AD-A226 789

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

2

. AGENCY USE ONLY (Leave blank)	2. REPORT DATE 8/30/90	3. REPORT TYPE AND DATES COVERED Annual TR/FTR - 7/21/89 - 7/20/90

. TITLE AND SUBTITLE
Analog Optical Neural Nets: A Noise Sensitivity Analysis

5. FUNDING NUMBERS
C: F49620-89-C-0115

. AUTHOR(S)
Dr. Michael W. Haney
Mr. James J. Levy
Dr. Ravindra A. Athale

. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
BDM International, Inc.
7915 Jones Branch Drive
McLean, VA 22102-3396

8. PERFORMING ORGANIZATION REPORT NUMBER
BDM/MCL-90-0757-TR

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
USAF, AFSC
Air Force Office of Scientific Research
Building 410
Bolling AFB, DC 20332-6448

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
2305/B1

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unlimited

AFOSR-TR- 90 0915

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

See Page iii in Report

14. SUBJECT TERMS
Optical Neural Nets, Opto-electronic noise

15. NUMBER OF PAGES
40

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	unlimited

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

90

BDM

A Ford Aerospace Company

BDM INTERNATIONAL, INC.
7915 JONES BRANCH DRIVE
McLEAN, VIRGINIA 22102-3396
(703) 848-5000

ANALOG OPTICAL NEURAL NETS: A NOISE SENSITIVITY ANALYSIS

ANNUAL TECHNICAL REPORT:

PHASE I FINAL REPORT

AUGUST 30, 1990

BDM/MCL-90-0757-TR

Sponsored by
Air Force Office of Scientific Research
Under Contract #F49620-89-C-0115
Technical Monitor: Dr. Alan Craig, AFOSR/NE

Abstract

artificial intelligence

Neural networks represent a promising alternative to traditional ~~AI~~ approaches. The development of analog optical implementations of neural networks such as the multilayer perceptron with learning by backward error propagation (BEP) requires an understanding of the noise sensitivity of such architectures. The objective of this program is to study the effects of component and system noise on the performance of such optical implementations. The method used is computer simulation.

In this first phase of the program, the one-hidden layer perceptron with back propagation was simulated using a simplified, device-independent noise model. The results point to a distinct noise threshold above which the learning mechanism is corrupted. The efficiency of learning based on variations within back propagation and on the initializing method was also studied.

In the next phase, a device-dependent noise model will be used. To this end a plausible all-optical architecture capable of both the forward pass and backward error propagation steps of training data presentation has been proposed.

Table of Contents

I. BACKGROUND

A. Program Motivation

Neural networks are computers that are based on organizational and functional principles of information processing systems found in nature, like the brain and retina. Hence, neural nets consist of many analog processing elements that are densely interconnected and they are applied to problems such as sensor pre-processing, pattern recognition/classification, and motor control. In neural networks, long-term information is stored as interconnection weights which signify the efficacy of interaction between neurons. Transient information is represented by the neural outputs, which, for a given neuron, is a nonlinear function of its state of activation. Each neuron's state of activation is determined by a number of factors: its external inputs, which are the weighted outputs from other neurons; its previous states of activation; and other specific and nonspecific global signals. The values of the interconnection weights change more slowly than the neuron's state of activation. The gradual evolution of the interconnection weights has been widely postulated as the primary learning mechanism that makes animals adapt to a constantly changing environment. The proposed neural net models for problem solving attempt to emulate these intriguing characteristics of biological information processing systems. References 1, 2, and 3 contain the latest work on neural net development.

The primary advantage of adaptive neural net problem solving approaches over conventional methods is that the networks learn how to solve problems semi-autonomously; from labeled or unlabeled training data, the network learning rules calculate weights which will produce the appropriate outputs. Thus, there is no need for standard artificial intelligence techniques like investigation of the nature of the problem and extensive programming of solution strategies---all that is required is access to raw data. This approach is especially useful in several scenarios: when designing systems that can be applied to a variety of problems with little modification; when the size and complexity of the problem makes rule discovery by hand prohibitively expensive; when rapid solution to a problem is desired; when the nature of the problem is dynamic; or when it is difficult to ascertain the structure of the problem due to noisy and/or distorted data.

Nonadaptive neural networks are programmed to solve specific problems. With this approach, the appropriate set of weights and initial conditions are determined by the user. Later, when inputs are presented, the state of the network converges to the proper answer through the network dynamics. Since either extensive calculations must be performed to find the appropriate set of weights or prior knowledge of the desired processing must be used to simplify the network structure, a given network must be used many times to justify the cost of its construction.[4] Hence, this type of network is most suitable for sensor pre-processing type applications where the same operations must be performed on many data sets.

Optical systems have been proposed as candidates for neural network implementation for a number of reasons. Foremost are the analog and parallel natures of neural computation---optical systems have been employed for a number of years to solve significant problems, like synthetic aperture radar imaging and RF spectrum analysis, with parallel, analog hardware. In addition, neural nets often require complex and dense interconnections between the neurons---and interconnection and communication are the primary advantages optics has over electronics. Finally, neural nets require analog storage of interconnection weights that can be accessed and updated in parallel and several 2-D and 3-D optical devices exist that can provide this functionality.[5,6]

For analog optical numeric processors, the accuracy of the overall computation is strongly dependent on the accuracy of the analog optical devices. When such numeric processors were first proposed, the numeric computations considered for analog optical systems were primarily linear (matrix operations) and the accuracy of devices was quite low; therefore, so was the accuracy of the overall processor. Hence, the available applications were limited to those requiring low precision. This motivated the investigation of analog optical systems for neural nets, which were postulated to require low accuracy computation. The first neural network that optics researchers chose to implement, the Hopfield model,[7,8] was indeed relatively insensitive to inaccuracies in the response of the analog components. As it turns out, the very features of the Hopfield model that make it relatively insensitive to inaccurate components, namely the particular type of distributed and redundant storage/computation, also limits its storage capacity, and hence. its utility.

Since the publication of the Hopfield model, there has been a number of potentially useful neural network models reported in the literature along with proposals for their optical implementation.[9] However, it has been commonly assumed that since the Hopfield model was relatively impervious to hardware imprecision, so are these other models. This is not necessarily true---each model must be examined closely to determine its own sensitivity to analog hardware imprecision. Since many modern neural net models improve upon the storage capacity of the Hopfield model by means of less redundant storage techniques, they may lead to systems which are more sensitive to noise. Except for a few models, most notably those examined in our previous work, the noise tolerance of neural network models is unknown. Dependence of the noise sensitivity on the size of a given neural net model is an issue of great concern since the neural net approach to problem solving may become competitive with conventional approaches only when the network is very large.

It is clear from the above discussion that the detailed study of a neural network when implemented in any analog technology (optical or electronic) is of critical importance. Without an adequate understanding of neural nets, the success of the neural net research endeavor itself can be jeopardized.

B. Program Goal

The main goal of the proposed research program is to study the effects of component and system noise on the performance of optical implementations of selected neural net models. The noise could be due to variations in the response of different components, finite accuracy in controlling signal amplitudes, or signal- and time-dependent noise due to quantum and thermal fluctuations. It is suspected that the tightest bounds on the size and speed of optically implemented neural nets will come from these noise sources rather than the scaling limits of optical technology. Since the size and speed of neural net processors will be factors that influence their utility, the results of this study will be critical in determining the ultimate applicability of optical neural nets. The identification of those parts of the neural net model that are particularly sensitive to system noise will stimulate investigations into new techniques of data representations and formatting to increase the robustness of the models. Similarly, the identification of the limiting devices or materials in the optical implementation of the selected neural net models will lead to exploration of different technological and architectural alternatives for improved performance.

C. Statement of Work

This report covers the first phase and year of a two-year two-phase research and development program to quantitatively study the effects of system noise on the performance of optical neural net processors. The first phase is concerned with the noise effects on neural nets without regard to technological implementation. The second phase is focusing on the noise effects in optical implementations. The two phases are, however, intimately coupled and the results from the first phase will provide the foundation for the second phase. Following is the detailed breakdown of each phase into tasks and a description of the goal of each task. Although Phase II is not reported in this final report, its tasks are included below for completeness and continuity.

1. Phase I: Simulation of the Noise Sensitivity of Neural Net Models

In this phase, a technology-independent analysis of the noise sensitivity of selected neural net models was performed. This phase was divided into the following tasks.

a. Task I.1: Selection of Neural Net Models

Given the explosion in the number of different neural net models that have been proposed over the last four years, it is critical that this research effort be focused on a few selected models. Examples will be selected from both adaptive and non-adaptive networks. The potential for optical implementation will be an important criterion for selection. Another critical factor will be the successful and competitive application of the model to an important problem. The backward error propagation, higher-order threshold logic units, and MAXNET are candidate network models.

b. Task I.2: Identification and Analysis of the Functional Units of the Neural Net Models

Processing elements (or neurons) and interconnection weights are the two ubiquitous functional units of any neural net model. The operational characteristics of these functional units also vary widely for different models, such as the bidirectional neuron for the backward error propagation model, whose transfer function depends on the direction of signal flow. In this task the exact operation of these functional units will be detailed and the parameters for statistical variations will be selected. Although this phase is meant to perform technology independent analysis, it is clear that the selection of the parameters will be influenced by the potential physical realization.

c. Task I.3: Computer Simulations of the Neural Net Models with Statistical Variation

This is the heart of the first phase. In this task, quantifiable performance metrics will first be identified for specific applications. Simulations of the selected neural net models will be performed with different degrees of statistical variations in the functional units. The noise sensitivity will also be studied as a function of the network size and the statistics of the inputs. These simulations will provide a good idea about the impact of noise in different functional units on the network.

