RESOURCE CONSTRAINED DESIGN OF ARTIFICIAL NEURAL NETWORKS

Benjamin W. Wah and Harish Kriplani

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 West Springfield Avenue
Urbana, IL 61801
wah%aquinas@uxc.cso.uiuc.edu

ABSTRACT

In this paper, we address the problem of automating the design of artificial neural networks based on a gradient descent learning algorithm for solving a given application. There are many possible network configurations for solving the given application, and enumerating and training all of them is impossible in any reasonable amount of time. We present a heuristic design method for selecting and training promising neural networks, with the goal of maximizing a given objective function of cost and training time and satisfying given resource constraints. Our proposed method consists of two parts. For a given amount of time divided into quanta, the first part of our method 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 our method refines the error measure used in the first part. When the time limit is exceeded, a set of promising but possibly partially trained networks are reported. We have applied the method to a variety of benchmark problems and found it to be effective in identifying promising network configurations.

KEYWORDS AND PHRASES. Automated design, back error propagation, goodness, guidance, learning experiments, search.

1. INTRODUCTION

Back-error-propagation (BEP), or simply back-prop, is a powerful gradient descent training algorithm for artificial neural networks [6]. The algorithm has been successfully applied to a variety of problems [1,4,8]. The algorithm minimizes the total sum of squared errors (or TSSE) between the actual outputs and the desired outputs of the network by repeatedly adjusting the weights $w(t)$ by an amount proportional to the gradient of the error-weight surface. The TSSE is accumulated at all the output units over all the input-output training patterns. More precisely,

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

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

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

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. Output o_{pi} of a unit is computed by summing the products of all incoming inputs (y_{pj}) and their associated weights (w_{ij}), adding a bias (θ_i) to it and applying a sigmoidal nonlinear function (F) on the result.

Research supported partly by National Aeronautics Administration Contract NCC 2-481 and National Science Foundation Grant MIP 88-10584.

International Joint Conference on Neural Networks, IEEE, 1990.

$$o_{pi} = F\left[\sum_j w_{ij} \times y_{pj} + \theta_i\right] \quad \text{where } F(x) = \frac{1}{1+e^{-x}}. \tag{3}$$

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

$$\Delta w(t) = -\eta \frac{\partial TSSE(t)}{\partial w(t)} + \alpha \Delta w(t-1), \tag{4}$$

where α is a *momentum constant*. Details of this algorithm, including derivation of some of the important equations, can be found in a chapter by Rumelhart, Hinton, and Williams in [7].

Back-propagation is a heuristic training algorithm (Eq's (3) and (4) 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 paper, 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 to it a number of initial parameters. 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 chosen. 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.

BEP 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 [9]. The algorithm may get stuck in local minima and does not indicate when a local minimum is reached or how to get out of the local minimum. The training times obtained by applying it 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 paper. 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, in using the BEP algorithm, we need to specify a configuration and the learning parameter values.

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 the 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 we propose incorporates 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. The networks considered are *layered feedforward networks*, also known as *back-prop networks*, after the BEP algorithm. These networks have one layer of input units (or neurons), one layer of output units and zero, one or more layers of hidden units. The bottom layer is the input layer, and the topmost layer is the output layer. It is further assumed that every unit receives inputs from lower layers and sends the output to higher layers. Within the same layer, units are not connected.

In this paper, we assume that the objective function of the design is provided by the users and is a linear function of the parameters that are readily measured during the training process. This assumption is justified because the effectiveness of the method can only be measured when its objective is known. The objective function used is

$$OBJ = A - B \times TT - C \times Cost, \tag{5}$$

where A, B and C are problem-dependent constants specified by the users, and TT is the training time. We assume that the stopping criterion is the same for all the networks that converge; that is, TSSE is less than *ecrit*, and TT is less than a limit. For networks that do not converge, they cannot achieve the stopping criterion when the time limit is exceeded. For consistency, we assume that these networks actually converge after an infinite amount of training time is spent. Hence, the objective values for networks that do not converge are very large negative numbers.

