COMPARATOR NEURAL NETWORK FOR DYNAMIC PREDICTION[†]

Benjamin W. Wah, Pankaj Mehra and Chin-Chi Teng[††]

ABSTRACT

In this paper we present the design of a comparator neural network (CANN) and its associated learning algorithms. This network is useful for comparing two time series in order to predict which one will have a better future value based on a user-specified objective function. We illustrate its application in distributed load balancing and in an automated design system for neural networks. In the first application, we use a CANN as a workload predictor that takes as input a time window of measurable values of CPU occupancy, network traffic, disk traffic, and memory occupancy from two computers, and predicts which one of the two computers will have a shorter turnaround time for executing an incoming job. In the second application, we use a CANN to predict which one of two partially trained neural networks will have a shorter convergence time, given current TSSE traces of these networks. In each application, experimental results show that prediction is improved using the proposed CANNs.

1. INTRODUCTION

A large number of applications are concerned with the prediction of future behavior, given behavior observed in the past. This prediction is difficult when we do not have a model of the observed data and when the behavior is non-stationary with time.

In this paper, we present a prediction method based on a comparator neural network (CANN). Instead of predicting the future behavior of a time series, we study a restricted problem that predicts which one of two time series will achieve a smaller objective value in the future. This comparison is useful in practice because in many applications, one is interested to find the best alternative among a set of alternatives. Moreover, performing relative comparison is easier than finding the actual objective value. We discuss in Section 2 our CANN architecture and its learning algorithms.

We show in detail two applications of CANNs in Sections 3 and 4. The first application is for predicting workload in load balancing in order to determine where an incoming job should be executed. This prediction problem can be reduced to comparing the workload between two sites and to identify which site will have a smaller workload in the future. The workload at each site, however, is a complex function of many time-varying and interrelated measurables. Further, each time series has transients that should not be used in the prediction. As a result, we propose to filter each time series and extrapolate the trend by a number of alternative methods. The function of the CANN is, therefore, reduced to selecting the best smoothing and extrapolation methods.

In Section 4, we describe a design system for designing artificial neural networks (ANNs). One of the problems there is to predict whether a particular configuration will converge or not, and whether learning should be switched to a new configuration. We propose a design system that maintains a pool of alternative ANNs and uses a CANN to dynamically identify promising ANNs to train.

[†] Research supported by National Science Foundation grant MIP 92-18715 and National Aeronautics and Space Administration grant NAG 1-613.

[††] B. W. Wah (b-wah@uiuc.edu) and C. C. Teng are with the Coordinated Science Laboratory, University of Illinois, Urbana-Champaign, 1308 West Main Street, Urbana, IL 61801, USA. P. Mehra is with the Informations Sciences Division, NASA Ames Research Center, Moffett Field, CA 94035, USA.

Proceedings of the International Symposium on Artificial Neural Networks, Tainan, Taiwan, December 1994. Preliminary versions of this paper have appeared in references [7, 8].

2. COMPARATOR NETWORK: ARCHITECTURE AND LEARNING ALGORITHMS

Little is known about the problem of learning to compare time series of data. One exception is the work of Tesauro [9], who invented the comparator neural-network architecture for learning to compare alternative moves for the game of backgammon. His approach does not directly carry over to our problem of comparing functions of time series. Our approach, described in this section, was motivated by Tesauro's work; however, in adopting his work to time series prediction, we have made significant departures from both his network configurations and training algorithms.

Figure 1 shows a schematic of our CANN, the details of the training algorithm, and the flow of information during a typical learning trial. Each learning trial involves one training pattern from the training set: first, time-series data from a pair of randomly chosen sources are fed in the left and right inputs of the CANN; then, the *desired outputs* of the two time series are computed based on the *actual outputs* of the time series as follows.

$$O_L^D = O_L - 2\eta(O_L - O_R - O^D), \qquad O_R^D = O_R - 2\eta(O_L - O_R - O^D) \tag{2.1}$$

Finally, the weights of the CANN are modified so that future presentations of similar inputs will generate outputs closer to their respective desired outputs.

Raw time-series patterns enter from the left and right of Figure 1; the delay elements create a window of recent values; the traces from each window are smoothed using low-pass filtering, and an estimate of future performance is determined by extrapolating the smoothed trace. The cutoff frequencies of the filters as well as the extrapolation method depend on the application. For instance, in workload prediction, we use five different filters, with cutoff frequencies at, respectively, 1%, 5%, 10%, 25%, and 50% of maximum frequency. This yields five filtered traces for each input trace. Each of the filtered traces is projected a fixed interval into the future using two different extrapolation techniques (linear fitting and exponential fitting). In load balancing, we are interested in the area below the projected curve, as it represents the fraction of resource that is not available for the incoming job. In the automated design of ANNs, we are interested to predict which one of two networks will converge faster. Consequently, an input trace in the second application is the TSSE (total sum-of-squared errors) trace of a partially trained network, and extrapolation predicts when this trace will be below the prescribed error threshold.

