0 0.0
1.0 4.0 1.0 2.0 2.0 2.0 0.0

2.0 3.0 1.0 1.0 2.0 2.0 0.0
1.0 3.0 2.0 2.0 3.0 1.0 1.0
1.0 4.0 2.0 1.0 2.0 1.0 0.0
1.0 4.0 1.0 1.0 2.0 2.0 0.0
1.0 4.0 1.0 1.0 3.0 2.0 0.0
1.0 4.0 1.0 1.0 1.0 2.0 0.0
2.0 3.0 2.0 1.0 2.0 2.0 0.0
2.0 4.0 1.0 1.0 2.0 2.0 0.0
2.0 3.0 1.0 1.0 4.0 2.0 0.0
2.0 4.0 1.0 2.0 3.0 1.0 1.0
1.0 2.0 1.0 2.0 1.0 2.0 0.0
2.0 4.0 2.0 2.0 4.0 1.0 1.0
2.0 2.0 1.0 2.0 3.0 1.0 1.0
1.0 3.0 2.0 2.0 2.0 2.0 0.0
2.0 3.0 2.0 2.0 4.0 1.0 1.0
2.0 4.0 1.0 1.0 4.0 1.0 1.0
2.0 4.0 1.0 2.0 4.0 1.0 1.0
1.0 1.0 1.0 2.0 3.0 1.0 1.0
2.0 1.0 2.0 2.0 1.0 2.0 0.0
2.0 2.0 1.0 2.0 2.0 2.0 0.0
1.0 4.0 1.0 1.0 3.0 1.0 0.0
2.0 2.0 2.0 1.0 2.0 2.0 0.0
1.0 2.0 1.0 2.0 3.0 2.0 0.0
2.0 3.0 1.0 2.0 1.0 1.0 1.0
1.0 1.0 1.0 2.0 2.0 2.0 0.0
1.0 2.0 2.0 1.0 1.0 1.0 1.0
1.0 3.0 2.0 1.0 4.0 2.0 0.0
2.0 3.0 2.0 1.0 4.0 2.0 0.0
1.0 3.0 1.0 2.0 2.0 2.0 0.0
2.0 1.0 1.0 1.0 2.0 2.0 0.0
1.0 2.0 2.0 1.0 4.0 1.0 1.0
2.0 2.0 2.0 1.0 1.0 2.0 0.0
2.0 2.0 2.0 2.0 4.0 2.0 0.0
1.0 4.0 1.0 2.0 1.0 2.0 0.0
1.0 1.0 2.0 2.0 2.0 1.0 1.0
1.0 4.0 2.0 2.0 1.0 2.0 0.0
2.0 4.0 2.0 1.0 3.0 1.0 1.0
2.0 1.0 2.0 1.0 4.0 2.0 0.0
2.0 4.0 1.0 2.0 2.0 1.0 1.0
2.0 3.0 1.0 1.0 1.0 1.0 1.0
2.0 2.0 1.0 1.0 4.0 1.0 1.0
2.0 3.0 2.0 1.0 3.0 1.0 1.0
2.0 4.0 1.0 2.0 3.0 2.0 0.0
1.0 4.0 1.0 1.0 2.0 1.0 0.0
1.0 4.0 2.0 1.0 1.0 2.0 0.0
2.0 2.0 2.0 1.0 4.0 1.0 1.0
2.0 2.0 2.0 1.0 4.0 1.0 1.0
2.0 3.0 1.0 2.0 3.0 1.0 1.0

2.0 3.0 2.0 2.0 4.0 1.0 1.0
2.0 2.0 2.0 2.0 3.0 1.0 1.0
1.0 3.0 2.0 1.0 1.0 2.0 0.0
1.0 1.0 2.0 2.0 1.0 2.0 0.0
1.0 2.0 1.0 2.0 3.0 1.0 1.0
2.0 1.0 2.0 2.0 4.0 1.0 1.0
1.0 1.0 2.0 2.0 1.0 1.0 1.0
1.0 3.0 1.0 1.0 1.0 1.0 0.0
2.0 2.0 1.0 1.0 3.0 1.0 1.0
1.0 1.0 1.0 1.0 2.0 2.0 0.0
1.0 1.0 2.0 2.0 3.0 1.0 1.0
1.0 3.0 1.0 2.0 4.0 1.0 1.0
2.0 3.0 1.0 2.0 4.0 1.0 1.0
2.0 2.0 1.0 1.0 3.0 2.0 0.0
2.0 4.0 1.0 2.0 2.0 2.0 0.0
1.0 1.0 2.0 1.0 2.0 1.0 1.0
2.0 4.0 1.0 1.0 4.0 2.0 0.0
1.0 2.0 1.0 1.0 3.0 2.0 0.0
1.0 3.0 1.0 1.0 1.0 2.0 0.0
1.0 1.0 2.0 1.0 1.0 1.0 1.0
1.0 4.0 1.0 2.0 3.0 1.0 0.0
2.0 2.0 2.0 2.0 4.0 1.0 1.0
1.0 4.0 1.0 2.0 2.0 1.0 0.0
2.0 1.0 1.0 1.0 4.0 1.0 1.0
1.0 3.0 2.0 1.0 2.0 2.0 0.0
1.0 3.0 1.0 2.0 4.0 2.0 0.0
1.0 3.0 2.0 2.0 1.0 1.0 1.0
2.0 2.0 2.0 1.0 1.0 1.0 1.0
2.0 1.0 2.0 2.0 4.0 1.0 1.0
2.0 2.0 1.0 2.0 4.0 1.0 1.0
2.0 1.0 1.0 2.0 1.0 2.0 0.0
2.0 4.0 2.0 1.0 2.0 2.0 0.0
1.0 3.0 1.0 1.0 3.0 1.0 1.0
1.0 2.0 1.0 2.0 4.0 1.0 1.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 3.0 2.0 2.0 4.0 2.0 0.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 2.0 2.0 1.0 4.0 2.0 0.0
2.0 3.0 1.0 1.0 1.0 2.0 0.0
1.0 4.0 2.0 2.0 2.0 1.0 0.0
1.0 2.0 1.0 2.0 1.0 1.0 1.0
2.0 1.0 2.0 2.0 4.0 2.0 0.0
2.0 3.0 2.0 2.0 3.0 1.0 1.0
1.0 2.0 2.0 2.0 4.0 1.0 1.0
1.0 4.0 2.0 1.0 4.0 2.0 0.0
2.0 1.0 2.0 1.0 2.0 1.0 1.0
2.0 1.0 1.0 2.0 4.0 2.0 0.0
2.0 1.0 2.0 2.0 1.0 1.0 1.0
2.0 4.0 2.0 1.0 1.0 1.0 1.0

