A Self-Organizing Network for Computing *A Posteriori* Conditional Class Probability

George W. Rogers, Jeffrey Solka, D. Stephen Malyevac, and Carey E. Priebe

***Abstract*—This paper describes a neural network architecture whose goal is the computation of *a posteriori* conditional class probabilities for input vectors that belong to one of two input classes. The network architecture has been designed to adaptively produce Voronoi tessellation partitions of the input vectors in R^n based on the Euclidean distance metric, without regard to the actual *a priori* class probabilities of the input vectors. These prior probabilities are then used by the network to adaptively compute the *a posteriori* conditional class probability for the two classes for each tessellation partition. The network presented is thus a connectionist model for vector quantization clustering and includes the process of automatic node creation necessary for many unsupervised learning applications.**

I. Introduction

Discriminant analysis is a major focus of the general field of pattern recognition, and specifically of current neural network research. In particular, multilayer perceptrons [1], [2] and other neural network paradigms [3]–[8] have been developed and used for probabilistic classification and discrimination. Such neural networks are typically used in one of two ways. One use is to construct discriminant surfaces in some (higher dimensional) space in which the input data is represented and thereby form decision regions or output classes. It has been shown [9], [10] that such networks approximate the posterior probabilities for the output classes. One problem with this approach is that it makes it difficult to change the loss function associated with the discrimination problem after learning. In particular, it can be difficult to dynamically adjust the probability of misclassification.

The other primary use of networks in discrimination is to estimate the values of a continuous function such as a density. In this estimation case the data often occurs as discrete samples of continuous distributions. In this case, the data consists of input vectors with associated class memberships. It is often the case that no linear discriminant will separate the classes in the input space. The classical technique for dealing with this problem is to recover an approximation to the original continuous distribution and is depicted in Figs. 1–4 for a one-dimensional input space. Fig. 1 depicts sampling results for two populations in one dimension. Each vertical bar corresponds to a pattern of that class appearing with a feature value corresponding to its x-coordinate. Fig. 2 depicts the results of counting all the patterns of each class within a number of

Manuscript received September 8, 1991; revised May 26, 1992 and September 20, 1992.
The authors are with the Naval Surface Warfare Center, Dahlgren Division, Dahlgren, VA 22448.
IEEE Log Number 9207562.

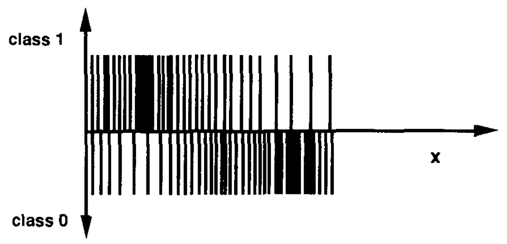


Fig. 1. Example training patterns as a function of observable x for a two class problem.

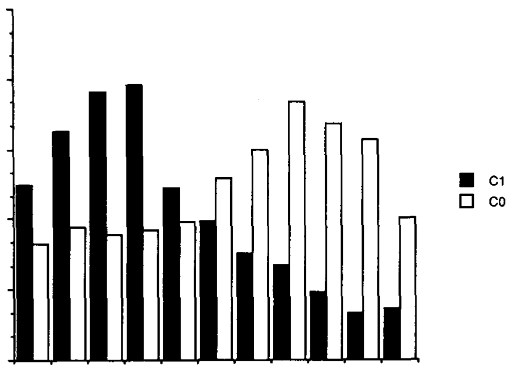


Fig. 2. Example joint probability histogram for a two-class problem.

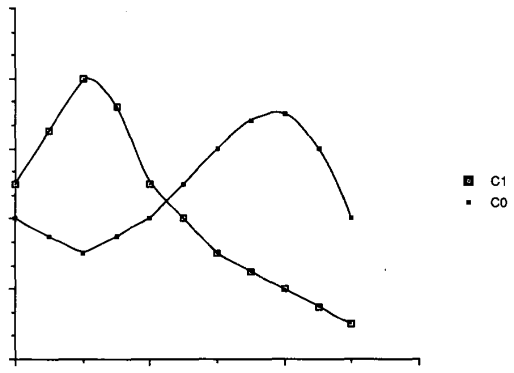


Fig. 3. Example joint probability distributions for a two-class problem.