We have implemented CANNs using feed-forward neural networks. Each network comprises three layers of units: an input layer, a hidden layer, and an output layer. The output layer has only one output unit. Links between units are uni-directional, and can connect either a unit in the input layer to a hidden unit or the output unit, or a unit in the hidden layer to the output unit. Associated with each link is a *weight* of that link. The weight on a link going from the i'th unit to the j'th unit of the network is denoted $w_{j,i}$. Every unit in the input layer is connected to every unit of the hidden and output layers, and every unit in the hidden layer is connected to every unit in the output layer. Given the actual and desired outputs for a feed-forward neural network, we used the back-propagation algorithm to determine the appropriate modifications to the weights of that network [5].

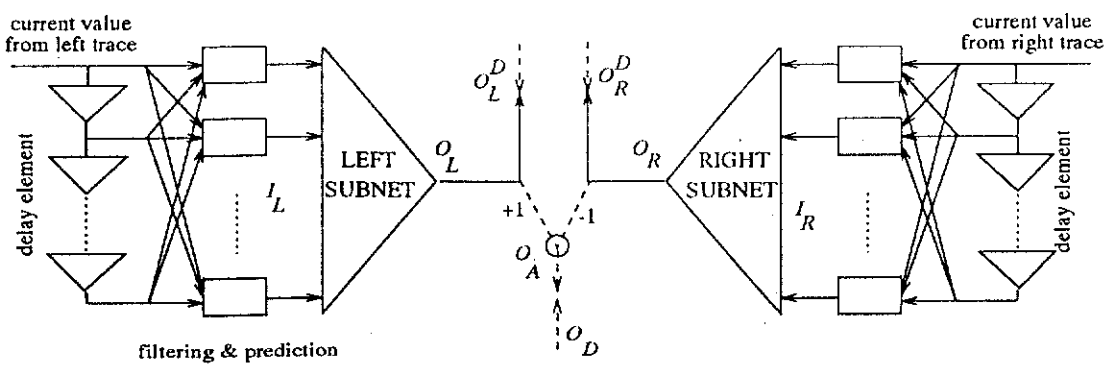


Figure 1 An episode in training a CANN.

The outputs of input-layer units are set at the beginning of each learning trial using filtered and extrapolated time-series values, which are in turn derived from information contained in the training pattern chosen for that trial. The outputs of hidden-layer and output-layer units depend upon their *net inputs*. The net input of unit i (in the hidden or output layer) is given by $\sum_j w_{j,i} o_j$, where o_j denotes the output of unit j. The output of each unit of a feed-forward network is given by the *sigmoidal function* of its net input:

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{2.2}$$

Let us denote by O_L the output of the left subnet; and, correspondingly, O_R, of the right subnet. In order to use the back-propagation learning procedure for training the subnets, we need to determine their desired outputs for every input. Let O_L^D denote the desired output of the left subnet; and, correspondingly, O_R^D, of the right subnet. Further, let us denote by O^A the actual output of the CANN; that is,

$$O^A = O_L - O_R, \qquad \text{and} \qquad O^D = O_L^D - O_R^D. \tag{2.3}$$

The error in the prediction will be reduced if both O^A and O^D have the same sign. That can be achieved by driving their values closer together. Let us denote by E_{LMS} the sum (over all raw training patterns in the training set) of squared errors between the actual and the desired outputs of the CANN. That is,

$$E_{LMS} = \sum_{\tau} (O^A - O^D)^2. \tag{2.5}$$

We can minimize E_{LMS} by performing gradient descent. This can be done by adjusting the outputs of the left and right subnets along their respective partial derivatives of error:

$$\Delta O_L = -\eta \frac{\partial E_{LMS}}{\partial O_L}, \qquad \Delta O_R = -\eta \frac{\partial E_{LMS}}{\partial O_R}, \tag{2.6}$$

where η is known as the *learning rate*. Hence, our training algorithm sets the desired outputs for the left and right subnets as shown in Eq. (2.1).

Finally, to ensure the anti-symmetry of comparison, we can bias the order of presentation of training patterns to the CANN. Raw training patterns are presented in pairs, one after another. If the first training pattern in the pair is τ_1, then the second training pattern must be τ_2. Thus, whenever index-functions are forced to predict that one time series has a larger predicted value than another, they must (in the very next learning trial) predict that the latter will be *smaller* than the former.

3. RELATIVE WORKLOAD PREDICTION IN LOAD BALANCING

In this section, we present the application of the CANN described in Section 2 for predicting workload in a distributed computing system. The goal in load balancing is to identify a site with the minimum load for executing an incoming job. Alternative destinations are ranked by their *load* values, which are usually computed by a manually-specified formula as functions of current and recent utilization levels of various resources. However, good load indices are difficult to design as they are sensitive to installation-specific characteristics of hardware devices as well as to the prevalent load patterns. This section first reviews current approaches to load-index design, discusses their deficiencies, and presents our approach based on prediction using a CANN.