2. Phase II: Simulations of Optical Implementations of Selected Models

The neural net models studied in the first phase will now be mapped onto optical technology. The results of the previous task will provide the foundation upon which the work in this phase will be built as described in the following task description.

a. Task II.1: Optical Implementation of the Functional Components

In this task, we will identify several potential optical implementations of the functional units of the neural nets that were identified in Task I.2. Then we will establish a connection between the statistical parameters chosen in Task I.2 and the physical parameters associated with their optical implementation. This, in effect, will be the first step toward applying the results from the previous tasks to analyzing optical neural net processors. The source of statistical variations in these optical components will then be identified and modeled. The underlying statistical distribution for the optical components is likely to be significantly different from that assumed in Task I.2. Therefore, the results from the previous simulations may not be directly transferable to the optical domain.

b. Task II.2: Conceptual Design of the Optical Neural Net Processor

The optical functional units identified in the previous task will now be integrated conceptually to design an optical neural net processor. The issue of compatibility between different optical technologies chosen for the functional units is the critical issue here. In this step, the interface subsystems needed to assemble the optical neural net processor will be identified and modelled. BDM will not attempt a detailed engineering design of the processor, since that in itself will be a separate research project and well beyond the scope of the proposed program. We will, however, work closely with other research groups in the country in developing a realistic view of the fully implemented system.

c. Task II.3: Simulation of the Optical Neural Net Processor

In this task, the conceptual design of the optical neural net processor will be fully simulated by incorporating the statistical models for the optical components in the neural net simulations developed during the first phase. Similar to task I.3, these simulations will be performed as a function of the network size and input signal statistics. In this task, however, the performance of the optical components and subsystems will be strongly governed by the network size and hence will affect the noise sensitivity of the processor even more strongly. The result of this simulation will determine the realistic limits on the size, complexity and speed of the optical neural net processor that can be realistically built with the current technology, and the technological improvements that will be needed in order to make it of practical importance.

II. TAXONOMY OF NEURAL NETWORKS AND TRAINING METHODS

A. Categories of neural network models and model selection philosophy

Neural network models can be designed to perform several operations, e.g., associative memory, optimization, filtering, pattern classification. The performance criteria for a given model are determined by its intended application. We choose neural pattern classifiers for further investigation under this program. For these models one performance criterion is simply the number of training or test patterns the model misclassifies. Another is corruption of the learning curve during training. The effect of system noise on the selected neural net classifiers is investigated using the above-mentioned criteria.

Neural net models can be categorized according to several different parameters. The first one of these parameters is the topology of the neural net (single or multiple layers of processing elements). The second one is the processing element response (linear, hardclipping nonlinear, or continuous nonlinear). The third parameter deals with the selection of learning algorithm. Error-driven algorithms can be used with labeled training data consisting of input patterns along with their correct classification. Unlabeled training data uses self-organizing algorithms capable of autonomously clustering the input patterns into distinct classes and adjusting the internal parameters of the neural net to generate the desired classification.

The limitations of a single layer neural net model in classification have been well documented.[10] Hence we have selected a multilayer neural net classifier. It can be readily seen that a linear processing element response reduces a multilayer neural net to an equivalent single layer neural net thereby suffering from the same limitations. Therefore we have chosen a nonlinear (hardclipping or continuous) response for the processing element. For the purpose of this study we choose the error-driven learning algorithms that are used with labeled training data. The self-organizing systems were not selected because optical implementations for them are relatively less developed.

B. Backward error propagation model

The most prominent multilayer, nonlinear, error-driven neural net is the multilayer perceptron trained by the method of backward error propagation (BEP).[11]

This is a least mean-square error approach that uses a gradient descent algorithm. The internal parameters of the neural net (the connection weights and processing element thresholds) are modified such that the new weights lead to a decrease in the total mean-square error. This model is a direct descendant of the Widrow-Hoff Adaline model that was developed for a linear, single layer neural net model.[12] It has also been successfully applied to interesting problems such as distinguishing underwater man-made objects from natural ones based on their sonar returns[13] and solar flare prediction.[14]

The BEP model uses a continuously differentiable nonlinear response for the processing element. Therefore the signals propagating between different layers are fully analog. This will lead to error accumulation. This feature makes the issue of system noise particularly relevant to the BEP model and hence appropriate for this study. Fully optical implementations of BEP model have been proposed.[15] Hybrid optical-digital electronic systems have also been proposed in which part of the training procedure is performed in an auxiliary digital electronic processor.[16,17] The effect of weight quantization on the learning performance of a BEP model has been previously reported.[17,18] These studies were specifically geared towards optical or electronic implementation in which the connections were stored electronically in a digital representation. The current study extends that work to quantify the effects of analog system noise in the weights as well as in the processing element computation.

III. DETAILS OF SIMULATIONS OF BACK PROPAGATION

A. Description of tool

Throughout Phase I, our goal has been two-fold: one, to understand the dynamics of a multilayer perceptron which is learning by back propagation of errors; and two, to observe the effects of noise on these dynamics.

To these ends we have been using PC-Matlab as a programming and analysis tool. PC-Matlab works with 1-D and 2-D variables. When it is running an ".m" file (a program), it can display in text or graphical form the ever-changing state of the network. Information such as a learning curve can be stored in a ".mat" file.

The core of our simulation program is found in two modules, "binit.m" and "bctrain.m." These modules are written specifically for the case of one hidden layer, although it is simple to modify them to add layers. Among other tasks, "binit.m" loads the training data, initializes the weights and biases, and sets up the beginning of a training session. Then "bctrain."m iteratively updates the weights, tracks the mean-square error, and provides for early termination of training upon fulfillment of a convergence criterion. Note that a training seed (trseed) is used with Matlab's random number generator, so that an identical set of "random" noise spikes may be used in different runs, if desired.

These two program modules run as though the hidden layer size, initializing criteria, the learning rate, and when training should terminate are already defined in PC-Matlab's work space. A ".m" file called "bframe."m prompts for these data so that "binit.m" and "btrain.m" may be run without causing undefined-variable errors.

These three ".m" files, as well as some of the nested ".m" files, are listed in Appendix A.

B. Backward error propagation details

To provide a deeper understanding of the simulation tools, we shall now describe exactly how back propagation proceeds, giving due attention to the order followed in updating. Back propagation is defined for a multilayer perceptron, a neural network containing nodes configured in an input layer, a number of hidden layers, and an output layer. The multilayer perceptron, whether or not it has learned, performs an evaluation of the inputs via a forward pass. First, the input elements are multiplied by the weights; then they are summed and transformed ("thresholding") by a sigmoid nonlinearity in the hidden nodes. The outputs of the hidden nodes are processed in similar fashion. A one-hidden-layer architecture is shown in Figure 1. (Strictly speaking, the input layer units are not full-fledged nodes. They do not perform summing and thresholding; they simply broadcast the input signals to the nodes in the hidden layer. Therefore, many call this a *two*-layer network.) The inputs and outputs (*o* terms) are unipolar, due to the thresholding; in some cases, the inputs may be restricted to binary values. The weights and biases are bipolar and continuous.

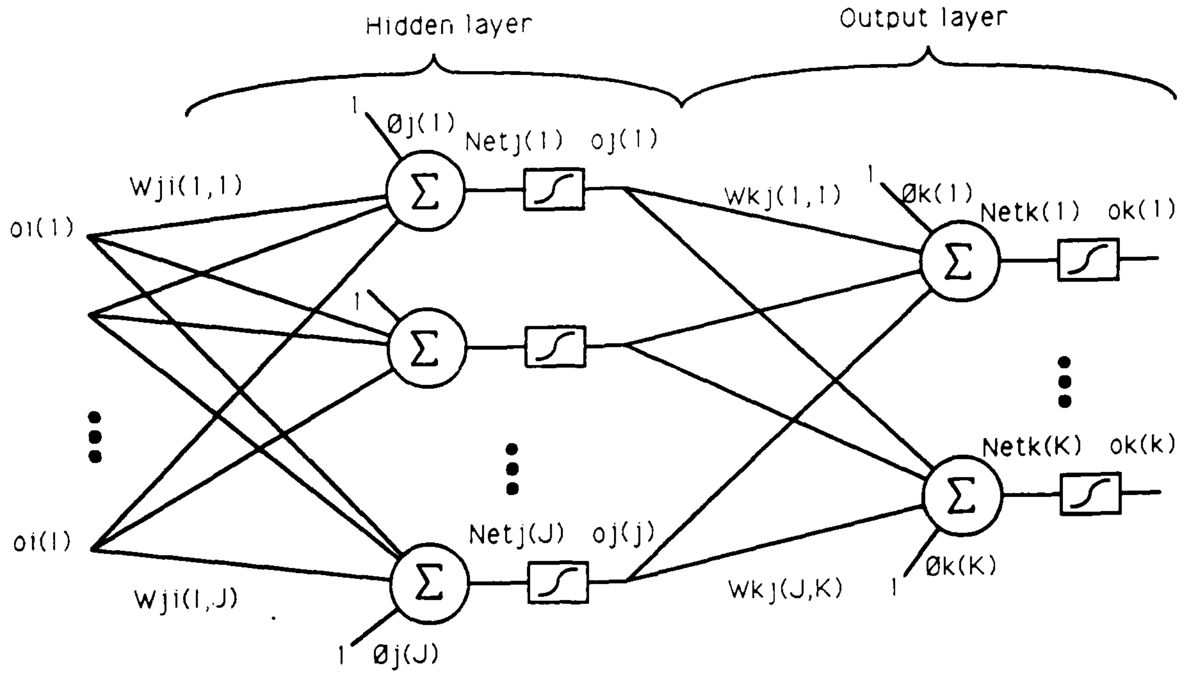

Figure 1. A perceptron with one hidden layer

1. Gradient descent

Back propagation rests entirely upon a derivation of the conditions required for gradient descent.[11] This term refers to a weight (and bias) update