Currently, we have considered only linear objective functions of the above form. With a linear objective function, our 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 output spaces of the problem, namely, the number of inputs and outputs and the format of the inputs. We are only studying 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.

There are two methods used in practice for training neural networks. One is the *ad hoc* method, as proposed by Rumelhart, Hinton, and Williams [7]. 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 stopped immediately, and a previously tested configuration or a new configuration should be trained next.

Another method for training neural networks is the *Restart Method* proposed by Scott Fahlman [2]. 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 a configuration does not get trained within T_{max}, then it is abandoned, and another configuration is selected for training. Obviously, the success of this 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 values for the learning parameters.

Both methods described above do not apply in situations with limited resources. In Section 2, we describe a resource-constrained design method which always proposes one or more promising network configurations at its termination. Our method also guides the selection of appropriate networks for solving the problem and the choice of initial weights and the values for the learning parameters. The proposed method uses learning-by-experimentation and aims at discovering heuristic configurations based on modeling the design process as a search [5]. The method is divided into two sets of experiments: Type I and Type II. These experiments are discussed in details in Sections 3 and 4. In Section 5, experimental results on testing the method on a variety of benchmarks selected from the list maintained at Carnegie-Mellon University by Scott Fahlman are shown [3]. Conclusions and future research directions are drawn in Section 6.

2. DESIGN METHOD

The design method starts with an objective function for comparing networks supplied by the user. Given this objective function, there are infinitely many networks with infinitely many initial weights and values of the learning parameters for solving the problem. Not all of these configurations would lead to networks that *maximize* the objective function. Since there is no systematic method to guide the selection of promising configurations and 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 we need to schedule the available resources efficiently to arrive at better configurations for solving the problem.

There are two approaches to allocating resources. We 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, we can divide up our 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 our design method, we divide up the available time resource into equal parts, each of which is known as a *time quantum*.

The block diagram of the design method is shown in Figure 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.

Type I experiments try to find a set of good configurations, given the problem specifications, 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. The total computational time is divided into quanta. In each quantum, limited experiments are performed on a chosen configuration. Evaluation metrics are computed at the end of the quantum based upon the

experimental results obtained for all the quanta of time spent on this configuration. Depending upon the results of this evaluation, the search continues with either the configuration with the best result or a new configuration.

We use two functions in the evaluation metrics E(I): *goodness function*, g^I, and *guidance function*, G^I.

The goodness function gives the order of *goodness* for various configurations, if the experiments were terminated immediately. It tries to select the best configuration or a set of best configurations 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 Section 3.

Since the evaluation criterion supplied to Type I experiments may be wrong, E(I) 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. These exhaustive training experiments are carried out under specified resource constraints separately.

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. Type II learning process utilizes feedback information on whether the E(I) proposed last has improved the quality of the configurations generated, and modifies its strategy based on E(II) for generating E(I) accordingly.

Type II experiments can be implemented in a way similar to Type I experiments (see Figure 2): time is divided into quanta, and in each quantum, limited experiments are performed on alternative E(I) 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 so 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 reexamining the Type II learning process, we find that it is not necessary to carry out higher-order experiments. We do not need to divide up time into intervals and perform overhead-intensive experiments on various E(I) 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. 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 need not be duplicated in Type II experiments. As a result, we do not need E(II) to guide the scheduling of resources in Type II experiments in a fashion similar to that in Type I experiments.

The way two configurations are compared in Type II experiments is shown in Figure 3. We assume that complete training of two configurations, C_1 and C_2, has been performed and C_1 has a higher objective 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 over all possible pairs of quanta in the training process 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 sections, we discuss the two types of experiments in detail.

3. TYPE I EXPERIMENTS

The main components of Type I experiments are identified in Figure 4. These components are discussed in detail below.

Heuristic Configuration Generator (HCG). This 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; and (d) generate an alternative initial weight configuration.