Many existing methods for computing load indices use simplified queuing models of computer systems [3, 4, 11]. Almost all implemented systems use a function known as *UNIX-style load average* (hereafter, load average), which is an exponentially smoothed average of the total number of processes competing for CPU. Load average meaningfully compares loading situations across configurationally identical sites, but fails when the distributed system is heterogeneous.

Typical workstation operating systems provide a number of different performance metrics. (For example, over 20 performance metrics are available in the SunOS kernel.) In the past, Zhou [11] has considered resource-queue lengths (the number of processes waiting for CPU, disk, and memory) in designing load indices. Although his approach of using multiple measurable metrics is similar to ours, he did not consider a systematic method for learning good workload indices. In this study, we use four measurable metrics: disk traffic, network traffic, memory

occupancy, and CPU occupancy, and design a CANN to relate these time-varying metrics in order to predict workload. To facilitate learning, we used a synthetic workload generator [7] to collect workload patterns and to regenerate the workload collected in the presence of foreground jobs.

3.1. Design Goals

Ideally, we would like to rank alternative destinations for an incoming job by their respective completion times. However, it suffices to determine only a relative (site-specific and configuration-specific) measure of completion time, and predict the *relative* completion times of a job at different sites. As a point of reference, we can use the completion time of the job on an idle file server. Hence, we now predict, *for each pair of sites*, the difference between their relative completion times. We express accuracy as the *percentage of correct comparisons*. We can stop refining our predictions when sufficient accuracy is achieved. We also need to ensure that the anti-symmetry of comparison — whenever A is better than B, B is worse than A — is preserved during learning.

Our goal in load-index learning is to estimate functions F_s, one per site, such that (i) the functions do not assume any prior information about the job being scheduled; (ii) they depend only on (smoothed, or otherwise processed) local resource-utilization values; and (iii) the ranking induced on alternative destinations by these functions is consistent with the ranking induced on those same destinations by the true completion times. Before the objective of learning is stated, we define the following symbols.

S: Set of S sites in the network.

F: Set of F foreground test jobs. 0 is the case with no foreground job.

T: Window size in time units during which workload is to be generated for the distributed computer system.

B: Set containing B background load patterns, where a background load pattern is defined as a collection of background jobs, each characterized by a prescribed site and time of arrival. 0 is the case with no background load.

$\hat{\mathbf{f}}_{b,f(t),s}$: Vector of values derived from recent behavior of loads on different locally accessible resources at site s for background load pattern b before foreground job f is started at time t. $\hat{\mathbf{f}}_{b,f(t),s}$, therefore, depends on t, the time that job f is started, but not on the behavior of job f. In subsequent discussions, we use f instead of $f(t)$ when t is constant. Note that $\hat{\mathbf{f}}_{b,f(t),s}$ is a matrix of 4 rows (representing the four resources) and T columns.

$F_s^W\left(\hat{\mathbf{f}}_{b,f,s}, f\right)$: Value of site s's load-index function, where W denotes the current values of the *weights* (or parameters) of F_s. We will omit superscript W from our equations whenever it is obvious.

$C\left(\hat{\mathbf{f}}_{b,f,s}, f\right)$: completion time of job f at site s under background load b.

The objective of load-index learning, then, is

$$F_s\left(\hat{\mathbf{f}}_{b,f,s_1}, f\right) - F_s\left(\hat{\mathbf{f}}_{b,f,s_2}, f\right) = \frac{C\left(\hat{\mathbf{f}}_{b,f,s_1}, f\right) - C\left(\hat{\mathbf{f}}_{b,f,s_2}, f\right)}{C\left(\hat{\mathbf{f}}_{0,f,s_{ref}}, f\right)} \tag{3.1}$$

for all foreground jobs f, for all background load patterns $\hat{\mathbf{f}}_{b,f,s}$, and for all pairs (s_1, s_2) of sites. Note that $C\left(\hat{\mathbf{f}}_{0,f,s_{ref}}, f\right)$ represents the completion time of job f at the idle reference site. Further, note that in Eq. (3.1), we assume that load indices are computed at time t for both sites s_1 and s_2. In practice, the above assumption is not correct, as it is difficult to synchronize the computation of indices at different sites. Moreover, there are delays in obtaining load indices from remote sites. These errors are not considered in the design of load-index function but rather in other parameters of the load balancing policy [6].

Given our objectives stated in Eq. (3.1), O^A defined in Eq. (2.3) corresponds to the left-hand side of Eq. (3.1). Therefore, O_D is given by the right-hand side of Eq. (3.1).

Eq. (3.1) is true when waiting time grows linearly with service time. This is a reasonable assumption of any fair scheduling policy, and is approximately true of UNIX-related operating systems that employ variants of round-robin scheduling.