$$\Delta W \propto -\partial E/\partial W \text{ where } E = \sum_r \sum_k (t_{rk}-o_{rk})^2$$

t_{rk} being the target vector's kth element for the rth training pair. That is, the weight update is supposed to move the weights in a direction in weight space in which the mean-square error over all training pairs decreases the most. This requires knowledge of the error in the output as well as the error in the internal representation. Back propagation represents the first successful method for calculating the latter error, which cannot be calculated directly since the target values for internal representation are not known *a priori.*

The steps in back propagation follow. These are specifically for the case where the thresholding ("activation function") is a sigmoid. The derivative of the activation function dictated by back propagation is manifest in the $o(1-o)$ terms. In the PC-Matlab modules, o_i is a row vector of length I; Net_j, o_j, θ_j, and δ_j are row vectors of length J, and similarly for k. The variable η is called the learning rate. In matrices, a comma denotes concatenation along columns, a semicolon along rows.

Forward Pass

1. $Net_j = [o_i,\ 1] \times [W_{ji}\ ;\ \theta_j]$ $\quad o_j = 1/(1 + \exp(-Net_j))$
2. $Net_k = [o_i,\ 1] \times [W_{kj}\ ;\ \theta_k]$ $\quad o_k = 1/(1 + \exp(-Net_k))$

Backward Error Propagation

3. $\delta_k = o_k\ (1-o_k)\ (t_k-o_k)$
4. $\delta_j = o_j\ (1-o_j)\ (\delta_k \times W_{kj}^{\mathrm{T}})$
5. $\Delta W_{kj} = \eta \times o_j^{\mathrm{T}} \times \delta_k$ $\quad \Delta\theta_k = \eta \times \delta_k$
6. $\Delta W_{ji} = \eta \times o_i^{\mathrm{T}} \times \delta_j$ $\quad \Delta\theta_j = \eta \times \delta_j$

The derivation, as presented in Rumelhart's seminal paper,[11] leaves some questions open. The first is, are the weights and biases to be updated after presentation of each training pair (pairwise updating)? Or are the weight changes to be accumulated in a separate register over the entire epoch of training pairs and then added to the weights (accumulated updating)? Rumelhart states (in text, not math) that the true gradient descent calls for the latter. Some researchers advocate pairwise updating as a means to improved performance. The listings in Appendix A are for the method of accumulated updating.

The second question is, if the weights *are* updated after each training pair, should the sequence of steps be 3, 4, 5, 6 or 3, 5, 4, 6? In the latter (layered updating), the W_{kj}'s are updated, and *then* the error δ_k is propagated through them (step 4). We believe, although it isn't explicitly stated, that the former approach (unlayered updating) is closer to gradient descent. The difference between layered and unlayered updating is often minor; be aware that for any one backward pass, only the hidden layer's weights are updated differently.

Recall what gradient descent means: the weights change in the *direction* of greatest decrease in the total mean-square error. True gradient descent requires infinitesimal weight changes. For larger weight changes, a decrease in mean-square error is not guaranteed. A n error landscape may be convoluted, and an update may move the weights too far, to a point where the error is higher. In other words, a large learning rate can create oscillations in the plot of mean-square error vs. epoch. A convoluted landscape possesses local minima, which can halt convergence, with the network stuck in an unsolved state.

These problems are often remedied by incorporating a momentum term α which introduces a component of the previous weight change into the current one. Step 5 above becomes

$$\Delta W_{kj}(n) = \eta \times o_j^{T} \times \delta_k + \alpha \Delta W_{kj}(n-1)$$
$$\Delta \theta_k(n) = \eta \times \delta_k + \alpha \Delta \theta_k(n-1)$$

and similarly for the hidden layer, where n tabulates the actual update, whether it was pairwise or not. According to Gilbert,[18] momentum is used in pairwise training to incorporate information about the previous pair, making training based on more complete information.

2. Initial conditions.

The convergence behavior produced by the back propagation algorithm depends on the initial values of the weights and biases. If all these start out equal, the algorithm keeps them so, and the network will not learn. Rumelhart's solution is to initialize the weights and biases with small random values, to provide symmetry breaking. He does not state what "small" means, however.

Is there a more intelligent way to initialize the weights? Our desire is a configuration which helps the network to solution, but is not specific to any one problem (does not "cheat"). Sheldon Gilbert[18] proposes one method based on Lippmann's[19] discussions on internal representations.

A network which has learned XOR is shown in Figure 2. Figure 3 shows the output of the upper, $j = 1$, hidden unit (vertical axis) as a function of the two inputs (horizontal axes). The magnitude of the input weight vector determines the steepness of the output---how close the "hill" of Figure 3 is to being a step function. The associated decision line, a one-dimensional hyperplane, is just the intersection of this output with the plane $o_j(1) = 0.5$. Each hidden unit in Figure 2 is labeled with a diagram of its decision region, showing this decision line. The arrow indicates in which half-plane the output is greater than 0.5. Note that the dividing decision lines actually pass through the decision region.

Gilbert's method starts with initially random weights and biases, and scales the weight vectors to a uniform value. Then it adjusts the biases so that the dividing hyperplane passes through (0.5, 0.5,...,0.5)---the middle of the decision region. Next, the weights to the output layer are all set to $1/J$, so that, regardless of the size of J, the output units start out with reasonably-sized Net_k's. For example, even if the initial $o_j = [1, 1, \ldots, 1]$, each Net_k term (with $\theta_k = 0$) will be only $J \times (1 \times 1/J) = 1$, well within the linear region of the sigmoid defined in Equation 2. (In point of fact, we have initialized the θ_k terms to small random values within [-0.5, 0.5] in our simulations.) In the simulations that Gilbert performed on 2-D problems, the initializing algorithm went on to force the dividing lines to span 360°. We chose to omit this latter restriction for two reasons: first, it seems too "forced" for typical problems, and second, it encourages deciding which dividing lines should run which way in input space, the answer to which is best found by already knowing the final state of the solved net.

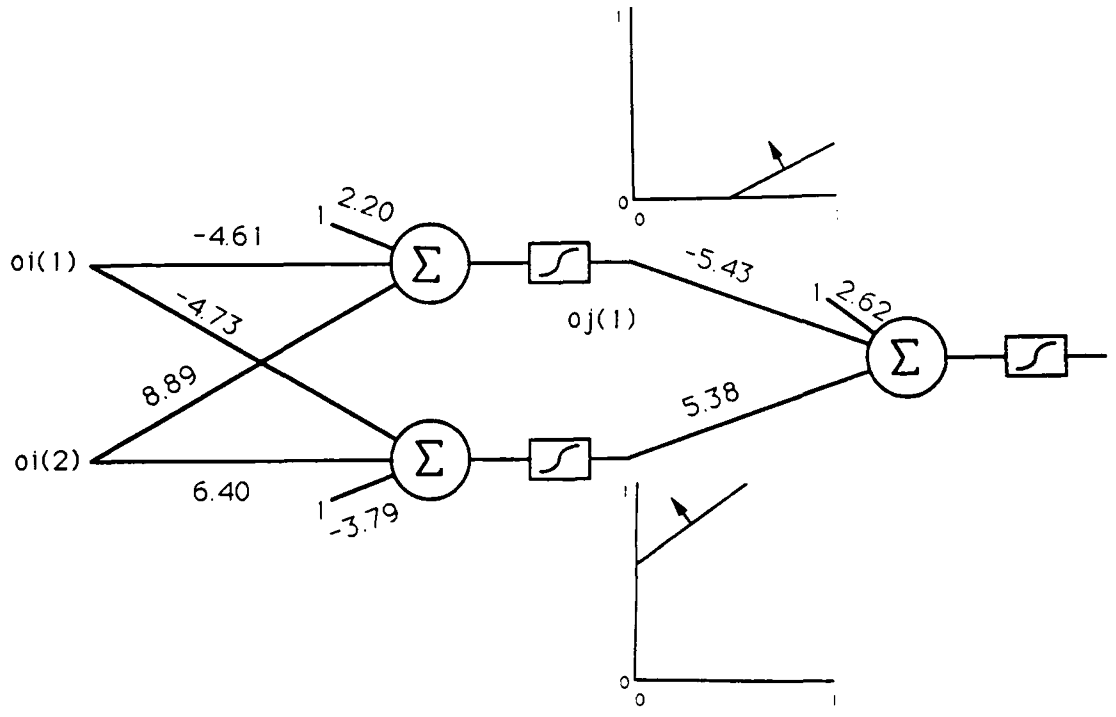

Figure 2. A $J = 2$ solved XOR configuration

3. Creating Target Vectors for Classes

For classification problems, one can express the classes one of two ways. The first way uses a dedicated output node for each class. A typical training pair has an input vector and an output vector with all elements low except for that corresponding to the correct class. The second way uses a binary representation for each class. Here a four-class problem could be implemented with two instead of four output nodes.