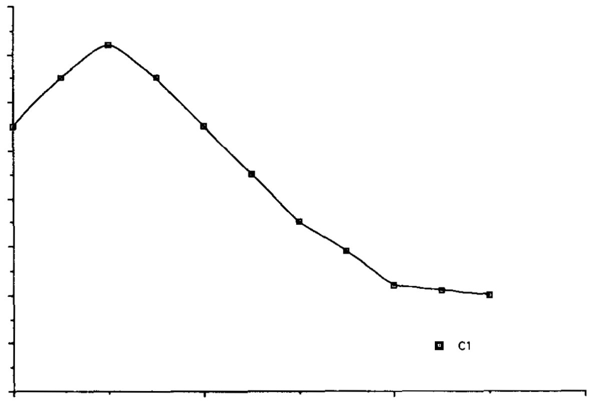


Fig. 4. Example posterior probability plot for a two-class problem.

small regions or boxes. This results in a pair of histograms that are approximations to the continuous distribution functions shown in Fig. 3. The continuous distribution functions are joint probability distributions normalized with respect to the entire set of observations. Discrimination can then be performed based on these estimated densities. A theoretical basis for using density estimates in discrimination is the asymptotic optimality of the procedure under reasonable conditions [11].

The usual problem encountered in developing training sets for estimation networks is that of finding a grid of input vectors spanning the region of interest of the input space with conditional probability values used as the answers for the back propagation supervised learning. These conditional probability values correspond to the probability of class membership given the input vector. The conditional class probability distributions for the example are depicted in Fig. 4.

This paper describes a serial algorithm and equivalent neural network architecture that takes feature vectors and class membership as input and computes the *a posteriori* conditional class probability distributions depicted in Fig. 4. The architecture is an unsupervised, self-organizing one that has proven useful in practice.

The paper is arranged as follows. In Section II, background in vector quantization is presented, followed by a review of conditional probability in Section III. Section IV presents a serial algorithm that is useful on serial or coarsely parallel digital computers and is equivalent to the neural network described in the following sections. Section V develops in detail the building blocks that go into the network, including the various node or neuron types. Section VI presents our network architecture in detail. An analysis of the network for several two-dimensional example problems, including comparison with a classical box counting approach, is presented in Section VII. We conclude with a discussion of the results presented.

II. Vector Quantization

Vector quantization can be defined as a mapping M from an n-dimensional vector space R^n into a finite subset V of R^n

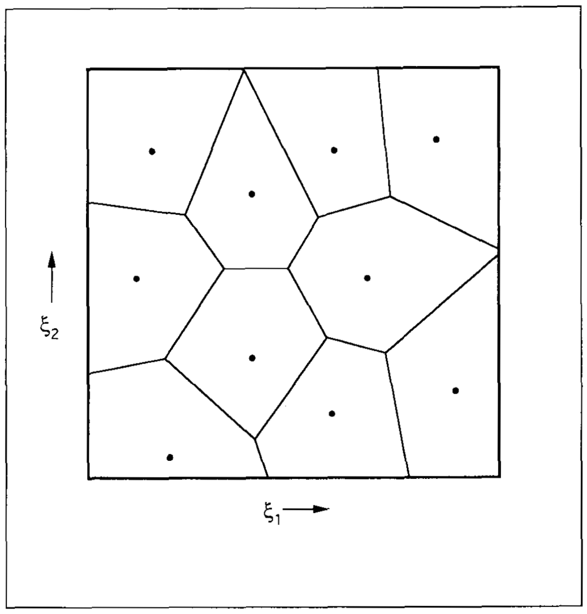


Fig. 5. Voronoi tessellation partitions in a two-dimensional ($x1$, $x2$) pattern space about a set of reference vectors. The reference vectors are represented as points. All vectors in the same partition have the same reference vector as a nearest neighbor and are placed in that reference vector's "class."