To achieve our goal of obtaining *accurate comparison in most situations*, we would like both sides of Eq. (3.1) to have the same sign. We, therefore, seek to learn load-index functions that, when compared across sites, correctly predict (for most training patterns) the site with the lower relative completion time; that is,

$$\text{Minimize} \sum_{\tau \in \mathbf{R}} E(\tau), \tag{3.2}$$

$$\text{where } E(\tau(b, f, s_1, s_2)) = \begin{cases} 0 & \text{if } \left[F_s\left(\hat{f}_{b,f,s_1}, f\right) - F_s\left(\hat{f}_{b,f,s_2}, f\right) \right] \times \left[C\left(\hat{f}_{b,f,s_1}, f\right) - C\left(\hat{f}_{b,f,s_2}, f\right) \right] > 0 \\ 1 & \text{otherwise} \end{cases}$$

3.2. Experimental Results

We can now achieve the design goal in Eq. (3.2) by a CANN that pre-process a window of workload data using filters and extrapolation. As mentioned earlier, we use five different filters, with cutoff frequencies at, respectively, 1%, 5%, 10%, 25%, and 50% of maximum frequency. We apply two extrapolation methods: one using linear fitting and the other using exponential fitting. The area under each extrapolated curve represents the anticipated fraction of resources that are not available to the incoming job to be scheduled, and is used as an input to the load-index function. Since there are 4 different resources, 5 filters, and 2 extrapolation methods, our indices are functions of 40 variables. Vectors of these variables constitute inputs I_L and I_R in Figure 1.

To demonstrate that a CANN can be designed to predict workload, we use a synthetic workload generator we have developed earlier [7] to collect load patterns on a configurationally heterogeneous system consisting of (i) a diskless Sun 3/50 with 4 Mbytes of RAM; (ii) a diskful Sun 3/50 with 4 Mbytes; (iii) a diskful Sun 3/260 with 8 Mbytes; and (iv) a diskless Sun 3/60 with 24 Mbytes. The four workstations were connected by a single 10 Mb/s Ethernet. We used a total of ten test jobs: three for sorting records of various sizes with various amounts of initial memory, two for uncompressing files of different sizes, and five from the Perfect Club suite of benchmarks (FLO52Q, TRFD, QCD, TRACK, and DYFESM) [1]. Each training pattern presented to the CANN contains 40 projected resource-utilization values and 2 additional fields, $C\left(\hat{f}_{b,f,s}, f\right)$ and $C\left(\hat{f}_{0,f,s_{ref}}, f\right)$.

The training algorithm described in Section 2 was applied to 40×40×1 networks (*i.e.*, networks containing 40 hidden units each). In determining the number of hidden units, we used the popular rule of thumb that a network must contain approximately half as many weights as there are training patterns. The learning parameter η of the back-propagation algorithm was set to 0.001, and the momentum parameter set to 0.99.

We began by randomly assigning 10% of the jobs to the test set and the remaining 90% to the training set. The networks were trained using the training set, and their accuracy measured on the test set. Training and testing were applied alternately; during each *epoch* of training, the networks were trained on 1,000 randomly chosen comparisons from the training set, and tested on 100 randomly chosen comparisons from the test set. Each comparison involved two loading situations for the same job. Training was done in two stages. Stage 1 started with two identical networks (with random initial weights) for each site; these were trained to compare different loading conditions for the same job at the same site. In Stage 2, just the left networks from each site were further trained to compare different sites for the same job under the same system-wide load pattern. Unlike in Stage 1, the roles of the left and the right load-index functions (Figure 1) could not be reversed during Stage 2.

Load indices output by CANNs at different sites are compared in order to determine the least-loaded site. Given that load balancing seeks maximum speed-up over local execution, we can evaluate the new load-index

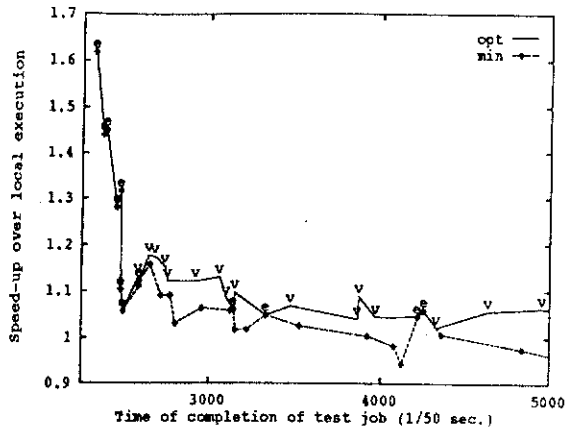


Figure 2 Performance under the worst-behaved load.

Performance is computed for all test jobs under workload pattern 23; policies applied are *opt* and *min*.

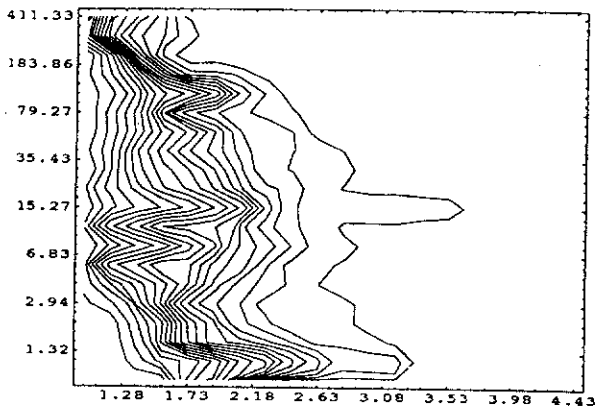


Figure 3 Policy: *min*; no overheads or delays.