2.0 2.0 2.0 2.0 4.0 1.0 1.0
2.0 4.0 2.0 1.0 4.0 1.0 1.0
2.0 3.0 1.0 1.0 2.0 1.0 1.0
2.0 4.0 1.0 1.0 2.0 1.0 1.0
2.0 2.0 1.0 1.0 1.0 1.0 1.0
1.0 2.0 2.0 2.0 4.0 2.0 0.0
1.0 4.0 1.0 2.0 4.0 1.0 1.0
2.0 2.0 2.0 1.0 3.0 2.0 0.0
1.0 3.0 2.0 2.0 2.0 1.0 1.0
1.0 3.0 1.0 1.0 4.0 1.0 1.0
1.0 3.0 2.0 2.0 3.0 2.0 0.0
2.0 1.0 1.0 2.0 3.0 1.0 1.0
1.0 2.0 2.0 2.0 4.0 1.0 1.0
2.0 1.0 2.0 1.0 1.0 2.0 0.0
1.0 2.0 2.0 2.0 2.0 2.0 0.0
2.0 1.0 1.0 1.0 3.0 2.0 0.0
1.0 3.0 2.0 2.0 4.0 1.0 1.0
1.0 1.0 2.0 2.0 4.0 2.0 0.0
1.0 1.0 2.0 2.0 4.0 1.0 1.0
2.0 2.0 1.0 1.0 4.0 2.0 0.0
1.0 1.0 1.0 1.0 4.0 2.0 0.0
2.0 2.0 1.0 1.0 2.0 1.0 1.0
1.0 4.0 2.0 2.0 1.0 1.0 0.0
1.0 1.0 2.0 1.0 3.0 2.0 0.0
2.0 2.0 1.0 1.0 2.0 2.0 0.0
1.0 2.0 2.0 1.0 4.0 1.0 1.0
2.0 3.0 2.0 2.0 3.0 2.0 0.0
2.0 1.0 1.0 1.0 4.0 2.0 0.0
1.0 1.0 1.0 1.0 2.0 1.0 1.0
1.0 2.0 2.0 2.0 3.0 1.0 1.0
1.0 3.0 1.0 2.0 1.0 2.0 0.0
2.0 2.0 1.0 2.0 3.0 2.0 0.0
2.0 1.0 1.0 1.0 1.0 2.0 0.0
1.0 1.0 1.0 2.0 2.0 1.0 1.0
1.0 2.0 2.0 2.0 2.0 1.0 1.0
2.0 2.0 2.0 2.0 1.0 2.0 0.0
2.0 1.0 1.0 2.0 4.0 1.0 1.0
2.0 1.0 1.0 1.0 4.0 1.0 1.0
1.0 2.0 1.0 1.0 1.0 2.0 0.0
1.0 1.0 2.0 1.0 4.0 2.0 0.0
2.0 3.0 2.0 1.0 2.0 1.0 1.0
1.0 4.0 2.0 2.0 3.0 1.0 0.0
1.0 2.0 2.0 2.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 4.0 1.0 1.0
2.0 4.0 2.0 2.0 4.0 2.0 0.0
1.0 4.0 2.0 1.0 2.0 2.0 0.0
1.0 4.0 1.0 2.0 1.0 1.0 0.0
2.0 3.0 1.0 1.0 3.0 2.0 0.0
2.0 1.0 2.0 1.0 4.0 1.0 1.0

1.0 1.0 1.0 2.0 4.0 1.0 1.0
1.0 3.0 2.0 2.0 1.0 2.0 0.0
1.0 1.0 2.0 1.0 1.0 2.0 0.0
1.0 1.0 2.0 1.0 4.0 1.0 1.0
1.0 2.0 1.0 1.0 3.0 1.0 1.0
1.0 1.0 1.0 2.0 1.0 2.0 0.0