A Figure-of-Merit (FOM) can be used to monitor the performance of 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. 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 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. 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 our current implementation, user specifies a pool of possible candidate networks and the values of the learning parameters to solve the problem.

The HCG is invoked to generate a new network whenever one of the following two conditions is true: (a) whenever a trained network exits Type I experiments, and (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 networks generated by the HCG is an important parameter in the space-time tradeoff of scheduling the available time for Type I experiments. If too many networks are generated, then the amount of time spent on each may not be enough to obtain stable and meaningful performance results. If too few networks are generated, then insufficient number of networks are tested, and the resulting configurations reported by Type I experiments may not represent the best choices. Another consideration on the number of networks 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* in which networks are constantly added to or removed from. The number of networks present at any time in the window is called the *window size*. The window size is set dynamically so that a sufficient number of networks are either experimented or pruned.

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* 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.

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 value is the same as its final objective value; otherwise, it represents an estimate of the objective value based upon the training information available so far. If a configuration has not been trained completely, it is necessary to estimate its training time at completion and its TSSE at that point.

In our experiments, we set a ceiling (TT_{norm}) on the maximum training time that the Scheduler would schedule for a given configuration. This time limit is a practical representation of infinite time in our experiments. 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 TSSE at TT_{norm}, which is equivalent to predicting the convergence point in the learning curve.[1] We describe in the next section 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 minimum training time of a configuration or its TSSE at TT_{norm} include the time spent in training the configuration thus far, its current TSSE value, its current slope and higher-order derivatives of the learning curve, and 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

[1] The learning curve for a configuration is a plot of the TSSE as a function of training time. It shows how error in the configuration varies as training proceeds.

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 paper, we heuristically express cost by the following expression.

$$Cost = Number\ of\ units + 0.1 \times \sqrt{Number\ of\ weights} \tag{6}$$

The *Guidance Function* selects the best configuration for experimentation, if Type I experiments were to be continued, with the goal that the remaining computational resources should not be spent on unpromising networks or be dominated by very promising ones. It heuristically measures the weighted performance of a configuration as a function of its goodness value, resource constraints specified, amount of resources spent on this configuration, and amount of resources remaining. By assigning a penalty to the goodness value of a configuration based on the amount of time spent on it, it ensures that a configuration which is very promising in the beginning does not dominate all the resources. The reason for this is two folds. First, the guidance and goodness functions are heuristic in nature and may be fallible. Consequently, a configuration that looks very promising in the beginning may get stuck in a local minimum at 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 *Pruning Criterion* uses the upper-bound and the memory-constraint functions to prune unpromising configurations in the experiments.

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 values. Initially, when no existing solution for the problem is known, the UBF values for all the configurations are set to TT_{norm}. As configurations are trained to completion, the value of UBF for a configuration is chosen so that the objective value of this configuration is no worse than the objective value of the best configuration obtained so far. Using the objective function defined in Eq. (5), the following equality must be satisfied.

$$A - B \times TT_{best} - C \times Cost_{best} = A - B \times UBF_{config} - C \times Cost_{config}, \tag{7}$$

where TT_{best} and $Cost_{best}$ are, respectively, the training time and cost of the best configuration known, and UBF_{config} and $Cost_{config}$ are, respectively, the upper bound and cost of the configuration under consideration. Rewriting Eq. (7) yields

$$UBF_{config} = TT_{best} + \frac{C \times (Cost_{best} - Cost_{config})}{B} \tag{8}$$

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 critical limit. This effectively improves the processing efficiency of the processors because it eliminates the extensive swapping of pages when memory usage is large.