The Y axis represents length of jobs in seconds; the X axis represents speedup over local execution.

functions by the speed-up attained if each incoming job is scheduled at the least-loaded site. Figure 2 shows the results for our 4-processor system. Test jobs were introduced on a synthetic workload 2,252 time units into the experiment, each time at a different site. Two policies were compared: (i) *opt*, which places each incoming job at the site with the optimum (least) completion time; (ii) *min*, which schedules incoming jobs at the site with the smallest load index. Assuming no overhead of remote execution, we calculated, for each test-case, the speed-up over local execution achieved using these policies. Figure 2 plots these speed-ups against the time of completion; it shows that while the site with the least load index behaves as well as the optimal site for short jobs, its performance is suboptimal for long jobs. The letters on individual curves indicate the optimal site ('e' for elaine, and 'v' for 'vyasa'). The policy *min* runs all the jobs at elaine, the site with the least load index at the time of arrival; therefore, its performance is suboptimal only when a site other than elaine is optimal.

We evaluated *min* on all 10 jobs and all 24 load patterns. To relate job length with achieved speed-up, we created data-pairs $<l_i,e_i>$, where l_i is the length of the job and e_i, the corresponding speed-up. Figure 3 shows a contour plot of the probability of achieving certain speed-ups for jobs of certain length. The X-values represent speed-ups over local execution, and the Y-values, length of the job in seconds. Y-values range from 0.66 seconds to 439.94 seconds and, X-values, from 0.924524 and 4.92187. We used logarithmic scaling for the Y-axis. Nineteen contours, each connecting X-Y points having equal cumulative probability of speed-up, are shown; they divide up the space into twenty regions of equal (5%) probability. While speed-ups higher than 1.5 (more than 50% improvement over local execution) occur quite frequently for short jobs, they rarely occur for jobs more than 3 minutes long.

4. PREDICTING RELATIVE CONVERGENCE TIMES IN DESIGNING ANNS

Our second application of the CANN presented in Section 2 is in a population-based system for the automated design of ANNs. Population-based learning methods [10] maintain a population of completing ANNs and choose to modify a specific ANNs depending on partial performance results of all ANNs tested. In designing ANNs, a population-based method maintains a pool of ANNs [8]. Within the time allowed, it divides the time allowed into quanta, selects and trains one promising ANN for a quantum using a traditional learning method (such as back propagation), updates performance obtained at the end of the quantum, generates new ANNs when none of the existing ANNs is promising, and discards an existing ANN when it is found to be inferior.

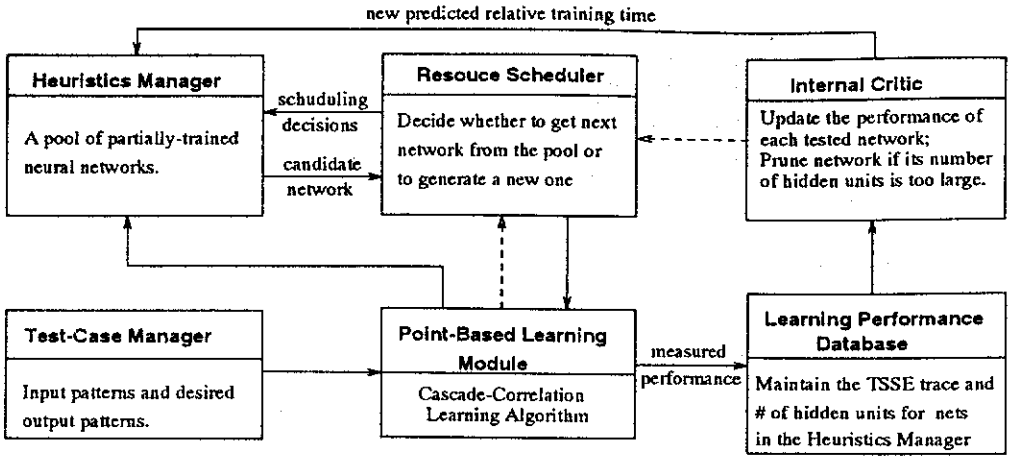


Figure 4 Population-based learning system for designing ANNs.
(Dashed arrows indicate completion signals sent to the Resource Scheduler.)

4.1. Population-based Design System for ANNs

In general, learning can be classified as point-based and population-based. In *point-based methods*, only one incumbent is considered at a time, and the learning system switches to a new incumbent ANN after discarding the previous incumbent. Traditional ANN learning algorithms are point-based and are generally based on gradient-descent search. The major drawback of these algorithms is that they may reach a local minimum, and further learning would not reduce the error. One way to overcome this problem is to keep a population of partially trained ANNs and to train the ANN that is most likely to converge in the shortest time. This prediction can be done by the CANN discussed in Section 2. Figure 4 shows the structure of our design system.