---

net0

1
6 1
1.0
1
0.1
0.1
0.5
1

B.5 Sample Data Files for the Iris Plants Database Problem

norm.data

```
48.0
170.0
50
```

patterns

```
150
7
5.2 3.5 1.5 0.2  1.0 0.0 0.0
4.9 2.4 3.3 1.0  0.0 1.0 0.0
5.6 3.0 4.5 1.5  0.0 1.0 0.0
6.7 3.0 5.2 2.3  0.0 0.0 1.0
5.0 2.0 3.5 1.0  0.0 1.0 0.0
6.4 2.9 4.3 1.3  0.0 1.0 0.0
5.1 2.5 3.0 1.1  0.0 1.0 0.0
5.8 2.7 4.1 1.0  0.0 1.0 0.0
6.5 3.0 5.8 2.2  0.0 0.0 1.0
4.8 3.0 1.4 0.1  1.0 0.0 0.0
7.2 3.2 6.0 1.8  0.0 0.0 1.0
5.7 4.4 1.5 0.4  1.0 0.0 0.0
5.1 3.5 1.4 0.2  1.0 0.0 0.0
6.1 2.6 5.6 1.4  0.0 0.0 1.0
7.7 2.8 6.7 2.0  0.0 0.0 1.0
5.2 4.1 1.5 0.1  1.0 0.0 0.0
5.0 2.3 3.3 1.0  0.0 1.0 0.0
4.6 3.4 1.4 0.3  1.0 0.0 0.0
6.5 3.0 5.2 2.0  0.0 0.0 1.0
5.8 2.6 4.0 1.2  0.0 1.0 0.0
6.1 2.8 4.7 1.2  0.0 1.0 0.0
5.0 3.3 1.4 0.2  1.0 0.0 0.0
6.3 2.7 4.9 1.8  0.0 0.0 1.0
6.4 2.7 5.3 1.9  0.0 0.0 1.0
6.4 2.8 5.6 2.2  0.0 0.0 1.0
6.8 3.0 5.5 2.1  0.0 0.0 1.0
6.3 2.5 5.0 1.9  0.0 0.0 1.0
```



```
4.7 3.2 1.6 0.2  1.0 0.0 0.0
6.5 2.8 4.6 1.5  0.0 1.0 0.0
5.8 4.0 1.2 0.2  1.0 0.0 0.0
7.7 3.0 6.1 2.3  0.0 0.0 1.0
5.5 2.4 3.8 1.1  0.0 1.0 0.0
6.9 3.1 5.4 2.1  0.0 0.0 1.0
5.5 2.4 3.7 1.0  0.0 1.0 0.0
5.0 3.4 1.5 0.2  1.0 0.0 0.0
5.1 3.3 1.7 0.5  1.0 0.0 0.0
4.9 3.1 1.5 0.1  1.0 0.0 0.0
5.3 3.7 1.5 0.2  1.0 0.0 0.0
7.6 3.0 6.6 2.1  0.0 0.0 1.0
5.8 2.7 3.9 1.2  0.0 1.0 0.0
4.5 2.3 1.3 0.3  1.0 0.0 0.0
6.9 3.1 4.9 1.5  0.0 1.0 0.0
4.8 3.4 1.9 0.2  1.0 0.0 0.0
5.1 3.7 1.5 0.4  1.0 0.0 0.0
5.6 2.9 3.6 1.3  0.0 1.0 0.0
6.7 3.3 5.7 2.5  0.0 0.0 1.0
6.1 3.0 4.9 1.8  0.0 0.0 1.0
4.4 3.0 1.3 0.2  1.0 0.0 0.0
5.9 3.2 4.8 1.8  0.0 1.0 0.0
5.7 3.0 4.2 1.2  0.0 1.0 0.0
4.9 3.0 1.4 0.2  1.0 0.0 0.0
4.7 3.2 1.3 0.2  1.0 0.0 0.0
7.7 3.8 6.7 2.2  0.0 0.0 1.0
5.4 3.0 4.5 1.5  0.0 1.0 0.0
6.5 3.0 5.5 1.8  0.0 0.0 1.0
6.3 3.3 6.0 2.5  0.0 0.0 1.0
6.9 3.2 5.7 2.3  0.0 0.0 1.0
6.7 3.1 5.6 2.4  0.0 0.0 1.0
7.1 3.0 5.9 2.1  0.0 0.0 1.0
5.0 3.2 1.2 0.2  1.0 0.0 0.0
5.7 2.8 4.1 1.3  0.0 1.0 0.0
6.1 2.9 4.7 1.4  0.0 1.0 0.0
6.2 2.8 4.8 1.8  0.0 0.0 1.0
7.7 2.6 6.9 2.3  0.0 0.0 1.0
5.7 2.5 5.0 2.0  0.0 0.0 1.0
5.0 3.5 1.6 0.6  1.0 0.0 0.0
5.5 3.5 1.3 0.2  1.0 0.0 0.0
5.5 2.3 4.0 1.3  0.0 1.0 0.0
5.6 2.8 4.9 2.0  0.0 0.0 1.0
5.9 3.0 4.2 1.5  0.0 1.0 0.0
6.2 3.4 5.4 2.3  0.0 0.0 1.0
6.8 2.8 4.8 1.4  0.0 1.0 0.0
5.7 2.8 4.5 1.3  0.0 1.0 0.0
6.7 3.1 4.7 1.5  0.0 1.0 0.0
4.4 2.9 1.4 0.2  1.0 0.0 0.0
4.6 3.2 1.4 0.2  1.0 0.0 0.0
```