4. TYPE II EXPERIMENTS

As pointed out earlier, the evaluation criteria used in Type I experiments, which consist of the goodness and guidance functions, could be wrong and must be refined in subsequent experiments. This is done in Type II learning experiments. The goal of Type II experiments is to produce goodness and guidance functions that are capable of identifying configurations with better objective values in Type I experiments. Since the goodness values are used to compare configurations at the end of Type I experiments, a good goodness function is one which can arrange configurations in the order of their final objective values, whenever the experiments are stopped. If it is not possible to achieve such a goodness function, we would like to have 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 during Type I experiments. A few possible places where such an inconsistency might occur are shown in Figure 5.

Before the goodness function, and hence the guidance function, can be learned, it is necessary to normalize the parameters of the goodness function so that the results obtained from one set of experiments are applicable across different experiments. In this paper, we measure the goodness of a configuration by three parameters, viz, TSSE, cost and the training time. Other parameters, such as the slope and the higher-order derivatives of the learning curve, depend upon these three parameters. Therefore, we need to normalize these three parameters only.

TSSE. Due to the choice of the initial weights and the sigmoidal non-linearity(eq. 3), we could show 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.

$$TSSE_{av} = No\ of\ patterns \times No\ of\ output\ units \times 0.25 \tag{9}$$

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

$$TSSE_{norm} = 1.5 \times TSSE_{av} \tag{10}$$

In case that 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}.

Training Time. As with cost, TT_{norm} should also be a multiple of the average training time of the networks being considered. Again, we start with some initial guess for TT_{norm}. As more experiments are conducted, the system refines the value of this parameter.

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

$$\begin{aligned}
\underset{Function}{Goodness} &= Objective\ Function \\
&= A - B \times TT - C \times Cost \\
&= A - B \times (TT - t + t) - C \times Cost, \quad where\ \ t \le TT \\
&= [A - B \times (TT - t)] - [B \times t + C \times Cost] \\
&= \left[A - B \times TT_{norm} \times \frac{(TT - t)}{TT_{norm}} \right] - \left[B \times TT_{norm} \times \frac{t}{TT_{norm}} + C \times Cost \right]
\end{aligned} \tag{11}$$

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

$$Goodness\ Function = \left[A' - \frac{(TT - t)}{TT_{norm}} \right] - \left[\frac{t}{TT_{norm}} + C' \times Cost \right] \tag{12}$$

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,

$$Expected\ Goodness\ Function = g^{EG} = \left[A' - \frac{(TT - t)}{TT_{norm}} \right] \tag{13}$$

$$Penalty\ Function = -\left[\frac{t}{TT_{norm}} + C' \times Cost \right] \quad where\ \ C' = \frac{C}{B \times TT_{norm}} \tag{14}$$

The *penalty function* is the cost of training the network up to time t. It can be readily computed given the amount of time spent in training the configuration and the complexity of the configuration.

The *expected goodness function* is a measure of the proximity of the present configuration from the final 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 configurations. We further note that g^{EG} is a monotonically increasing function of training time already spent 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 (13) has to be approximated heuristically. One such possible heuristic for measuring g^{EG} is as follows.

$$\hat{g}^{EG} = \left(1.0 - \frac{TSSE}{TSSE_{norm}} \right) \tag{15}$$

This heuristic satisfies all the necessary properties of g^{EG}. It can be improved by including other information, such as the slope of the learning curve and its higher-order derivatives. This is done in Type II experiments, which starts with $\hat{g}^{EG}$ defined in Eq. (15) and refines it further.

An essential step in Type II experiments is to be able to compare alternative goodness functions and 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 that

$$\hat{g}_1^{EG} \geq \hat{g}_2^{EG}, \tag{16}$$

since $config_1$ is closer (as far as training goes) to the final state as compared to $config_2$. Thus, if the proposed heuristic in Eq. (15) does not satisfy Eq. (16), 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 proposed by Type I experiments at all intermediate stages of training. More precisely, we would like to minimize the following *Inconsistency Sum* (IS), where

$$IS = X \times SMI + Y \times SIC + Z \times SID \tag{17}$$

where X, Y and Z are some constants. SMI is the sum of monotonic inconsistencies. Wherever the proposed function violates the monotonicity constraint, we add the difference to SMI for the training traces we obtained for all the configurations in the intermediate step between Type I and Type II experiments (see Figure 1). SIC is the sum of inconsistency counts. Wherever Eq. (16) is violated, SIC count is incremented. This is done for all pairwise combinations of configurations. SID is the sum of inconsistency differences. It is similar to SIC, except that we sum the difference of the two proposed function values.