In this paper, we use Fahlman's Cascade Correlation learning algorithm (CAS) [2] as the point-based learning mechanism. There are two training phases in CAS: *TRAIN_INPUT* for adding new hidden units, and *TRAIN_OUTPUT* for getting appropriate weights in the output layer. These two phases are applied alternately. Here we define one *learning episode* as one TRAIN_INPUT phase followed by one TRAIN_OUTPUT phase. Note that the number of hidden units required when convergence is reached is unknown ahead of time.

The minimum number of hidden units required and the corresponding ANN configuration for solving the application problem are kept in the *incumbent* N_{incum} in the Internal Critic. Initially, the system trains a feasible ANN using the CAS algorithm, and stores the number of hidden units needed in N_{incum}.

The Heuristics Manager in Figure 4 is responsible for maintaining a fixed number of candidate ANNs, each partially trained, and for generating new candidate ANNs at the request of the Resource Scheduler. Initially, it generates a pool of candidate ANNs starting from a primary ANN (which includes only the input and output layers) with random initial weights; this is followed by N_{incum}/k training episodes, where k is heuristically set to 4.

In the design process, the time allowed is divided into quanta. At the beginning of each quantum, the Resource Scheduler decides whether to generate a new candidate or to train an existing candidate for one episode. If an existing candidate is to be trained, then the Resource Scheduler chooses a candidate from the pool that requires the minimum predicted (relative) training time for convergence. This candidate ANN is trained by CAS for one training episode. The performance of training is then saved in the Learning Performance Database which stores the ANN configuration and its temporal trace of TSSE.

Using the performance data collected, the Internal Critic predicts using a CANN the relative convergence time of one candidate ANN with respect to another. The prediction leads to the following alternative actions.

(1) If the ANN selected has been trained to convergence and its number of hidden units is less than N_{incum}, then N_{incum} is updated. The Resource Scheduler then instructs the Heuristics Manager to generate a new candidate ANN, and schedules time to train the new ANN for $N_{incum}/4$ training episodes.

(2) If the ANN has not been trained to convergence and the number of hidden units is smaller than $N_{incum} - 1$, then the Internal Critic will compute its new predicted relative convergence time using the CANN and updates the Performance Database. Note that if a non-converged candidate ANN has $N_{incum} - 1$ hidden units, then this candidate will require at least N_{incum} hidden units when training converges, and, hence, can be pruned.

(3) Otherwise, the candidate ANN is pruned, and the Heuristics Manager generates a new ANN. The cycle of selection/training/generation is repeated until the time allowed is expended.

4.2. Experimental Results

To implement our system for designing ANNs, we trained a CANN that can accurately predict for any two partially trained candidate ANNs which one will converge faster if both are trained to completion. We generated the training patters for this CANN as follows. First, we trained 15 ANNs using CAS for the two-spiral problem [2]. Let $T_{\max}$ and $T_{\min}$ be the maximum and minimum training times, respectively, for these 15 ANNs. Second, we obtained 20 different segments from each TSSE-versus-time trace, each in the time interval $[0, t_i]$, where t_i was a random point in the trace. (These resulted in 300 different combinations.) We then filtered each segment by 4 different Butterworth filters and extrapolated the smoothed trace using linear and exponential fitting methods in order to obtain 8 predicted convergence times. These 8 predicted times and the TSSE at the time when the segment was cut, formed an input vector for our CANN. This input vector, when combined with the actual time needed for convergence, formed a training pattern.

Since the training times in these 300 training patterns were unnormalized, and our CANN was trained to predict relative training times, we need to normalize the training times using the following equations.

$$T_{norm} = \begin{cases} 1 & \text{if } T > T_{\max} \text{ or } T < 0 \\ \dfrac{T - T_{\min}}{T_{\max} - T_{\min}} & \text{if } T_{\max} > T > T_{\min} \\ 0 & \text{if } T < T_{\min} \end{cases}, \qquad TSSE_{norm} = \frac{TSSE}{TSSE_{\max}}, \tag{4.1}$$

where T is the unnormalized training time, and $TSSE_{\max}$ is the maximum TSSE in the 15 ANNs trained.

Using the normalized training patterns, we then trained our CANN to differentiate between any two patterns which one would have a smaller convergence time. The configuration of each subnet in our CANN is 9-15-1. We stopped training when we reached 80% accuracy.

We tested the CANN developed using a simple two-region classification problem and the two-spiral problem. In the classification problem, the goal is to identify which one of two regions (Figure 5) that a point belongs to, whereas in the two-spiral problem, the goal is to identify which one of two spirals that a point belongs to. The procedure of generating test patterns is the same as that of generating training patterns. Table 1 compares the accuracies of our CANN with those of simple prediction methods based on filtering and extrapolation alone. We see that the accuracy of prediction is improved by using our CANN.

Figure 6 shows two contour graphs to depict the accuracy of our CANN. A point (x, y) represents the case when x% of the TSSE trace of one ANN and y% of the TSSE trace of another ANN are fed into our CANN, and the corresponding z value is the average accuracy that our CANN can predict the ANN with a shorter convergence time. These contour graphs indicate that our CANN is accurate when both ANNs have been trained to a reasonable amount, and is the least accurate when one or both of them have received very little training.