4.9 3.1 1.5 0.1 1.0 0.0 0.0
6.7 2.5 5.8 1.8 0.0 0.0 1.0
6.8 3.2 5.9 2.3 0.0 0.0 1.0
5.9 3.0 5.1 1.8 0.0 0.0 1.0
5.2 2.7 3.9 1.4 0.0 1.0 0.0
6.0 2.9 4.5 1.5 0.0 1.0 0.0
5.6 2.7 4.2 1.3 0.0 1.0 0.0
4.6 3.1 1.5 0.2 1.0 0.0 0.0
6.6 2.9 4.6 1.3 0.0 1.0 0.0
4.4 3.2 1.3 0.2 1.0 0.0 0.0
5.7 3.8 1.7 0.3 1.0 0.0 0.0
5.6 3.0 4.1 1.3 0.0 1.0 0.0
7.4 2.8 6.1 1.9 0.0 0.0 1.0
6.3 2.9 5.6 1.8 0.0 0.0 1.0
6.5 3.2 5.1 2.0 0.0 0.0 1.0
6.0 3.0 4.8 1.8 0.0 0.0 1.0
5.4 3.9 1.7 0.4 1.0 0.0 0.0
5.5 2.5 4.0 1.3 0.0 1.0 0.0
6.4 2.8 5.6 2.1 0.0 0.0 1.0
5.8 2.8 5.1 2.4 0.0 0.0 1.0
5.0 3.0 1.6 0.2 1.0 0.0 0.0
5.1 3.4 1.5 0.2 1.0 0.0 0.0
5.4 3.4 1.5 0.4 1.0 0.0 0.0
6.9 3.1 5.1 2.3 0.0 0.0 1.0
5.2 3.4 1.4 0.2 1.0 0.0 0.0
6.3 2.5 4.9 1.5 0.0 1.0 0.0
6.3 2.3 4.4 1.3 0.0 1.0 0.0
5.1 3.5 1.4 0.3 1.0 0.0 0.0
6.0 2.7 5.1 1.6 0.0 1.0 0.0
6.2 2.2 4.5 1.5 0.0 1.0 0.0
5.1 3.8 1.5 0.3 1.0 0.0 0.0
5.4 3.7 1.5 0.2 1.0 0.0 0.0
5.7 2.6 3.5 1.0 0.0 1.0 0.0
6.4 3.2 5.3 2.3 0.0 0.0 1.0
6.0 2.2 4.0 1.0 0.0 1.0 0.0
5.1 3.8 1.9 0.4 1.0 0.0 0.0
7.0 3.2 4.7 1.4 0.0 1.0 0.0
5.6 2.5 3.9 1.1 0.0 1.0 0.0
5.5 2.6 4.4 1.2 0.0 1.0 0.0
5.8 2.7 5.1 1.9 0.0 0.0 1.0
4.8 3.1 1.6 0.2 1.0 0.0 0.0
5.4 3.4 1.7 0.2 1.0 0.0 0.0
7.3 2.9 6.3 1.8 0.0 0.0 1.0
6.4 3.2 4.5 1.5 0.0 1.0 0.0
4.9 2.5 4.5 1.7 0.0 0.0 1.0
6.4 3.1 5.5 1.8 0.0 0.0 1.0
6.7 3.3 5.7 2.1 0.0 0.0 1.0
6.7 3.0 5.0 1.7 0.0 1.0 0.0
7.2 3.0 5.8 1.6 0.0 0.0 1.0

6.2 2.9 4.3 1.3 0.0 1.0 0.0
6.1 2.8 4.0 1.3 0.0 1.0 0.0
6.7 3.1 4.4 1.4 0.0 1.0 0.0
4.3 3.0 1.1 0.1 1.0 0.0 0.0
4.8 3.4 1.6 0.2 1.0 0.0 0.0
6.3 3.4 5.6 2.4 0.0 0.0 1.0
5.1 3.8 1.6 0.2 1.0 0.0 0.0
6.6 3.0 4.4 1.4 0.0 1.0 0.0
7.2 3.6 6.1 2.5 0.0 0.0 1.0
7.9 3.8 6.4 2.0 0.0 0.0 1.0
5.0 3.4 1.6 0.4 1.0 0.0 0.0
6.0 3.4 4.5 1.6 0.0 1.0 0.0
5.5 4.2 1.4 0.2 1.0 0.0 0.0
6.0 2.2 5.0 1.5 0.0 0.0 1.0
5.8 2.7 5.1 1.9 0.0 0.0 1.0
6.1 3.0 4.6 1.4 0.0 1.0 0.0
4.9 3.1 1.5 0.1 1.0 0.0 0.0
5.0 3.6 1.4 0.2 1.0 0.0 0.0
6.3 3.3 4.7 1.6 0.0 1.0 0.0
5.0 3.5 1.3 0.3 1.0 0.0 0.0
5.7 2.9 4.2 1.3 0.0 1.0 0.0
6.3 2.8 5.1 1.5 0.0 0.0 1.0
5.4 3.9 1.3 0.4 1.0 0.0 0.0
4.6 3.6 1.0 0.2 1.0 0.0 0.0
4.8 3.0 1.4 0.3 1.0 0.0 0.0