We have developed an expert system in LISP to generate various heuristic functions for the expected goodness function. This system takes an initial guess for the heuristic function and tries out various possible functions by applying a set of pattern operations. The set of pattern operations implemented include increasing or decreasing the value of one of the coefficients, changing the sign of one of the terms, and adding a new term to the function. The system is supplied with a library of possible candidate terms of the heuristic function. Sequencing these pattern operations is performance driven: based upon the inconsistency count of a function and the effect of applying the above operations to it, one of the pattern operations is selected. If it leads to a function which has better performance, the above operation is repeated; otherwise, some other operation is selected. The performance of this system would be presented in the next section.

The guidance function is a function of the goodness value for the configuration, resource constraints specified, 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 one at an early stage in Type I experiments. Thus, a new guidance function learned in Type II experiments can result in 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.

5. EXPERIMENTAL RESULTS

In this section, we report the results of applying the proposed method to some of the benchmarks collected and maintained at Carnegie Mellon University [3]. Most of the experiments in this research were performed on a Sun 3/60 workstation and a Stardent 4-processor computer. We have experimented extensively with the encode-decode problem with 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 the parameters 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, so it is suitable for demonstrating that our method is applicable to large problems as well.

In Figure 6, we show how Type I experiments proceed as a function of time with the 4-bit encode-decode problem. Our 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 4^{th} 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 9^{th} quantum. At the end of the 9^{th} quantum, $config_3$ exits the window, and another

[5] J. Pearl, "On the Discovery and Generation of Certain Heuristics," *The AI Magazine*, pp. 23-33, AAAI, Winter/Spring 1983.

[6] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations By Back-Propagating Errors," *Nature*, vol. 323, pp. 533-536, Oct. 1986.

[7] 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.

[8] 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.

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

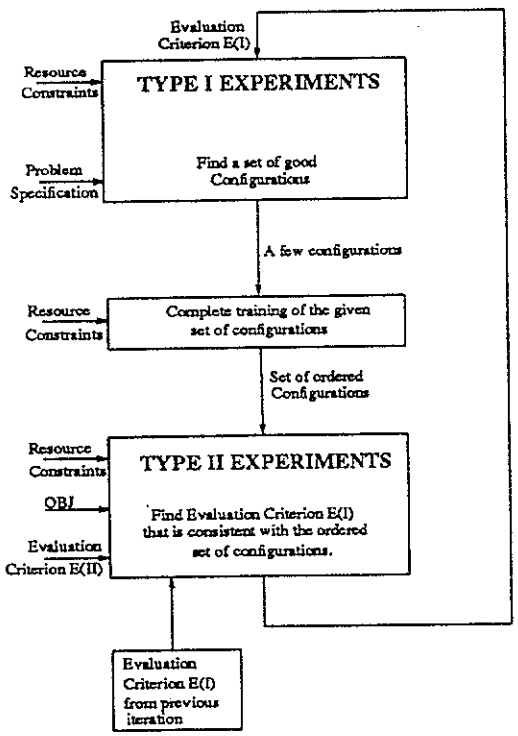

Figure 1. Block diagram of the design method.