It is important to point out that our design system tries to avoid the dark regions in the contour graphs because it trains each candidate ANN by $N_{incum}/4$ learning episodes when the ANN is generated. Hence, in our system, all the input TSSE traces to our CANN are from ANNs that have been trained for some time. The average accuracies of our CANN are, therefore, higher than those reported in Table 1. For instance, the average accuracy in the first contour graph in Figure 6 is 71.59% when we consider only the part of the graph where $x > 20\%$ and $y > 20\%$.

Finally, we compare in Figure 7 the average performance of our design system with that of a naive method and an *improved naive method*. In the naive method, a network is trained to convergence before switching to a new network, whereas in the improved naive method, training stops whenever the number of hidden units is larger than

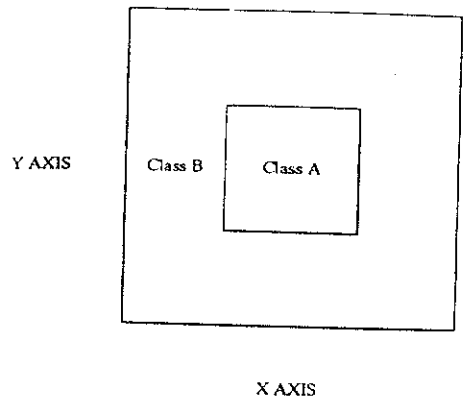
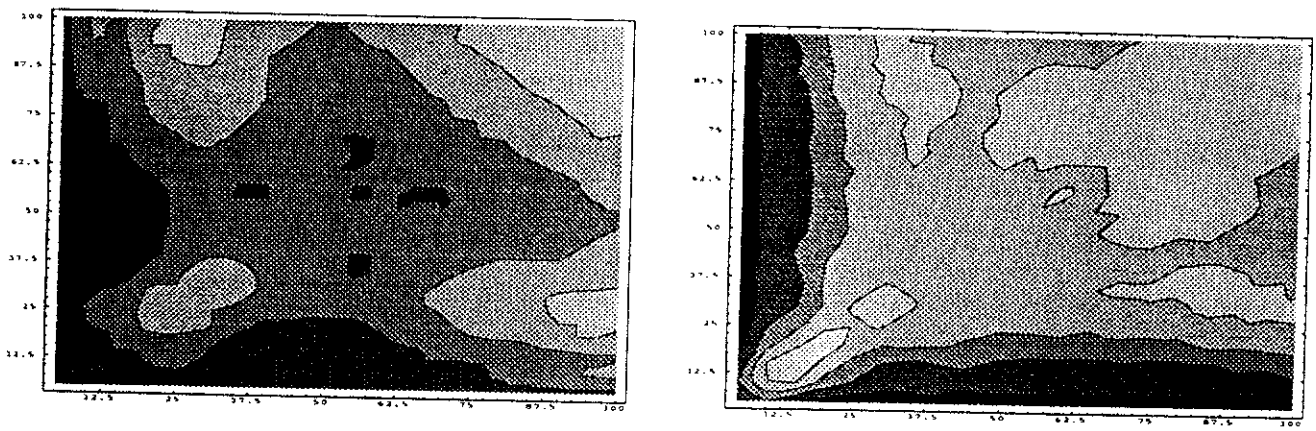


Figure 5 Two-region classification problem.

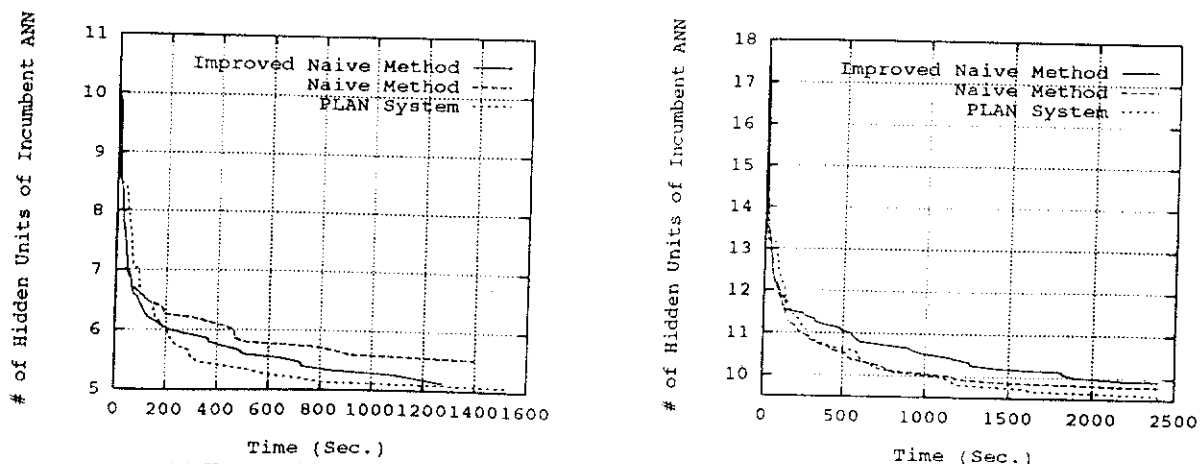
Table 1 Accuracies of CANN and those of simple prediction methods.