---

net0

2
4 2 3
3.0
1
0.1
0.1
0.5
1

B.6 Sample Data Files for the Lymphography Domain Problem

norm.data

```
60.0
111.0
100
```

patterns

```
148
20
4.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 2.0 4.0 8.0 1.0 1.0 2.0 2.0  1.0 0.0
3.0 2.0 1.0 1.0 2.0 2.0 1.0 2.0 1.0 3.0 3.0 2.0 3.0 4.0 2.0 2.0 2.0 2.0  0.0 1.0
3.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 1.0 4.0 3.0 3.0 4.0 8.0 3.0 2.0 2.0 7.0  1.0 0.0
3.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 3.0 3.0 4.0 4.0 4.0 3.0 1.0 2.0 6.0  1.0 0.0
3.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 2.0 4.0 3.0 5.0 1.0 2.0 2.0 1.0  0.0 1.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 3.0 3.0 3.0 3.0 6.0 3.0 1.0 2.0 4.0  0.0 1.0
2.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 3.0 2.0 3.0 8.0 2.0 1.0 1.0 1.0  0.0 1.0
3.0 2.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 2.0 2.0 2.0 2.0 1.0 3.0 1.0 1.0 1.0  0.0 1.0
2.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 3.0 2.0 2.0 2.0 8.0 3.0 1.0 2.0 5.0  1.0 0.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 3.0 3.0 5.0 3.0 1.0 1.0 2.0  1.0 0.0
2.0 2.0 2.0 1.0 2.0 2.0 1.0 2.0 1.0 3.0 2.0 4.0 3.0 5.0 1.0 2.0 2.0 3.0  0.0 1.0
4.0 2.0 1.0 1.0 1.0 2.0 1.0 1.0 1.0 3.0 3.0 4.0 3.0 8.0 3.0 2.0 2.0 2.0  0.0 1.0
4.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 2.0 2.0 3.0 2.0 1.0 1.0 1.0  0.0 1.0
4.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 4.0 2.0 2.0 4.0 7.0 3.0 2.0 2.0 2.0  1.0 0.0
3.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 3.0 1.0 1.0 2.0 2.0 8.0 1.0 2.0 2.0 4.0  1.0 1.0
2.0 2.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 2.0 3.0 3.0 3.0 5.0 3.0 2.0 2.0 2.0  0.0 1.0
3.0 2.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 2.0 3.0 3.0 3.0 2.0 2.0 2.0 2.0 3.0  0.0 1.0
2.0 2.0 1.0 1.0 1.0 2.0 1.0 1.0 1.0 2.0 2.0 2.0 3.0 5.0 2.0 1.0 2.0 2.0  0.0 1.0
3.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 2.0 3.0 2.0 1.0 2.0 2.0 1.0  0.0 1.0
2.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 3.0 3.0 3.0 3.0 2.0 2.0 2.0 1.0 1.0  0.0 1.0
3.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 4.0 2.0 4.0 3.0 2.0 2.0 3.0  1.0 0.0
4.0 1.0 1.0 1.0 2.0 2.0 1.0 2.0 1.0 3.0 3.0 3.0 4.0 5.0 3.0 2.0 2.0 4.0  1.0 0.0
4.0 2.0 1.0 1.0 2.0 2.0 1.0 2.0 1.0 4.0 3.0 4.0 3.0 4.0 2.0 2.0 2.0 2.0  0.0 1.0
3.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 2.0 2.0 5.0 1.0 1.0 1.0 1.0  0.0 1.0
3.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 2.0 1.0 2.0 2.0 2.0 8.0 3.0 1.0 2.0 8.0  1.0 0.0
3.0 1.0 1.0 1.0 1.0 2.0 1.0 1.0 1.0 3.0 2.0 3.0 3.0 8.0 3.0 2.0 2.0 2.0  1.0 0.0
4.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 3.0 3.0 4.0 2.0 4.0 3.0 2.0 2.0 4.0  1.0 0.0
```



```
4.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 3.0 2.0 4.0 4.0 8.0 3.0 2.0 2.0 1.0  1.0 0.0
4.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 2.0 3.0 3.0 2.0 2.0 2.0 1.0  0.0 1.0
3.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 2.0 4.0 2.0 8.0 2.0 2.0 2.0 1.0  1.0 0.0
4.0 2.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 4.0 3.0 3.0 3.0 7.0 3.0 2.0 2.0 3.0  1.0 0.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 3.0 2.0 3.0 3.0 8.0 3.0 2.0 2.0 3.0  1.0 0.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 3.0 2.0 2.0 8.0 1.0 2.0 1.0 1.0  0.0 1.0
2.0 2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 2.0 3.0 3.0 1.0 1.0 1.0 1.0  0.0 1.0
4.0 2.0 1.0 1.0 2.0 2.0 1.0 2.0 1.0 3.0 3.0 4.0 3.0 4.0 3.0 2.0 2.0 2.0  0.0 1.0
2.0 2.0 2.0 1.0 2.0 2.0 1.0 2.0 1.0 3.0 3.0 3.0 3.0 8.0 3.0 1.0 2.0 2.0  0.0 1.0
3.0 1.0 1.0 1.0 2.0 2.0 2.0 1.0 3.0 1.0 1.0 4.0 2.0 5.0 3.0 1.0 2.0 4.0  1.0 1.0
3.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 4.0 3.0 4.0 2.0 4.0 3.0 2.0 2.0 6.0  1.0 0.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 3.0 3.0 2.0 3.0 4.0 3.0 2.0 2.0 2.0  1.0 0.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 2.0 1.0 7.0 1.0 2.0 2.0 2.0  1.0 0.0
2.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 2.0 2.0 3.0 3.0 3.0 3.0 1.0 2.0 1.0  1.0 0.0
2.0 2.0 2.0 1.0 2.0 2.0 2.0 2.0 1.0 4.0 2.0 2.0 2.0 4.0 3.0 2.0 2.0 6.0  1.0 0.0
3.0 2.0 2.0 1.0 1.0 2.0 1.0 1.0 1.0 3.0 2.0 3.0 3.0 4.0 3.0 1.0 2.0 2.0  0.0 1.0
4.0 2.0 1.0 1.0 2.0 1.0 1.0 2.0 1.0 3.0 3.0 3.0 3.0 2.0 2.0 2.0 2.0 3.0  0.0 1.0
3.0 2.0 2.0 2.0 2.0 2.0 2.0 1.0 2.0 2.0 2.0 4.0 2.0 4.0 3.0 2.0 2.0 7.0  1.0 1.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 2.0 2.0 8.0 2.0 2.0 2.0 1.0  1.0 0.0
3.0 2.0 2.0 1.0 2.0 1.0 1.0 2.0 1.0 2.0 2.0 3.0 3.0 8.0 3.0 1.0 2.0 1.0  0.0 1.0
4.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 3.0 4.0 2.0 8.0 1.0 2.0 2.0 1.0  0.0 1.0
3.0 2.0 2.0 1.0 1.0 2.0 1.0 1.0 1.0 2.0 3.0 3.0 3.0 5.0 8.0 1.0 2.0 2.0  0.0 1.0
2.0 2.0 2.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 4.0 3.0 8.0 2.0 2.0 2.0 2.0  0.0 1.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 2.0 3.0 1.0 2.0 2.0 1.0  0.0 1.0
3.0 2.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 2.0 2.0 2.0 4.0 8.0 3.0 1.0 2.0 3.0  1.0 0.0
3.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 1.0 2.0 2.0 4.0 3.0 5.0 2.0 1.0 2.0 1.0  0.0 1.0
2.0 2.0 1.0 1.0 2.0 2.0 1.0 1.0 1.0 3.0 3.0 4.0 3.0 4.0 3.0 1.0 1.0 1.0  0.0 1.0
2.0 2.0 2.0 1.0 2.0 2.0 1.0 1.0 1.0 2.0 2.0 3.0 3.0 4.0 2.0 1.0 2.0 1.0  0.0 1.0
4.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 3.0 3.0 3.0 3.0 2.0 1.0 1.0  1.0 0.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 3.0 2.0 4.0 2.0 8.0 3.0 2.0 2.0 4.0  1.0 0.0
4.0 2.0 1.0 1.0 2.0 2.0 1.0 2.0 1.0 2.0 3.0 2.0 3.0 2.0 3.0 2.0 2.0 4.0  0.0 1.0
4.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 1.0 4.0 3.0 4.0 4.0 7.0 3.0 2.0 2.0 8.0  1.0 0.0
2.0 2.0 2.0 1.0 2.0 2.0 1.0 2.0 1.0 3.0 3.0 3.0 3.0 8.0 3.0 2.0 2.0 2.0  0.0 1.0
4.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 4.0 2.0 2.0 4.0 7.0 3.0 2.0 2.0 7.0  1.0 0.0
3.0 2.0 2.0 1.0 2.0 2.0 1.0 2.0 1.0 3.0 2.0 2.0 3.0 4.0 1.0 2.0 2.0 1.0  0.0 1.0
3.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 4.0 2.0 4.0 4.0 2.0 3.0 2.0 2.0 3.0  1.0 0.0
4.0 2.0 1.0 1.0 1.0 2.0 1.0 1.0 1.0 2.0 3.0 2.0 3.0 2.0 2.0 1.0 1.0 1.0  0.0 1.0
2.0 2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 2.0 3.0 3.0 5.0 2.0 2.0 2.0 1.0  0.0 1.0
4.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 1.0 3.0 2.0 3.0 4.0 5.0 3.0 2.0 2.0 7.0  1.0 0.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 4.0 3.0 4.0 4.0 5.0 3.0 2.0 2.0 5.0  1.0 0.0
3.0 2.0 2.0 1.0 2.0 2.0 1.0 1.0 1.0 3.0 3.0 4.0 3.0 4.0 2.0 2.0 2.0 3.0  0.0 1.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 3.0 2.0 3.0 3.0 2.0 2.0 1.0 1.0  1.0 0.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 3.0 2.0 2.0 2.0 6.0 3.0 2.0 2.0 6.0  1.0 0.0
3.0 2.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 3.0 3.0 3.0 3.0 4.0 3.0 2.0 2.0 1.0  0.0 1.0
3.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 2.0 2.0 2.0 5.0 1.0 1.0 2.0 2.0  1.0 0.0
4.0 2.0 1.0 1.0 2.0 2.0 1.0 1.0 1.0 1.0 3.0 3.0 3.0 3.0 3.0 2.0 2.0 3.0  0.0 1.0
3.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 4.0 3.0 3.0 4.0 5.0 3.0 2.0 2.0 3.0  1.0 0.0
2.0 1.0 1.0 1.0 1.0 2.0 1.0 1.0 1.0 2.0 2.0 2.0 2.0 3.0 1.0 1.0 1.0 1.0  1.0 0.0
4.0 2.0 2.0 1.0 2.0 2.0 1.0 2.0 1.0 3.0 3.0 4.0 3.0 8.0 2.0 2.0 2.0 3.0  0.0 1.0
```