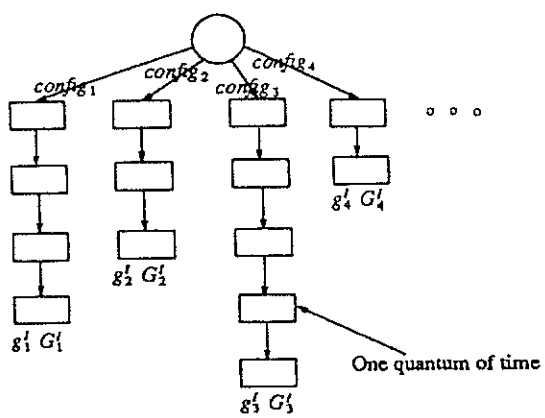

Figure 2. Organization of Type I experiments.

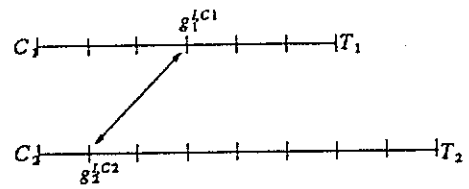

Figure 3. A step in evaluating two configurations in Type II.

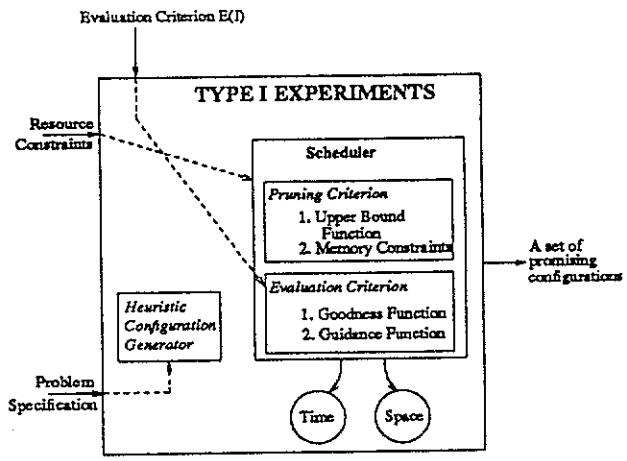

Figure 4. Main components of Type I experiments.

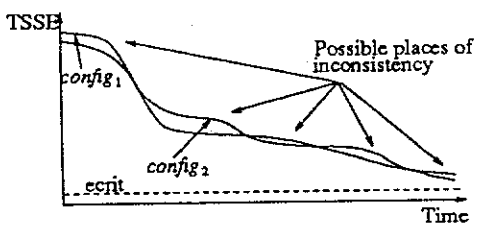

Figure 5. Possible places of inconsistency in two learning curves.

configuration, $config_4$, is included in the active list of configurations. This configuration is trained for one quantum (10^{th}). Again, at the start of the 11^{th} 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 time limit is reached.

All the configurations in the experiment shown in Figure 6 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 21^{st} time quanta, $config_5$ is purged from the window, since 6 time quanta have been spent on it and its TSSE still does not satisfy the stopping criterion.

In Figure 7, we have plotted the average objective values (averaged over 10 experiments) for 4-bit encode-decode problem against various durations of Type I experiments. This curve shows that, on the average, the resulting objective value improves as Type I experiments are allowed to run longer, and that beyond a critical point, allowing Type I experiments to run longer does not improve the objective values significantly. Figure 8 depicts similar behavior for the 8-bit encode-decode problems and illustrates 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 re-learned for solving the 8-bit encode-decode problems. Figure 9 shows a similar plot for the two-spiral benchmark.

In Figure 10, we have plotted a typical trace of Type II experiments for the 4-bit encode-decode problem. A set of configurations were selected and were trained either to completion or until the time spent on them exceeded a fixed ceiling. These traces were fed to Type II experiments, which generated various heuristic functions for the expected goodness function. Figure 10 shows a plot of the inconsistency measure for each of the functions generated versus the step number at which the particular function was generated. Each Δ in the figure marks places where a new term was added to the heuristic function. The addition of a new term required the adjustment in values of all other coefficients, which explains why the performance of the new function worsened at times. After each step, the current function was compared with 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 output as the best possible heuristic found in the time and resources available. This best function output is marked by $\square$ in the figure.

6. CONCLUSIONS

In this paper, 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 and two-spiral 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 rules that we use 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.