In addition, how the output training vector expresses "high" and "low" is important. Since the activation function asymptotically approaches 0 and 1, using these in the training vector may cause excessively large error signals to be propagated back.

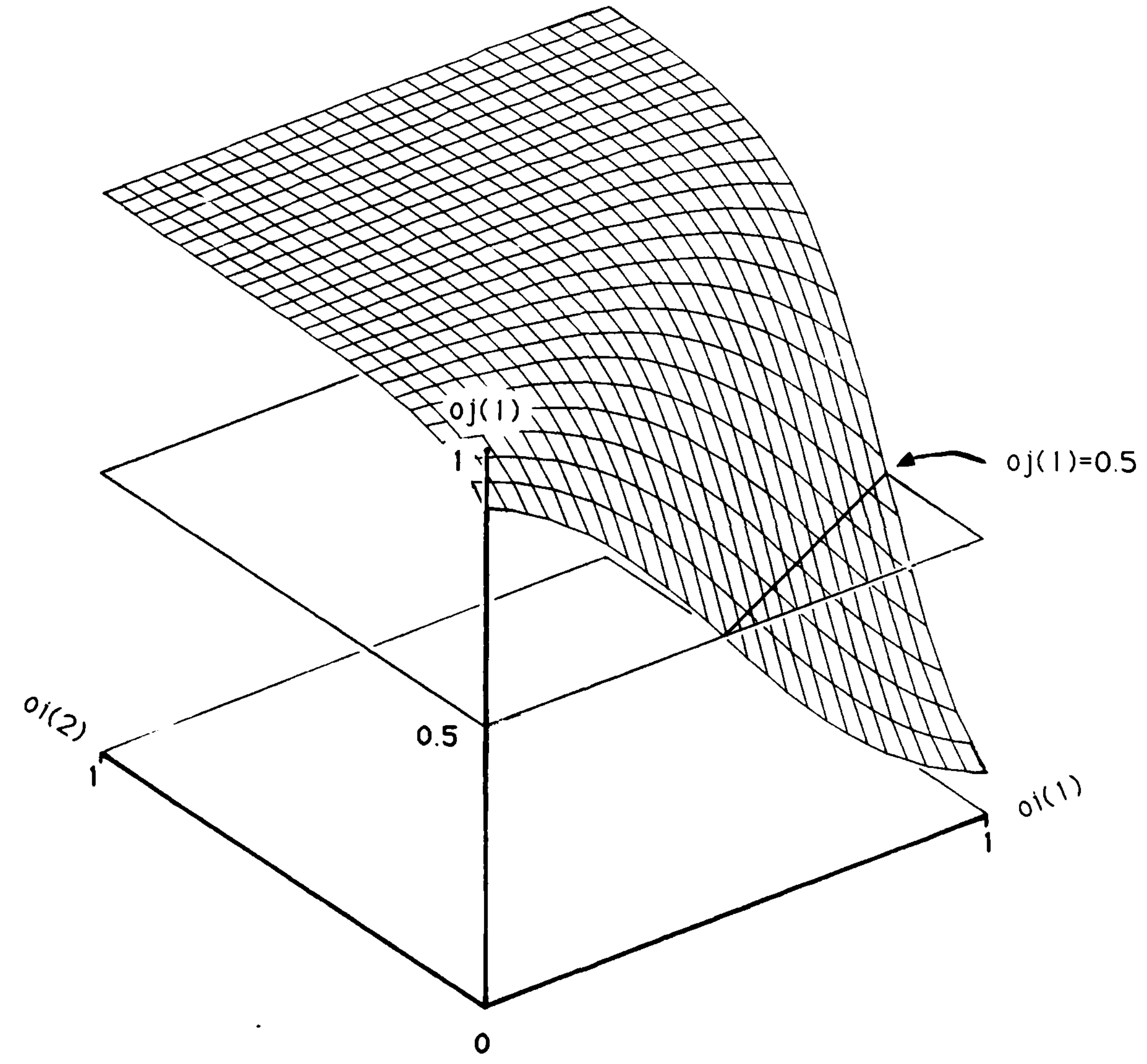

Figure 3. The output of the j = 1 hidden unit as a function of the inputs to the XOR network

All of our simulations were two-class problems, where we used $K = 1$ (binary representation). Also, unless otherwise noted, we trained our output on {0.1, 0.9} rather than {0, 1} to avoid the excessive error signals.

C. Simulation results

While the initial simulations with noise are based on a simplified noise model, they are inspired by a realistic architecture. From the outset, we realized that the storage requirements imposed by the use of the momentum term would greatly complicate any optical implementation. Having observed that the algorithm converges reasonably well with $\alpha = 0$, we chose to omit it.

The majority of our simulations are based upon pairwise layered updating during training. The only all-optical multilayered neural net architecture ever proposed, one based using a photorefractive volume hologram to store weights, implements this sort of training.[20]

Figure 4 shows a typical learning curve for XOR with layered updating, $J = 2$, $\eta = 2$, and weights and biases initially spread randomly with a uniform distribution within [-0.5, 0.5] The curve is characterized by a rapid descent, a slight leveling off ("plateau") and then a continued descent. The $J = 2$ XOR error landscape contains local minima. We have observed a probability of convergence of about 0.6.

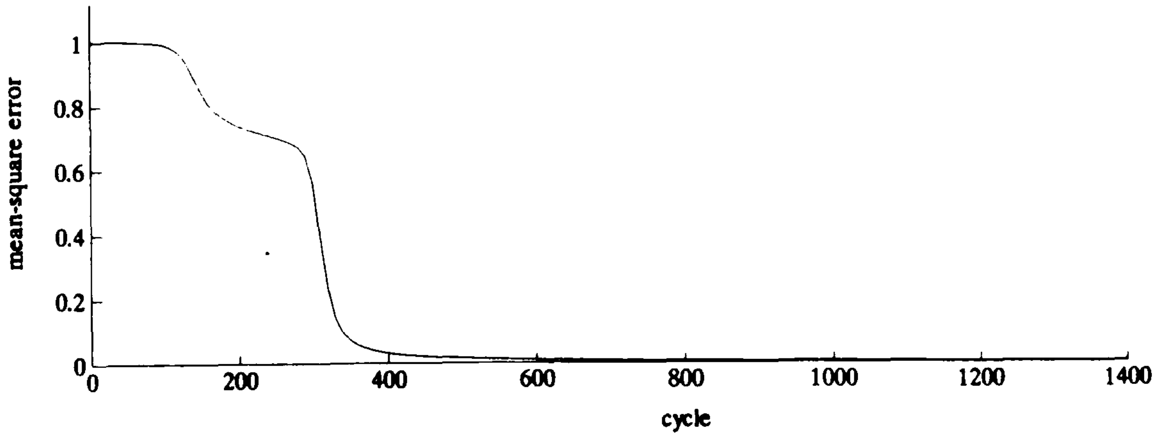

Figure 4. Learning curve for XOR with $J = 2$, $\eta = 2$, with pairwise layered updating of the weights

1. Noise

The noise was assumed to be additive; small random numbers were added to the weights and biases after every update. Unless otherwise stated, the random numbers were normally distributed.

At first, we ran a large grid of simulations, to observe the effects of noise on the learning curve as a function of three parameters: which of five sets of initial conditions were used, which of fifteen training seeds were used, and at which of three noise values (standard deviation $\sigma = 0.2$, 0.25, and 0.3).

In the learning curves, we saw that these noise values were indeed large. Noise seemed to hasten some convergences, delay others, and create stuck conditions in still others. We also noticed that a given training seed leaves a unique "signature" upon the learning curve, giving the spikes therein the same appearance among different σ's and among different initial conditions.

In our next large group of runs, we simply let each simulation terminate when the mean-square error had dropped to 0.1. Letting each run having noise use a different training seed, we tracked terminal n values to fill in the data table represented in Figure 5. In the table, each row of data corresponds to a different initial condition (i.c.) set. If n reached 2000, we assumed that the network was stuck in a local minimum.

		σ_1			σ_2	... σ_{10}
	σ=0	TRIAL 1	TRIAL 2	...TRIAL 10		
i.c. 1						
i.c. 2						
:						
i.c. 10						
i.c. 11			✗	✗		
:			✗	✗		
i.c. 100			✗	✗		

Figure 5. Data table for noise sensitivity analysis

Using the first column corresponding to each σ, we prepared the data table in Figure 6. The data from the first ten rows of each σ were used as a consistency check.

Our metric for the effects of noise was this: How many of the runs that were stuck with no noise became "unstuck" by the noise? And how many "unstuck" runs became stuck?