4.0 2.0 2.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 2.0 3.0 3.0 2.0 2.0 2.0 2.0 0.0 1.0
3.0 1.0 1.0 1.0 2.0 1.0 1.0 2.0 1.0 2.0 3.0 3.0 3.0 5.0 3.0 1.0 1.0 1.0 0.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 0.0 0.0
3.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 3.0 2.0 2.0 4.0 8.0 3.0 2.0 2.0 3.0 1.0 0.0
2.0 2.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 3.0 3.0 3.0 4.0 8.0 3.0 2.0 2.0 2.0 1.0 0.0
4.0 2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 4.0 3.0 3.0 3.0 4.0 2.0 2.0 1.0 1.0 0.0 1.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 3.0 2.0 2.0 2.0 8.0 3.0 2.0 2.0 5.0 1.0 0.0
2.0 2.0 2.0 1.0 2.0 2.0 2.0 2.0 1.0 3.0 3.0 4.0 4.0 5.0 3.0 2.0 2.0 5.0 1.0 0.0
4.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 4.0 3.0 4.0 2.0 5.0 3.0 2.0 2.0 2.0 1.0 0.0
2.0 2.0 1.0 1.0 2.0 2.0 1.0 1.0 1.0 2.0 2.0 4.0 2.0 8.0 3.0 2.0 2.0 1.0 0.0 1.0
2.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 3.0 3.0 3.0 5.0 2.0 1.0 2.0 1.0 0.0 1.0
2.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 2.0 3.0 8.0 3.0 1.0 2.0 1.0 0.0 1.0
2.0 2.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 3.0 3.0 4.0 2.0 8.0 3.0 2.0 2.0 2.0 1.0 0.0
3.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 3.0 2.0 2.0 2.0 8.0 1.0 1.0 1.0 1.0 0.0 1.0
4.0 2.0 2.0 1.0 2.0 2.0 1.0 2.0 1.0 1.0 2.0 2.0 1.0 3.0 1.0 2.0 2.0 2.0 1.0 0.0
2.0 2.0 2.0 1.0 1.0 2.0 1.0 1.0 1.0 2.0 3.0 2.0 3.0 3.0 2.0 1.0 1.0 1.0 0.0 1.0
3.0 2.0 2.0 1.0 2.0 2.0 1.0 2.0 1.0 2.0 3.0 3.0 4.0 2.0 2.0 2.0 1.0 1.0 0.0 1.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 2.0 3.0 3.0 1.0 2.0 2.0 1.0 0.0 1.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 3.0 2.0 4.0 4.0 4.0 3.0 2.0 2.0 5.0 1.0 0.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 4.0 2.0 8.0 3.0 2.0 2.0 3.0 1.0 0.0
3.0 2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 2.0 3.0 3.0 3.0 3.0 1.0 1.0 2.0 0.0 1.0
2.0 2.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 2.0 3.0 2.0 2.0 4.0 2.0 1.0 1.0 2.0 0.0 1.0
3.0 2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 2.0 3.0 3.0 5.0 1.0 1.0 1.0 1.0 0.0 1.0
3.0 2.0 1.0 1.0 1.0 2.0 1.0 1.0 1.0 2.0 2.0 4.0 3.0 4.0 1.0 2.0 2.0 2.0 0.0 1.0
2.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 3.0 3.0 4.0 2.0 8.0 3.0 2.0 2.0 3.0 1.0 0.0
2.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 3.0 3.0 2.0 3.0 2.0 2.0 2.0 0.0 1.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 2.0 2.0 4.0 1.0 2.0 1:0 2.0 0.0 1.0
3.0 2.0 1.0 1.0 2.0 2.0 1.0 1.0 1.0 2.0 2.0 4.0 3.0 2.0 1.0 2.0 2.0 3.0 0.0 1.0
4.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 3.0 3.0 2.0 2.0 4.0 2.0 2.0 2.0 2.0 0.0 1.0
2.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 4.0 3.0 4.0 2.0 7.0 3.0 2.0 2.0 5.0 1.0 0.0
3.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 3.0 3.0 3.0 8.0 2.0 2.0 2.0 1.0 0.0 1.0
2.0 2.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 4.0 3.0 4.0 4.0 8.0 3.0 2.0 2.0 7.0 1.0 0.0
3.0 2.0 1.0 1.0 1.0 2.0 1.0 1.0 1.0 2.0 3.0 4.0 2.0 4.0 1.0 1.0 1.0 1.0 0.0 1.0
2.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 3.0 3.0 3.0 5.0 3.0 1.0 2.0 1.0 0.0 1.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 3.0 3.0 3.0 8.0 3.0 1.0 1.0 1.0 0.0 1.0
4.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 4.0 3.0 4.0 3.0 3.0 3.0 2.0 2.0 5.0 1.0 0.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 2.0 2.0 2.0 8.0 1.0 1.0 1.0 1.0 0.0 1.0
2.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 2.0 2.0 2.0 2.0 8.0 2.0 1.0 2.0 2.0 1.0 0.0
3.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 4.0 4.0 2.0 3.0 2.0 1.0 1.0 1.0 0.0
3.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 2.0 2.0 4.0 3.0 8.0 2.0 2.0 2.0 1.0 1.0 0.0
4.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 3.0 2.0 2.0 4.0 3.0 2.0 2.0 1.0 1.0 1.0 0.0
4.0 2.0 2.0 1.0 1.0 2.0 1.0 2.0 1.0 2.0 2.0 3.0 3.0 8.0 3.0 2.0 2.0 2.0 0.0 1.0
2.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 3.0 2.0 2.0 2.0 4.0 3.0 1.0 2.0 5.0 1.0 0.0
4.0 2.0 1.0 1.0 2.0 2.0 1.0 2.0 1.0 3.0 3.0 4.0 2.0 5.0 3.0 2.0 2.0 3.0 1.0 0.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 4.0 3.0 8.0 2.0 2.0 2.0 2.0 0.0 1.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 2.0 2.0 3.0 8.0 2.0 1.0 2.0 1.0 0.0 1.0
2.0 2.0 2.0 2.0 2.0 2.0 1.0 1.0 1.0 2.0 2.0 4.0 3.0 8.0 2.0 2.0 2.0 3.0 0.0 1.0
3.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 3.0 3.0 3.0 4.0 3.0 1.0 2.0 2.0 0.0 1.0
4.0 2.0 1.0 1.0 1.0 2.0 1.0 1.0 1.0 2.0 2.0 3.0 3.0 5.0 2.0 1.0 2.0 1.0 0.0 1.0