$$M: R^n \rightarrow V \tag{1.1}$$

where $V = \{v_i: i = 1, 2, \cdots, N\}$ is the set of N quantization vectors in V defined by the mapping M. Each quantization vector v_i is often called a codeword and V the codebook for the vector quantization scheme [12], [13]. We will follow the convention of calling v_i a reference vector or exemplar and V the set of reference vectors.

A useful concept for visualizing the effects of vector quantization in both classical pattern recognition and neural networks is Voronoi tessellation [14]. Fig. 5 gives an example of the Voronoi tessellation that results for the reference vectors or codewords illustrated in R^2 with a Euclidian distance metric. The effect is to partition the vector space into a number of quantization regions, bordered by hyperplanes (line segments in two dimensions) that are defined as the set of points that are equidistant from the two nearest reference vectors. The tessellation partitions may also be defined in terms of other metrics. In particular, the method of normal mixture models [15] can be considered a generalization of these ideas using Mahalonobis distance and the associated Gaussian distributions. This idea will be explored further in Section IV.

The usual use of this approach is to match or classify an input vector with one of the reference vectors or codewords. Once this has been done, all the characteristics of the reference vector are ascribed to the input vector, that is, the input vector is replaced with the reference vector in any subsequent computation or use.

The procedure presented below, in both its serial and connectionist manifestations, can be considered as a method for adaptively developing Voronoi partitions and using these partitions probabilistically in assigning class membership to unknown input vectors.

III. *A posteriori* Conditional Class Membership Probability

The *a posteriori* conditional class membership probability $P(c_1|x)$ can be written in terms of probability distributions involving the continuous variable x as

$$P(c_1|x) = p(x|c_1)P(c_1)/p(x) \tag{1}$$

where $P(c_1)$ is the *a priori* probability for membership in class c_1, $p(x)$ is the probability density function for the continuous variable x, and $p(x|c_1)$ is the probability density function for x, given that x belongs to the class c_1. This is just a statement of Bayes' relation for distributions. The probability density function $p(x)$ can be written

$$p(x) = p(x|c_1)P(c_1) + p(x|c_2)P(c_2) \tag{2}$$

which lets us write $P(c_1|x)$ as

$$P(c_1|x) = p(x|c_1)P(c_1)/[p(x|c_1)P(c_1) + p(x|c_2)P(c_2)] \tag{3}$$

where now both $p(x|c_1)$ and $p(x|c_2)$ can be independently normalized as

$$\int p(x|c_i)\,dx = 1. \tag{4}$$

This result is useful for both understanding the operation of the neural network that builds an estimate of $P(c_1|x)$ and in generating test cases for the network. Equation (2.3) can be written equivalently as

$$P(c_1|x) = p(x, c_1)/[p(x, c_1) + p(x, c_2)]. \tag{5}$$

Here $p(x, c_i)$ is the joint probability density that the pattern is x and that its class membership is c_i. From (5) we see that $P(c_1|x)$ can be interpreted as a fractional joint probability density.

IV. A Serial Implementation for the Self-Organizing Process

The serial algorithm for digital implementations can be logically broken into two parts; a training mode, in which the Voronoi partitions described in Section I are developed, and an exercise mode where tessellation learning has been disabled. In each case, the input pattern is assumed to be of the same format as in the previous section. The training mode algorithm is given below, followed by the exercise mode algorithm. The input vector is assumed to be encoded so that x_0 is the binary valued class membership of the input vector while x_i is the ith component of the n-dimensional input vector that is to be classified based on the Euclidian metric. The components of the clusters centers or exemplars are denoted by $b_{ji}(n_j)$, where $b_{ji}(n_j)$ denotes the ith component of exemplar j when n_j patterns have been clustered with j. The output or class membership fractions for each exemplar are denoted by $z_j(n_j)$ where n_j patterns have been clustered with j.