REFERENCES

[1] 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.

[2] 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.

[3] S. E. Fahlman, "NN-Benchmarks Mailing List," *nn-bench@cs.cmu.edu*, Carnegie Mellon University, Pittsburgh, PA, 1988.

[4] 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.

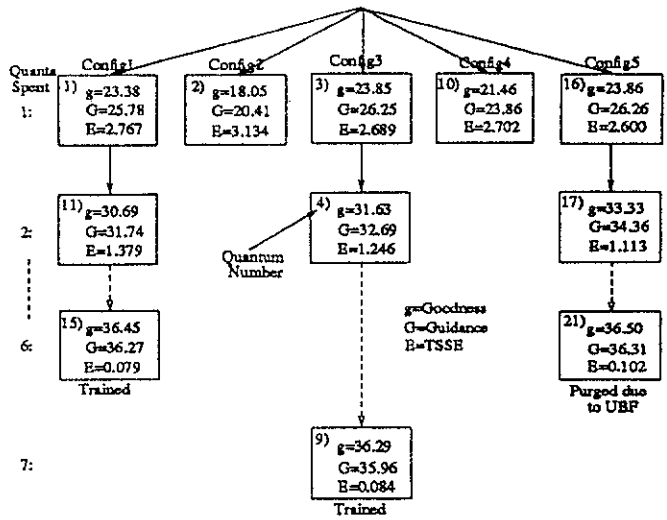

Figure 6. Trace of Type I experiments for the 4-bit encode-decode problem.

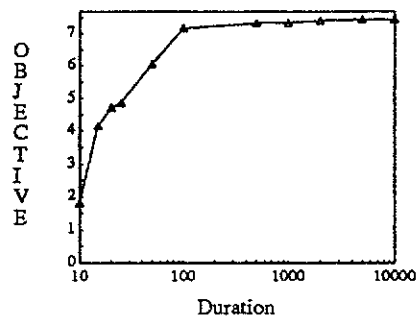

Figure 7. Average objective values versus durations in TYPE I experiments on 4-bit encode-decode problems. (1 step = 10 epochs)

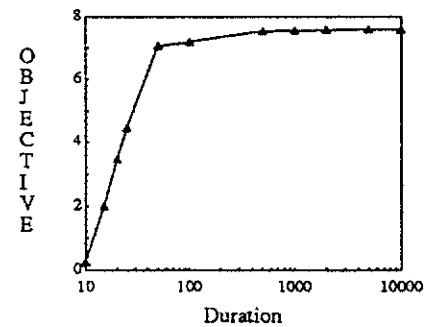

Figure 8. Average objective values versus durations in TYPE I experiments on 8-bit encode-decode problems. (1 step = 10 epochs)

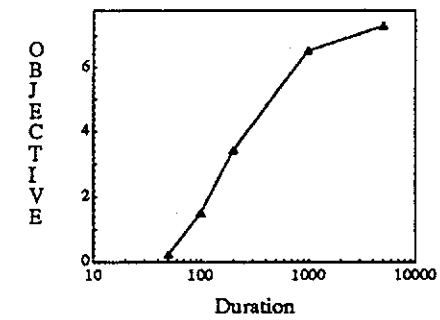

Figure 9. Average objective values versus durations in TYPE I experiments on two-spiral problems. (1 step = 100 epochs)

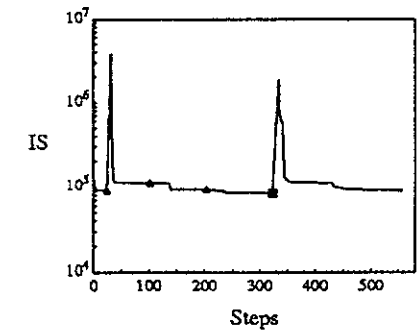

Figure 10. Inconsistency count versus steps in Type II experiments. (1 step = = 55 sec.)