σ	0.04	0.06	0.08	0.10	0.12	0.14	0.16	0.18	0.20	0.22
# s-c	2	4	7	13	16	17	20	17	18	21
# c-s	3	6	7	15	20	25	29	27	25	26

Figure 6. Numbers of significant changes in learning curves induced by noise, based only on TRIAL 1 data (out of 100 runs)

As the data show, both changes increase in number for noise values above $\sigma = 0.08$.

We observed that in any run, the weights and biases start out at small values. These increase in magnitude during the course of the run until they asymptotically converge upon a final configuration. We theorized that noise's chief mechanism of corruption of the learning curve might well amount to a change in initial conditions. If this is true, then delaying the onset of noise until the weights have had time to increase to a more robust configuration should give rise to a greater apparent noise immunity. This conveniently delayed noise is not so far-fetched; it is almost akin to signal-dependent noise.

Using the same σ values as before, we modified the algorithm for noise introduction. Recall Figure 4. We inserted in "btrain.m" a step that computes the derivative (∂ mse/∂ cycle) of the learning curve. When the middle plateau is reached, ∂ mse/∂ cycle reaches a local maximum, and the noise is introduced. In all, twenty-five sets of initial conditions were used.

In some cases, entrapment in a local minimum in the noiseless case caused the algorithm to never call for the noise in the first place. But in twenty-one cases, the noise did initiate; of these, eight were stuck in the noiseless case.

Figure 7 shows the number of stuck-to-converged and converged-to-stuck occurrences for each noise level. Note that the noise level for significant changes appears to have moved up from $\sigma = 0.08$ to greater than $\sigma = 0.25$, verifying that greater robustness has set in by this point in training.

σ	0.04	0.08	0.12	0.16	0.20	0.25	0.30	0.35	0.40
# s-c	1	1	2	3	0	2	3	4	6
# c-s	0	0	3	2	4	1	4	7	5

Figure 7. Numbers of significant changes in learning curves induced by delayed noise (out of 22 runs)

Is accumulated updating perhaps less sensitive to noise? One might expect this since updates are less frequent. We tested this hypothesis using the same noise values as for pairwise layered updating. With the same metric, we assembled the data table in Figure 8.

σ	0.04	0.08	0.12	0.16	0.20	0.25	0.30	0.35	0.40
# s-c	1	1	2	2	2	2	2	1	1
# c-s	0	0	1	1	4	3	3	4	4

Figure 8. Numbers of significant changes in learning curves induced by noise (out of 15 runs)

As the table shows, the noise shows a significant increase in it effect for σ values of greater than 0.16. Recall that for layered updating, the crucial noise value fell around 0.08. Indeed, accumulated updating is the more robust.

We took similar data for delayed noise introduction. These data are less conclusive. The learning curve for accumulated updating is not characterized by a plateau with an inflection. We had to settle for simply introducing the noise when the mean-square error fell below a certain value. We tried 0.639 on some trial runs, but we do not know that these results can be compared to the runs for delayed noise in pairwise layered updating.

2. Updating methods and initial conditions

For five sets of initial conditions, we ran the $J = 2$ XOR problem with $\eta = 2$, using accumulated updating, pairwise layered updating, and pairwise unlayered updating. The average time to convergence was longest for accumulated updating, but it wasn't consistently the longest for each of the five cases. The average time to convergence was shortest for pairwise layered updating. Generally, the learning curves for accumulated updating were the smoothest; the pairwise learning curves had more inflections, especially the pairwise layered.

We repeated this experiment using Gilbert's method of initializing. This time, the times to convergence varied little between the updating methods. On the whole, this initializing more than halved convergence time.

We found similar results when we repeated both experiments running the $J = 4$ 2-D corners problem with $\eta = 0.4$. Figure 9 shows the targets as a function of the input space for this problem. Layered updating was usually the fastest, and Gilbert's method considerably shortened the time to convergence.

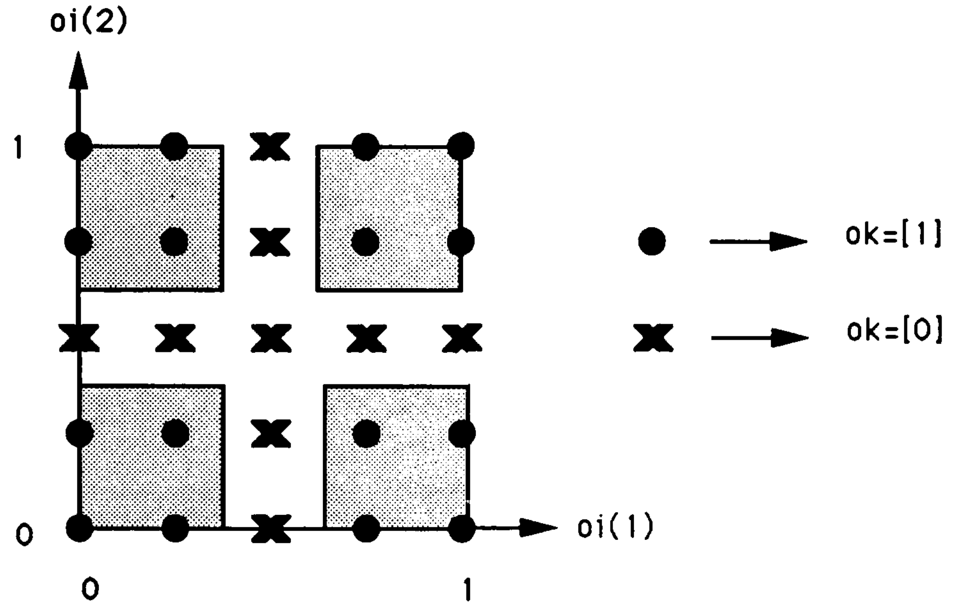

Figure 9. Training data for the 2-D corners problem

D. Summary

The Matlab program modules have enabled us to run simulations of the back propagation algorithm using a simplified additive noise model. In addition, we have learned much about the back propagation model itself. Specifically, the order of steps in the updating procedure, which is borne out by the optical implementation, has much to do with convergence speed and whether or not gradient descent is actually being performed. Although it is the farthest from gradient descent, pairwise layered updating converges the fastest.

One must also choose a suitable learning rate, large enough to speed performance without oscillations through the error landscape. Oscillations and entrapment in local minima can be avoided using the momentum term α, but this does not lend itself well to optics. It is also wise to choose targets away from the asymptotic tails of the threshold function.

Performance is dramatically improved by initializing the weights so that the dividing hyperplanes created by the hidden units cross the center of the input vector space, and normalizing the weight vector magnitudes.

In our simulations using pairwise layered updating on to solve XOR with $J = 2$ and $\eta = 2$, we observed significant degradation with simple additive noise having $\sigma = 0.08$. Greater noise levels ($\sigma = 0.25$) can be allowed if its onset is suitably delayed in the learning period. Accumulated updating is less noise sensitive; with noise always present, degradation becomes significant at $\sigma \geq 0.16$.

IV. PLANS FOR PHASE II

A. Optical architecture for BEP

The simulation of a true optical neural net will incorporate more than simply additive noise applied to the weights and biases after update. It is necessary to take into account noise introduced by sources, modulators, and detectors. This requires designing an architecture capable of handling the whole process, including the forward pass, error back propagation, accumulation of the ΔW's over the epoch, and addition of them at the end. The architecture need not be *optimum*, only plausible.

Our hypothetical architecture uses optics to perform operations of $O(N^2)$, where N is the typical layer size, e.g., I, J or K; other operations may take place in the electrical domain. The inputs and weight/bias matrices are assumed to be implemented as spatial light modulators (SLMs).

Figure 10 shows our multilayer perceptron architecture for a net in which $I = 3$, $J = 4$ and $K = 2$. SLMs are represented as unshaded planar regions; detectors are shown shaded. Thick arrows represent the propagation of information-carrying light; thin arrows refer to signals in the electrical domain. For simplicity, the figure does not include the necessary cylindrical and spherical imaging lenses, nor the switchable birefringent wave plates needed to direct light the proper way using the polarizing beamsplitters. Wherever possible, we have striven to use the same weights in backward as in forward passes. This saves hardware and reduces noise accumulation.