A. *Training Mode*

1) Present the new pattern $\{x_i\}$ to the program.

2) Compute the Euclidean distance between $\{x_i;\ i = 1, \cdots, n\}$ and each stored exemplar or reference vector $\{b_{ji}\}$ by

$$y_j = \sum (b_{ji} - x_i)^2.$$

3) Find the minimum y_j. This is the closest exemplar.

4) Verify that $\{x_i\}$ truly belongs to cluster j by testing to see of $y_j < \rho$ where ρ is a user-defined vigilance parameter. If $y_p < \rho$, go to step 5), otherwise go to step 6).

5) Update b_{ji} for node j by the following update rule:

$$b_{ji}(n_j + 1) = b_{ji}(n_j)[n_j/(n_j + 1)] + x_i[(1/(n_j + 1)]$$

where n_j is the number of patterns previously clustered with exemplar j. Update the output value z_j for node j by the following update rule:

$$z_j(n_j + 1) = z_j(n_j)[n_j/(n_j + 1)] + x_0[1/(n_j + 1)].$$

Start over at 1).

6) Create a new exemplar node (reference vector) with $n_j = 0$

$$b_{ji}(1) = x_i$$

and

$$z_j(1) = x_0.$$

Start over at 1).

B. *Exercise Mode*

The exercise or static mode algorithm is as follows.

1) Present the new pattern $\{x_i\}$ to the program.

2) Compute the Euclidian distance between $\{x_i;\ i = 1, \cdots, n\}$ and each stored exemplar or reference vector $\{b_{ji}\}$ by

$$y_j = \sum (b_{ji} - x_i)^2.$$

3) Find the minimum y_j. This is the closest exemplar.

4) Update the output value z_j for node j by the following update rule:

$$z_j(1) = x_0, \qquad \text{for } n_j = 0$$

and

$$z_j(n_j + 1) = z_j(n_j)[n_j/(n_j + 1)] + x_0[1/(n_j + 1)]$$

where n_j is the number of patterns previously clustered with exemplar j. Start over at 1).

C. *Discussion*

In the training mode, the clusters adaptively change the locations of their centers while the number of (active) clusters is increased in a dynamic manner based on the results of a vigilance test. The recursive update formulas used give mean values based on the observations clustered with each individual cluster center. This results in a procedure that is suitable for unsupervised clustering as well as *a posteriori* estimation. In the former case, the centers will migrate toward local cluster centers. In the case where the observations are drawn from continuous distributions, we have observed a diffusion phenomena.

For continuous distributions with a well-defined gradient, there is a marked tendency for the cluster centers to slowly migrate from regions of lower density toward the higher density regions. This leads to a greater concentration of cluster centers at the regions of higher density and the occasional creation of new centers at the region of lowest density. In practice, the rate of creation of new clusters falls off very rapidly. This diffusion phenomena has both desirable and undesirable qualities. First, the tendency to make the cluster density a function of the probability density has the desirable effect of providing higher resolution of the vector quantization where the observation density is higher and estimation accuracy can be maintained. The undesirable qualities consist of both the small resultant estimation bias that is introduced along with the failure of the algorithm to necessarily reach a maximum number of clusters in the limit.

The exercise mode keeps the number of clusters fixed as well as the cluster centers fixed. It does not employ the vigilance test. It can either update the class membership fraction for the clusters as the patterns are presented as a restricted training mode, or hold them fixed for pure testing. In the restricted training mode, the diffusion problems associated with the training mode can be circumvented.