Application		2-Region Class.	2-Spiral Problem
CANN		67.72%	72.88%
Linear Fitting	Filter 1	57.32%	55.83%
	Filter 2	56.43%	56.12%
	Filter 3	61.38%	60.19%
	Filter 4	58.33%	57.21%
Exponential Fitting	Filter 1	59.21%	61.22%
	Filter 2	57.15%	54.31%
	Filter 3	59.54%	61.92%
	Filter 4	57.98%	52.65%



(a) Two-region classification problem (b) Two-spiral problem

Figure 6 Average accuracies of our CANN in identifying which one of two ANNs has shorter convergence time. The 5 grey levels in these graphs represent the following accuracy intervals: [0, 0.525), [0.525, 0.621), [0.621, 0.717), [0.717, 0.813), and [0.813, 1.0], and darker colors mean lower accuracies. }



(a) Two-region classification problem (b) Two-spiral problem

Figure 7 Performance of our ANN design system.

$N_{incum} - 1$. Each curve in Figure 7 represents an average of 15 runs.

The interpretation of the graphs in Figure 7 is as follows. When the total time allowed is small, the two naive methods are slightly better in finding ANNs with smaller number of hidden units. In this case, our design system spends more time to generate new candidate ANNs and does not have enough time to train any one to completion. In contrast, when the time allowed is large, our design system is superior in identifying ANNs with small number of hidden units. For instance, in the first graph in Figure 7, our design system finds an ANN with an average of 5.5 hidden units using only 21% (*resp.* 57%) of the time needed by the naive (resp. improved naive) method to find a similar ANN. Similarly, in the second graph in Figure 7, our design time takes 8.4% and 77.8%, respectively, of the average times needed by the naive and improved naive methods to obtain an ANN with 10 hidden units. It is important to note that the improvement is even greater when the time allowed is large.

5. CONCLUSIONS

In this paper we have presented the design of a comparator neural network that is useful for relative prediction of two time series based on a user-specified objective measure. A relative comparison is simpler than an absolute comparison as objective values are normalized, hence removing range-dependent objective values from consideration. Our design takes a time window of values, filters them by filters of various cut-off frequencies, extrapolates them into the future, and evaluates the objective based on the extrapolated curves. We have applied this design in workload prediction and in an automated neural-network design system. In both cases, we have shown that prediction is significantly enhanced by the proposed comparator neural network.

References

1. M. Berry et al., "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," *International Journal of Supercomputing Applications* 3(3) pp. 5-40 (1989).
2. S. E. Fahlman and Christian Lebiere, "The Cascade-Correlation Learning Architecture," pp. 524-532 in *Advances in Neural Information Processing Systems 2*, ed. D. S. Touretzky, Morgan Kaufmann, San Mateo (1990).
3. D. Ferrari, G. Serazzi, and A. Zeigner, *Measurement and Tuning of Computer Systems,* Prentice-Hall, Englewood Cliffs, NJ (1983).
4. D. Ferrari, "A Study of Load Indices for Load Balancing Schemes," pp. 91-99 in *Workload Characterization of Computer Systems and Computer Networks*, ed. G. Serazzi, Elsevier Science, Amsterdam, Netherlands (1986).
5. N. H. Goddard, K. J. Lynne, T. Mintz, and L. Bukys, *Rochester Connectionist Simulator*, Tech. Rep., Univ. of Rochester, (Oct. 1989).
6. P. Mehra, *Automated Learning of Load Balancing Strategies for a Distributed Computer System*, Ph.D. Thesis, Dept. of Computer Science, Univ. of Illinois, (Dec. 1992).
7. P. Mehra and B. W. Wah, "Automated Learning of Workload Measures for Load Balancing on a Distributed System," *Proc. Int'l Conference on Parallel Processing*, pp. III-263-III-270 CRC Press, (Aug. 1993).
8. C.-C. Teng and B. W. Wah, "An Automated Design System for finding the Minimal Configuration of a Feed-Forward Neural Network," *Int'l Conf. on Neural Networks* 3 pp. 1295-1300 IEEE, (June 1994).
9. G. Tesauro and T. J. Sejnowski, "A Parallel Network that Learns to Play Backgammon," *Artificial Intelligence* **39** pp. 357-390 Elsevier Science Pub., (1989).
10. B. W. Wah, "Population-Based Learning: A New Method for Learning from Examples under Resource Constraints," *Trans. on Knowledge and Data Engineering* 4(5) pp. 454-474 IEEE, (Oct. 1992).
11. S. Zhou, *Performance Studies of Dynamic Load Balancing in Distributed Systems*, Tech. Rep. UCB/CSD 87/376 (Ph.D. Dissertation), Computer Science Division, Univ. of California, (1987).