As depicted in Figure 10, the architecture is performing a forward pass. Recall the governing equations presented in Section III. B. 1. The forward passes are vector-matrix multiplies shown in Figure 10 by light gray left-to-right arrows. While the inputs o_i and o_j are unipolar and restricted to the interval [0, 1], the weights and biases are bipolar and can have larger magnitudes. We have tracked the weights and biases through simulations on 2-D and 5-D problems and seen values as large as 15. Since the SLMs do not amplify, we let them express values W_{ji}/a, where a is a suitably chosen constant, say, 15. Also, we have divided the matrix elements into positive and negative subelements. The Net_j and Net_k are also so divided. The subtraction of $Net(-)$ from $Net(+)$ is performed electronically. The thresholding elements incorporate sigmoids which are a steep, thus performing the operation

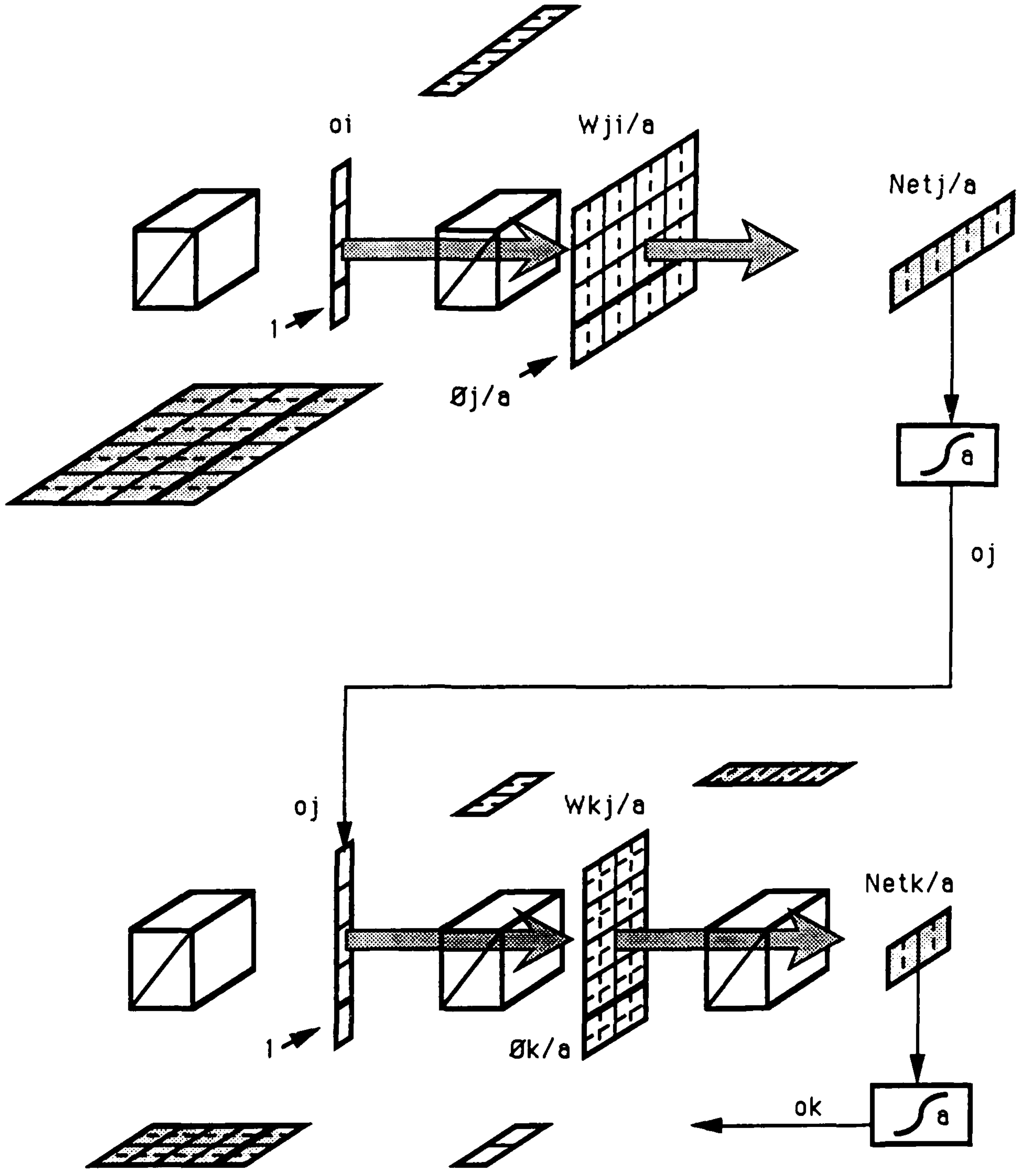

Figure 10. The proposed multilayer perceptron architecture for $I = 3$, $J = 4$ and $K = 2$, performing a forward pass

$$o_j = 1/(1 + \exp(a \times -Net_j/a)) = 1/(1 + \exp(-Net_j))$$

and similarly for o_k.

Figure 11 shows the same architecture, now performing backward propagation of errors. The error term δ_k is computed electronically (bottom of Figure 11). Note that δ_k is bipolar and usually small. In the above simulations, the largest element never exceeded 0.16. To better utilize the full range of the 1-D SLMs, we multiply δ_k by a constant b. Note that $\delta_k b$ is sent to two SLMs, one with + and - subelements, one not. The former SLM is to be used for computing the outer product as in Equation 5 of the governing equations.

The latter SLM is for calculating the vector matrix multiply inside Equation 4. This requires dividing the W_{kj} elements *horizontally* as shown. As depicted by the dark gray arrow in Figure 11, the operation

$$W_{kj}^{T}/a \times \delta_k b$$

occurs in two passes, one for each sign of δ_k. In the first pass, $\delta_k(+)$ is presented; each element of the receiving detector will evaluate two terms, positive and negative. In the second pass, when $\delta_k(-)$ is presented, the formerly positive subelement now receives a negative term

$$(\delta_k b(-) \times W_{kj}^{T}/a(+))$$

---and vice versa. The receiving detector must be "smart" enough to handle this sign interchange.

When this vector matrix multiply is complete, δ_j (from the rest of Equation 4) can be computed.

With δ_k and δ_j computed, all that remains is two outer products (Equations 5 and 6). These use the light paths shown in black arrows. The receiving SLMs perform accumulation over the epoch by time integration. Here the operation multiplies bipolar-by-unipolar, as in the forward passes. At the end ot the epoch, the accumulated ΔW_{ji}

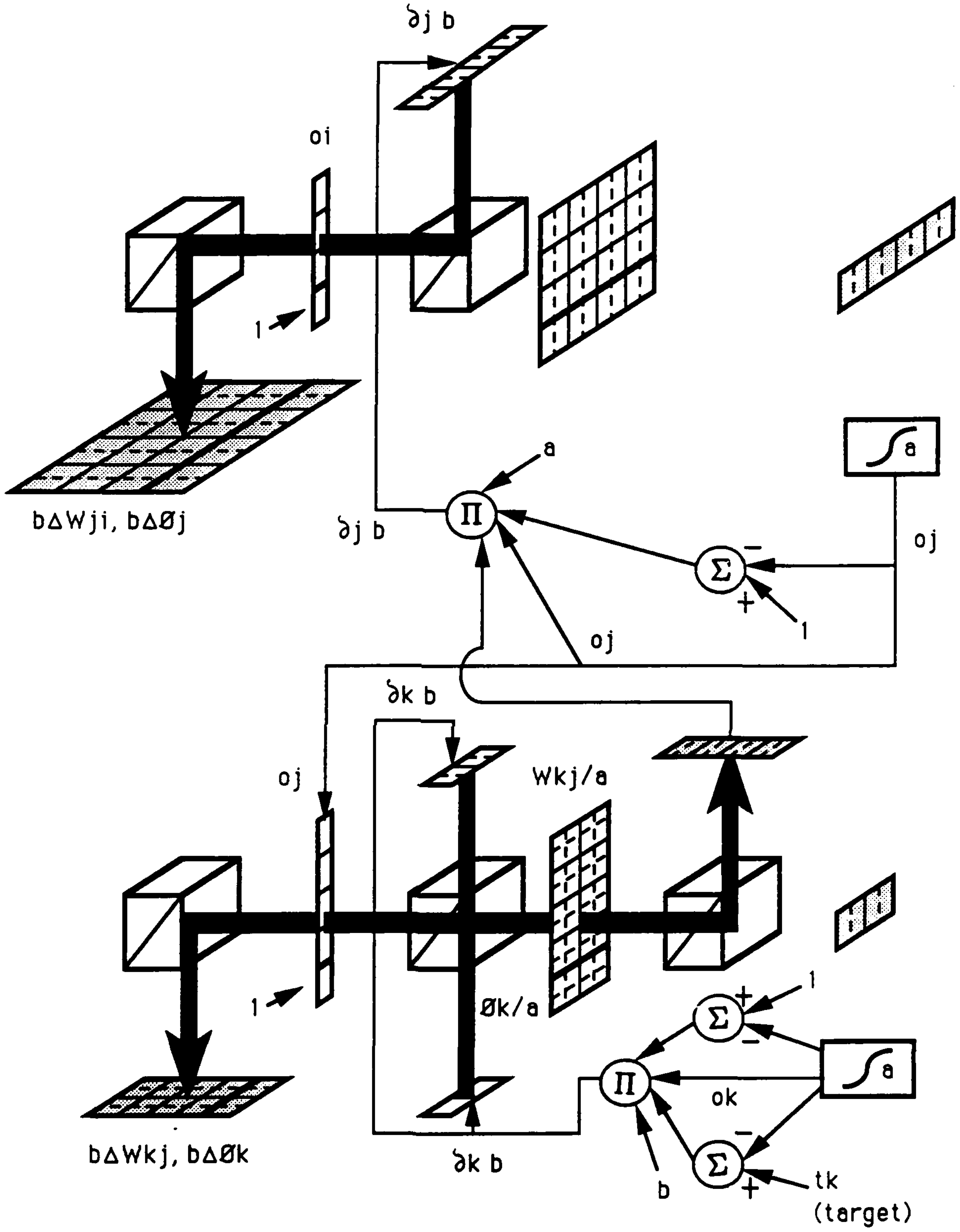

Figure 11. The proposed multilayer perceptron architecture performing backward error propagation

and ΔW_{kj} are to be added to the weight matrices themselves, which are presumably "smart" SLMs with capacitive storage. The light paths for this operation are not shown.