3.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 4.0 2.0 2.0 2.0 4.0 3.0 2.0 2.0 7.0 1.0 0.0
2.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 3.0 3.0 3.0 3.0 3.0 3.0 2.0 1.0 2.0 1.0 0.0 1.0
2.0 2.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 2.0 2.0 3.0 3.0 4.0 2.0 1.0 2.0 1.0 0.0 1.0
2.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 2.0 2.0 2.0 2.0 4.0 8.0 1.0 2.0 2.0 2.0 1.0 0.0 0.0
3.0 2.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 3.0 2.0 4.0 3.0 4.0 2.0 2.0 2.0 2.0 0.0 1.0
2.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 4.0 3.0 4.0 2.0 8.0 2.0 1.0 2.0 4.0 0.0 1.0
4.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 1.0 2.0 1.0 4.0 3.0 4.0 4.0 7.0 3.0 2.0 2.0 6.0 1.0 0.0
4.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 4.0 2.0 4.0 2.0 8.0 3.0 2.0 2.0 6.0 1.0 0.0
3.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 3.0 3.0 3.0 5.0 2.0 2.0 2.0 1.0 1.0 0.0 1.0
2.0 2.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 2.0 2.0 4.0 4.0 5.0 3.0 2.0 2.0 1.0 0.0 1.0
3.0 2.0 2.0 2.0 2.0 2.0 1.0 2.0 1.0 2.0 3.0 3.0 3.0 4.0 3.0 2.0 2.0 7.0 0.0 1.0
1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 2.0 2.0 1.0 1.0 2.0 1.0 1.0 1.0 2.0 0.0 0.0
2.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 3.0 3.0 2.0 2.0 4.0 3.0 2.0 2.0 6.0 1.0 0.0
2.0 2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 3.0 3.0 3.0 3.0 3.0 1.0 2.0 1.0 0.0 1.0
3.0 1.0 1.0 1.0 2.0 2.0 2.0 1.0 3.0 1.0 1.0 2.0 1.0 5.0 3.0 1.0 1.0 7.0 1.0 1.0
2.0 2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 3.0 3.0 3.0 2.0 2.0 2.0 2.0 1.0 0.0 1.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 4.0 4.0 8.0 2.0 1.0 2.0 2.0 1.0 0.0
2.0 2.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 2.0 3.0 3.0 3.0 5.0 3.0 2.0 2.0 1.0 0.0 1.0
3.0 2.0 1.0 1.0 2.0 2.0 1.0 2.0 1.0 2.0 2.0 4.0 3.0 5.0 2.0 2.0 2.0 4.0 1.0 0.0
2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 3.0 1.0 2.0 2.0 1.0 0.0 1.0
2.0 2.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 3.0 3.0 3.0 3.0 8.0 3.0 2.0 2.0 4.0 1.0 0.0
2.0 1.0 1.0 1.0 1.0 2.0 1.0 2.0 2.0 4.0 2.0 2.0 1.0 2.0 2.0 1.0 1.0 0.0 1.0
2.0 2.0 2.0 1.0 2.0 2.0 1.0 2.0 1.0 3.0 3.0 4.0 3.0 4.0 3.0 2.0 2.0 6.0 0.0 1.0