The Phase II simulations will incorporate opto-electronic noise caused by the various transductions in the architecture. In addition to random noise such as shot and thermal noise, the effects of fixed pattern noise, as from nonuniformity and limited contrast ratio, will be included.

B. Exemplar-based systems

The BEP model for a neural net classifier separates the input patterns by a series of hyperplanes. Recall Figure 2, which shows such a hyperplane classifier for a 2-D input pattern. Each of the hyperplanes is associated with a hidden processing element and the orientation and location of the hyperplane is completely specified by the weight vector associated with that processing element. One or more layers of hidden processing elements are needed to form an arbitrary decision boundary via intersections of half spaces.[19] The BEP classifier is characterized by a long training interval in which the training pairs have to be presented as many as several thousand times before the correct weight vectors are identified. This learning procedure also has the tendency to get stuck in a local minimum for the total error function. On the other hand the number of processing elements (and hence the total weight storage requirement) is independent of the number of exemplars, i.e., training input patterns. The BEP learning procedure, thus, makes efficient use of resources.

Another category of neural net classifiers with different tradeoffs is termed exemplar-based classifiers.[21] The weight vectors associated with the processing elements in the first layer simply represent the input patterns (exemplars). The subsequent layers then group outputs from the first layer processing elements and make a classification decision. The subsequent processing differs between k-nearest neighbor classifiers,[22,23] Restricted Coulomb Energy classifiers[24,25] or Radial Basis Function Classifiers.[26,27] The common characteristic of these models is that they have high storage requirements (proportional to the exemplars in the training set) but short training times. The error signals in the training cycle are used mainly to adjust the weights in the subsequent layers or some simple parameters (e.g., the size of the basin of attraction) associated with the processing elements in the first layer.

It appears that the tradeoffs associated the exemplar-based classifiers are much more suited to optical technology that provides large storage capacity but poor computational accuracy. In the second phase we will select one of the exemplar-based classifiers for further study. Optical processors can be restricted to the first or second layer of the neural net performing the computationally intensive operations. High precision analog or digital electronics can be used in the final layers that perform the classification. The comparison between optical implemented BEP classifiers and hybrid opto-electronic exemplar-based classifiers on identical problems and with identical hardware limitations will be performed during the second phase of this contract.

V. REFERENCES

1. T. Kohonen, *Self-Organization and Associative Memory*, New York: Springer-Verlag, 1984.

2. S. Grossberg, *Studies of Mind and Brain*, Boston, MA: Reidel, 1982.

3. M. Caudill and C. Butler, Eds., *Proceedings of the IEEE First International Conference on Neural Networks*, vol. I-IV, San Diego, June 21-24, 1987.

4. M. Takeda and J. W. Goodman, "Neural Networks for Computation: Representations and Programming Complexity," Appl. Opt. **25**, 3033-3046 (1986).

5. A. D. Fisher, W. L. Lippincott, and J. N. Lee, "Optical Implementations of Associative Networks with Versatile Adaptive Learning Capabilities," Appl. Opt. **23**, 5039-5054 (1987).

6. D. Brady, X. G. Gu, and D. Psaltis, "Photorefractive Crystals in Optical Neural Computers," Proc. SPIE **882**, 20 (1988).

7. J. J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," Proc. Natl. Acad. Sci **79**, 2554-2558 (1982).

8. D. Psaltis and N. Farhat, "Optical Information Processing Based on an Associative Memory Model of Neural Nets with Thresholding and Feedback," Opt. Lett. **10**, 98 (1985).

9. Special Issue on Neural Networks, Appl. Opt. **26**, Dec. 1988.

10. M. Minsky and S. Papert, *Perceptrons*, Cambridge, MA: MIT Press, 1969.

11. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation," in *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, D. E. Rumelhart and J. L. McClelland, Eds. Cambridge, MA: MIT Press, 1986, pp. 318-362.

12. B. Widrow and M. E. Hoff, "Adaptive Switching Circuits," *1960 IRE WESCON Convention Record*, New York: IRE, pp. 96-104.

13. R. P. Gorman and T. Sejnowski, "Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets," Neural Networks, **1**, 75-89, (1988). See also IEEE Trans. ASSP **36**, 1135-1140 (1988).

14. G. Bradshaw, R. Fozzard, and L. Ceci, "A Connectionist Expert System that Actually Works," Adv. Neural Inf. Proc. Syst. **1**, 248-251 (1988).

15. K. Wagner and D. Psaltis, "Multilayer Optical Learning Networks," Appl. Opt. **26**, 5061-5076 (1987).

16. M. Kranzdorf, B. J. Bigner, L. Zhang, and K. M. Johnson, "Optical Connectionist Machine with Polarization-based Bipolar Weight Values," Opt. Engg. **28**, 844-848 (1989).

17. A. Von Lehmen, E. G. Paek, P. F. Liao, A. Marrakchi, and J. S. Patel, "Influence of Interconnection Weight Discretization and Noise in an Opto-electronic Neural Network," Opt. Lett. **14**, 928-930 (1989).

18. S. L. Gilbert, "Implementing Artificial Neural Networks in Integrated Circuitry: A Design Proposal for Back-Propagation," Lincoln Laboratory, MIT, Lexington, MA, Technical Report 810, Nov. 1988.

19. R. P. Lippmann, "An Introduction to Computing with Neural Nets," IEEE ASSP Magazine, pp. 4-22, April 1987.

20. D. Psaltis, D. Brady, and K. Wagner, "Adaptive Optical Networks Using Photorefractive Crystals," Appl. Opt. **27**, 1752-1759 (1988).

21. R. P. Lippmann, "Pattern Classification Using Neural Networks," IEEE Communications Magazine, pp. 47-64, November 1989.

22. S. M. Omohundro, "Efficient Algorithms with Neural Network Behavior," Complex Systems **1**, 273-347 (1987).

23. C. Stanfill and D. Waltz, "Toward Memory-based Reasoning," Commun. of the ACM **29**, 1213-1228 (1986).

24. D. L. Reilly, L. N. Cooper, and C. Elbaum, "A Neural Model for Category Learning," Bio. Cybernetics **45**, 35-41 (1982).

25. G. V. Puskorius and L. A. Feldkamp, "A Characterization and Emulation of the Nestor Learning System," Ford Motor Company, Dearborn, MI, Technical Report SR-88-126, December 1988.

26. D. S. Broomhead and D. Lowe, "Radial Basis Functions, Multi-Variable Functional Interpolation and Adaptive Networks," Royal Signals and Radar Establishment, Malvern, Worcester, Great Britain, Memorandum 4148, March 1988.

27. S. Renals and R. Rohwer, "Phoneme Classification Experiments using Radial Basis Functions," *Proc. Int'l Joint Conf. on Neural Networks*, Washington, DC: IEEE, pp. I.461-I.467, June 1989.

VI. KEY PERSONNEL

A. Dr. Michael W. Haney

Dr. Haney is the manager for optical computing technology at BDM International, Inc. and the program manager on this contract. He received his Ph.D. in electrical engineering in 1986 from the California Institute of Technology. His dissertation was: "Acousto-optical Time-and-Space Integrating Processors for Real-Time Synthetic Aperture Radar Imaging."

B. Dr. Ravindra A. Athale

Dr. Athale, former manager for optical computing technology at BDM, is currently an associate professor of electrical engineering at George Mason University and continues his involvement in this program as a regular consultant. He received his Ph.D. in electrical engineering in 1980 from the University of California at San Diego. The title of his dissertation was, "Studies in Digital Optical Computing."

C. Mr. James J. Levy

Mr. Levy is an associate staff member for optical computing technology at BDM International, Inc. He received his M.S. in electrical engineering in 1987 from the University of Delaware. His thesis title was, "Grating Couplers for Optical Waveguides."

VII. CONFERENCE PRESENTATIONS

Mr. James J. Levy will be presenting a paper at the Annual Meeting of the Optical Society of America, to be held November 4-9, 1990 in Boston. Below are given the abstract and summary, which will appear in the advance program and technical digest, respectively.

Abstract

Computer simulations reveal the effects of noise in optical weight matrix elements, updated by back propagation, upon the learning curve characteristics.

Summary

The back propagation algorithm[1] has become increasingly popular in the neural net research community. Various optical implementations have been proposed, with the hope of increased performance via the parallelism of optics. Realistic models of opto-electronic implementations must include the effects of noise. While noise often "anneals," increasing the convergence rate, excessive noise can effectively transform any updating algorithm into a random search among weight configurations. For the purpose of evaluating the effects of component noise on the performance of the back propagation algorithm, we have developed a simulation program which allows insertion of noise terms wherever appropriate. The program introduces noise processes into the weight updating process during the learning phase. The learning curve is defined as the mean-square error vs. iteration number; our analysis emphasizes examination of learning curves rather than simply probability of convergence. In this paper we describe the behavior and noise tolerance of back propagation as related to hidden layer size, learning rate, and initial conditions.

1. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in Parallel Distributed Processing, vol. 1, D. E. Rumelhart and J. L. McClelland, Eds. Cambridge, MA: M.I.T. Press, 1986, pp. 318-362.

APPENDIX A. PROGRAM LISTINGS

bframe.m

```
% bframe script
disp('This lets you run binit and btrain by having')
disp('you put in all the necessary variables beforehand.')
J=input('Enter J:  ');
icseed=input('Enter icseed:  ');
spread=input('Enter the initial condition spread:  ');
smartinit=input('Enter 1 to initialize by method of Gilbert:  ');
quan=input('Enter 0 to not quantize, 1 to quantize:  ');
if quan==1
   ss=input('   Enter the step size:  ');
   hr=input('   Enter the upper (and -lower) bound:  ');
end
eta=input('Enter eta:  ');
N=input('Enter N:  ');
DN=input('Enter DN:  ');
msetol=input('Enter the mse below which training stops:  ');
% We will keep quan (and ss and hr) the same in training
% as during binit.
inject=input('Enter 0 for no noise; 1, uniform; 2, Gaussian:  ');
if inject==1
   nspread=input('   Enter the noise spread:  ');
elseif inject==2
   var=input('   Enter the variance:  ');
end
mix=input('Enter 1 to mix up presentation order:  ');
if ((mix==1)|(inject~=0))
   trseed=input('Enter trseed:  ');
end
```

binit.m

```
% binit script
% The following variables must be known:
% J, icseed, spread, quan (and ss & hr if
% quan=1), and eta (alpha is not used).
load bpin
load bpout
I=ncols(bpin)
K=ncols(bpout)
R=nrows(bpout)
if exist('keepWji')
   Wji=keepWji; THETAj=keepTHETAj;
   Wkj=keepWkj; THETAk=keepTHETAk;
else
   % Initialize the weights and biases
   % random numbers about zero center
   rand('uniform')
   rand('seed',icseed)
```

```
  rand(1,3); % to exercise generator
  Wji=(rand(I,J)-1 ./2)*spread
  THETAj=(rand(1,J)-1 ./2)*spread
  Wkj=(rand(J,K)-1 ./2)*spread;
  THETAk=(rand(1,K)-1 ./2)*spread;
  if smartinit==1
    % the Wji
    % numerator is predet. magnitude of weights vector
    % which's normal to dividing hyperplane (line for I=2)
    x=1 ./sqrt(sum(Wji.^2))
    Wji=(ones(I,1)*x.^2).*Wji
    % the THETAj
    THETAj=-0.5*sum(Wji)
    % the Wkj
    Wkj=(1/J)*ones(J,K)
  end
  keepWji=Wji; keepTHETAj=THETAj;
  keepWkj=Wkj; keepTHETAk=THETAk;
  if quan==1
    levels=2*hr/ss+1
    Wji=discret(Wji,ss,hr);
    THETAj=discret(THETAj,ss,hr);
    Wkj=discret(Wkj,ss,hr);
    THETAk=discret(THETAk,ss,hr);
  end
end
% Preparation for mse calculation
% the zeroth-iteration column for mse
n=0;
% MSE CALCULATION
% Forward Pass
NETj=[bpin,ones(R,1)]*[Wji;THETAj]
Oj=1 ./(1+exp(-NETj))
NETk=[Oj,ones(R,1)]*[Wkj;THETAk]
Ok=1 ./(1+exp(-NETk))
detmse(1)=sum(meanc((Ok-bpout).^2));
ldetmse(1)=detmse(1); % literally, less detailed mse
                      % this first one's not an average
noff(1)=sum(sum((abs(Ok-bpout)>0.4))); % assumes {0.1, 0.9}
                                       % targets
```

<u>bctrain.m</u>

```
% bctrain script
% Accumulates weight change over epoch, THEN updates...
% The following variables, in addition to those created
% by binit, must be known:
% N, DN, msetol,
% quan (anew for training) (and ss & hr if quan=1),
% inject (and nsprd for uniform if inject=1 and var for
% normal if inject=2), trseed (if inject~=0),
if quan==1
  levels=2*hr/ss+1
  Wji=discret(Wji,ss,hr);
  THETAj=discret(THETAj,ss,hr);
  Wkj=discret(Wkj,ss,hr);
  THETAk=discret(THETAk,ss,hr);
end
if inject==1
  rand('uniform') % just in case it isn't
end
if inject==2
  rand('normal')
end
if ((n==0)&(inject~=0))
  rand('seed',trseed)
end
stop=n+N;
while n<stop
  n=n+1;
  DWkj=zeros(J,K);
  DTHETAk=zeros(1,K);
  DWji=zeros(I,J);
  DTHETAj=zeros(1,J);
  for r=1:R
    % presentation order irrelevant
    % Forward Pass
    NETj(r,:)=[bpin(r,:),1]*[Wji;THETAj];
    Oj(r,:)=1 ./(1+exp(-NETj(r,:)));
    NETk(r,:)=[Oj(r,:),1]*[Wkj;THETAk];
    Ok(r,:)=1 ./(1+exp(-NETk(r,:)));
    % Error back propagation
    Dk=(bpout(r,:)-Ok(r,:)).*Ok(r,:).*(1-Ok(r,:));
    DWkj=DWkj+eta*Oj(r,:)'*Dk;
    DTHETAk=DTHETAk+eta*Dk;
    Dj=Oj(r,:).*(1-Oj(r,:)).*(Dk*Wkj');
    DWji=DWji+eta*bpin(r,:)'*Dj;
    DTHETAj=DTHETAj+eta*Dj;
  end
  % first, the connections to the K output neurons
  Wkj=Wkj+DWkj;
  if quan==1, Wkj=discret(Wkj,ss,hr); end
  if inject==1, Wkj=addnoise(Wkj,nsprd); end
  if inject==2, Wkj=addgauss(Wkj,var); end
  THETAk=THETAk+DTHETAk;
  if quan==1, THETAk=discret(THETAk,ss,hr); end
  if inject==1, THETAk=addnoise(THETAk,nsprd); end
  if inject==2, THETAk=addgauss(THETAk,var); end
  % then, those to the J neurons in the hidden layer
  Wji=Wji+DWji;
  if quan==1, Wji=discret(Wji,ss,hr); end
  if inject==1, Wji=addnoise(Wji,nsprd); end
  if inject==2, Wji=addgauss(Wji,var); end
  THETAj=THETAj+DTHETAj;
  if quan==1, THETAj=discret(THETAj,ss,hr); end
  if inject==1, THETAj=addnoise(THETAj,nsprd); end
  if inject==2, THETAj=addgauss(THETAj,var); end
  % Forward Pass
  NETj=[bpin,ones(R,1)]*[Wji;THETAj];
  Oj=1 ./(1+exp(-NETj));
  NETk=[Oj,ones(R,1)]*[Wkj;THETAk];
  Ok=1 ./(1+exp(-NETk));
  n
  detmse(n-DN*fix((n-1)/DN))=sum(meanc((Ok-bpout).^2));
  if rem(n,DN)==0
    ldetmse(n/DN+1)=mean(detmse);
    disp('convergence check')
    % based on mse over r, not individuals
    if ldetmse(n/DN+1)<=msetol
      disp('less detailed mse within tolerance')
      break
    end
    noff(n/DN+1)=sum(sum((abs(Ok-bpout)>0.4)));
  end
end
```

discret.m

```
function db=discret(in,stepsize,hr)
% out=discretize(n,stepsize,hr)
%
% This function quantizes the vector n to one of
% range/stepsize + 1 levels equally spaced between
% +hr (half the range) and -hr.
% discretize
in=in/stepsize;
in=round(in);
in=stepsize*in;
% bind
db=-(in<=-hr).*hr+(abs(in)<hr).*in+(in>=hr).*hr;
```

addnoise.m

```
function ny=addnoise(in,spread)
% out=addnoise(in,spread)
%
% This function adds small random numbers
% (between spread/2 and -spread/2) to each of
% the elements in in.
[nr,nc]=size(in);
ny=in+(rand(nr,nc)-1 ./2)*spread;
```

addgauss.m

```
function ny=addgauss(in,var)
% out=addgauss(in,spread)
%
% This function adds small random numbers
% with a normal probability distribution
% to each of the elements in in.
% NOTE: It is ASSUMED here that the user has
% already switched the rand mode to 'normal';
% the default is uniform.
[nr,nc]=size(in);
ny=in+var*rand(nr,nc);
```

convtoi.m

```
function out=convtoi(a,v)
% This function takes a vector a and
% returns a matrix of dimensions ncols(a)/v by v.
a=a(:)';
out=zeros(v,ncols(a)/v);
out(:)=a;
out=out';
```

convtov.m

```
function out=convtov(a)
% This function takes a matrix a and
% returns a row vector of dimensions 1 by
% nrows(a)*ncols(a)
a=a';
out=a(:)';
```

meanc.m

```
function s=meanc(in)
% out=meanc(in)
%
% This function returns a column vector with the mean of
% each row of in. This includes the case where in has but
% one column (i.e., it won't then return the scalar).
% So it's not strictly the transpose of mean.

if ncols(in)==1
  s=in;
else
  s=(mean(in'))';
end
```

sumc.m

```
function s=sumc(in)
% out=sumc(in)
%
% This function returns a column vector with the sum of
% each row of in. This includes the case where in has but
% one column (i.e., it won't then return the scalar).
% So it's not strictly the transpose of sum.

if ncols(in)==1
  s=in;
else
  s=(sum(in'))';
end
```