---

net0

1
18 2
3.0
1
0.1
0.1
0.5
1

REFERENCES

[1] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "A learning algorithm for Boltzmann machines," *Cognitive Science*, vol. 9, no. 1, pp. 147-169, Ablex Pub. Co., Norwood, NJ, 1985.

[2] D. W. Aha, "Machine learning databases," *anonymous@ics.uci.edu*, University of California, Irvine, CA, 1990.

[3] G. Cestnik, I. Konenenko, and I. Bratko, "Assistant-86: A knowledge-elicitation tool for sophisticated users.," *Progress in Machine Learning*, pp. 31-45, Sigma Press, 1987.

[4] P. Cheeseman, J. Kelly, M. Self, J. Stutz, W. Taylor, and D. Freeman, "AutoClass: A Bayesian classification system," *Machine Learning*, pp. 54-64, Kluwer Academic Publishers, Boston, MA, 1988.

[5] P. Clark and T. Niblett, "Induction in noisy domains," *Progress in Machine Learning*, pp. 11-30, Sigma Press, 1987.

[6] B. V. Dasarathy, "Nosing around the neighborhood: A new system structureand classification rule for recognition in partially exposed environments.," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. PAMI-2, No. 1, pp. 67-71, 1980.

[7] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, John Wiley & Sons, 1973.

[8] J. L. Elman and D. Zipser, *Learning the Hidden Structure of Speech*, ICS Report 8701, Institute for Cognitive Science, Univ. of California, San Diego, La Jolla, CA, 1987.

[9] S. E. Fahlman, "NN-benchmarks mailing list," *nn-bench@cs.cmu.edu*, Carnegie Mellon University, Pittsburgh, PA, 1988.

[10] S. E. Fahlman, "Faster-learning variations on back propagation: An empirical study," *Proc. 1988 Connectionist Models Summer School*, pp. 38-51, Morgan Kaufmann, Palo Alto, CA, 1988.

[11] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Contributions to Mathematical Statistics*, John Wiley, NY, 1950.

[12] G. W. Gates, "The reduced nearest neighbor rule," *IEEE Trans. on Information Theory*, pp. 431-433, May 1972.

[13] R. P. Gorman and T. J. Sejnowski, "Learned classification of sonar targets using a massively parallel network," *Proc. Workshop on Neural Network Devices and Applications*, Document JPL D4406, Jet Propulsion Laboratory, Pasadina, CA, Feb. 1987.

[14] K. J. Lang and M. J. Witbrock, "Learning to tell two spirals apart," *Proc. 1988 Connectionist Models Summer School*, pp. 52-59, Morgan Kaufmann, Palo Alto, CA, 1988.

[15] R. Michalski, I. Mozetic, J. Hong, and N. Lavrac, "The multi-purpose incremental learning system AQ15 and its testing applications to three medical domains.," *Proc. of the Fifth National Conference on Artificial Intelligence*, pp. 1041-1045, Morgan Kaufmann, Philadelphia, PA, 1986.

[16] D. Michie, "The fifth generation's unbridged gap," *Rolf Herken (Ed.) The Universal Turing Machine: A Half-Century Survey*, pp. 466-489, Oxford University Press, 1988.

[17] R. Mooney, J.W. Shavlik, G. Towell, and A. Gove, "An experimental comparison of symbolic and connectionist learning algorithms," *Proc. Int'l. Joint Conf. on Artificial Intelligence*, pp. 775-780, Morgan Kaufman, 1989.

[18] J. Pearl, "On the discovery and generation of certain heuristics," *The AI Magazine*, pp. 23-33, AAAI, Winter/Spring 1983.

[19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533-536, Oct. 1986.

[20] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," pp. 318-362 in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, ed. D. E. Rumelhart, J. L McClelland and the PDP Research Group, MIT Press, Cambridge, MA, 1986.

[21] T. J. Sejnowski and C. R. Rosenberg, "Parallel networks that learn to pronounce English text," *Complex Systems*, vol. 1, pp. 145-168, Complex Systems Publications, Champaign, IL, 1987.

[22] H. A. Simon and J. B. Kadane, "Optimal problem-solving search: All-or-none solutions," *Artificial Intelligence*, vol. 6, no. 3, pp. 235-247, North-Holland, 1975.

[23] B. W. Wah and H. Kriplani, "Resource constrained design of artificial neural networks," *Proc. Int'l. Joint Conf. on Neural Networks*, IEEE, San Diego, CA, June 1990.

[24] B. Widrow and S. D. Stearns, *Adaptive Signal Processing*, Prentice-Hall, 1985.

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

GRADUATE COLLEGE DEPARTMENTAL FORMAT APPROVAL

THIS IS TO CERTIFY THAT THE FORMAT AND QUALITY OF PRESENTATION OF THE THESIS SUBMITTED BY HARISH KRIPLANI AS ONE OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE ARE ACCEPTABLE TO THE DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING.

Full Name of Department, Division or Unit

19 April 1990

Date of Approval

Departmental Representative