The optimal method of using the algorithm consists of the following steps. First, train on a data set with the network in training mode until the only node creation is due to the diffusion phenomena. Then, using the cluster centers from the first step, repeat the training in the restricted training mode where only the class membership fractions are updated. This results in an stable estimate based on a fixed number of cluster or quantization centers. Additionally, it takes advantage of the diffusion phenomena present in the first step to achieve higher resolution in the regions where there are adequate observations to support it. Finally, observations of unknown class can be processed with no updating to produce *a posteriori* estimates. Analysis of convergence properties based on simulation results is presented in Section VII. This procedure behaves in much the same way as the adaptive resonance theory [16].

The learning rules described should be compared to a maximum likelihood development [15], [17]. Considering the extension to Gaussian-based tessellation leads to the development of the recognizably similar maximum likelihood update procedures of mixture models, and the data-driven increasing of the total number of partitions is analogous to the method of adaptive mixtures [17].

V. Neural Network Building Blocks

A. *Basic Neural Types*

There are three basic artificial neuron types that are used as building blocks in the network described below. The simplest is the most familiar "sigma" neuron that forms a weighted sum of its inputs x_i and pipes the result through a transfer function that varies with the application

$$y = f\left(\sum w_i x_i + o\right) \tag{6}$$

where $f(\cdots)$ is the transfer function, y is the output, o is an offset, and the sum is over i. Common transfer functions in the neural network literature include threshold step functions, sigmoidal functions, linear mappings, and kernel functions. A generalization of this is the "sigma-pi" neuron [2] that involves not only a weighted sum but products as well:

$$y = f\left(\sum w_{ik} x_i x_k + \sum w_i x_i + o\right). \tag{7}$$

One highly useful variant of (7) is the simplified version: $y = x_i x_k$, where it is seen that the input x_i functions to to gate the analog value x_k. Thus we have a simple analog pass function neuron as a special case of the more general "sigma-pi" neuron. The third neuron type applies when there is dynamic feedback between neurons in the same layer or of a neuron to itself [18]

$$dx_k/dt = -\alpha\left[x_k - f\left(\sum w_i x_i\right)\right]. \tag{8}$$

B. *Specialized Neuron Types*

The primary purpose of the first layer is to form Euclidean distances between the active nodes and the input feature vector. Each node in this layer forms the Euclidean distance based on the feature vectors inputs and its stored offset weights w_{ji}:

$$y_{1j}(t_{k+1}) = y_{2j}(t_k) \sum [w_{ji}(t_k) - x_i(t_k)]^2 + \{1 - y_{2j}(t_k)\}\, y_1^{\max} \tag{9}$$

where $y_{1j}(t_{k+1})$ is the output produced by node j in this layer at time t_{k+1} and $y_1^{\max}$ is a constant with $y_1^{\max} > \rho$. This can be viewed as either a relatively complex artificial neuron or as a composite processor made up of a number of components. Fig. 6 shows a composite processor that computes the Euclidean distance portion of the computation contained in (10)–(12) below. In this latter case, the first sublayer merely takes differences for each input component:

$$y_{1^{s1}ji} = w_{ji}(t_k) - x_i(t_k), \qquad j = 1, N,\ i = 1, n \tag{10}$$

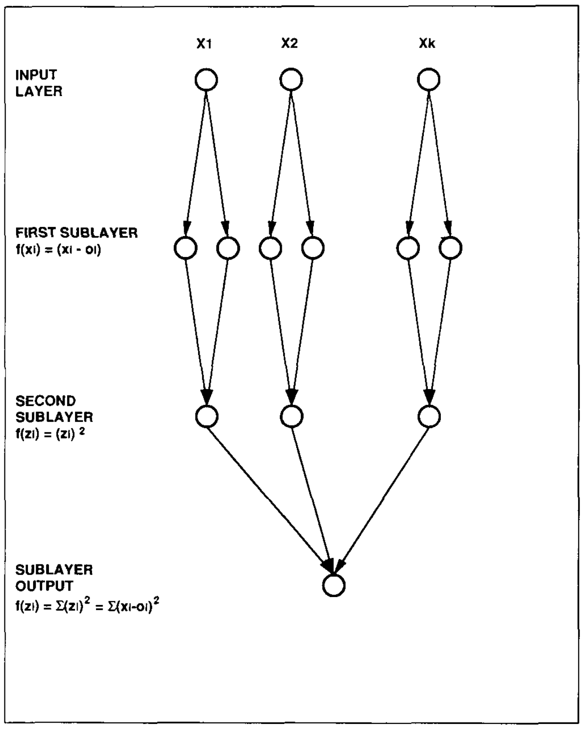

X1
X2
Xk
INPUT LAYER
FIRST SUBLAYER f(xi) = (xi - oi)
SECOND SUBLAYER f(zi) = (zi)²
SUBLAYER OUTPUT f(zi) = Σ(zi)² = Σ(xi-oi)²

Fig. 6. Architecture for the computation of Euclidian distance based on sigma and pi units.

while the second sublayer gates (multiplies) the first sublayer results with themselves:

$$y_{1^{s2}ji} = (y_{1^{s1}ji})^2. \tag{11}$$

The third sublayer forms the sum

$$y_{1^{s3}j} = \sum (y_{1^{s2}ji}) \tag{12}$$

while the final value is given by the product and sum

$$y_{1j}(t_{k+1}) = y_{2j}(t_k)\, y_{1^{s3}j} + \{1 - y_{2j}(t_k)\}\, y_1^{\max}. \tag{13}$$

While this expression shows little superficial resemblance to the "sigma-pi" neuron expression (7), a closer inspection reveals that it is of the form

$$y = w_{12}x_1x_2 + w_2x_2 + o$$

which is just the form of a simple higher order neuron with a linear transfer function.

The primary result of this exercise is simply that a nonbiological computation such as Euclidean distance in R^n can be broken down into a series of simple steps that are appropriate for biologically inspired artificial neurons. It is necessary to include not only "sigma"-type neurons but also "sigma-pi"-type neurons, as discussed in Rumelhart and McClelland [2]. Although many of the connections weights have the value one, the computation is embodied in the weight or connection structure along with the transfer functions (in this case linear), making this a "connectionist" or, equivalently a "parallel distributed processing" formulation as opposed to the equivalent algorithmic description of Section IV. The point of greatest departure from more conventional "connectionist" architectures such as the multilayer perceptron [2], is that the weights are fixed by virtue of the *a priori* structure based on theory and only the offsets are subject to learning.

The second-layer nodes are simply designed to flip from their initial values of 0 to their final permanent values of 1 when an inhibitory signal from the third layer first ceases:

$$y_{2j^{\text{new}}} = \theta\{\, y_{2j^{\text{old}}} - y_{3j^{\text{old}}} + (1 - \epsilon)\} \tag{14}$$

where θ is the familiar Heaviside step function, ϵ is a small positive constant, $y_{2j^{\text{new}}}$ is the new jth second-layer value, and $y_{2j^{\text{old}}}$ and $y_{3j^{\text{old}}}$ are the former second- and third-layer values, respectively.

The third-layer nodes are connected in a Maxnet [19] architecture, the output of which determines which of the third layer nodes cease sending out an inhibitory signal to the second layer. Maxnet architectures can be formulated either in terms of dynamically interacting neurons or as a purely feed-forward system [20]. The function of the module is to yield a value of one for the node that starts with the largest input value and a value of zero for all others. The fourth-layer nodes are also based on the Maxnet architecture, while the fifth-, activation-, and output-layer nodes are based on the Sigma-Pi unit described above. The details and function of each are provided in the next section.

VI. Neural Network Description

Overview: Before getting into the fine details of the network structure and operation, it is instructive to consider the basic structure of the network and the philosophy of its operation. This operation is depicted in Figs. 7 and 8, for adaptive computation and fixed computation, respectively. The connections for the first node in each layer are shown in detail. Connections for the other nodes in each layer are not generally shown. The input feature vector components and class membership index are fed to the network input layer. The first layer then computes the Euclidean distance between the feature vector and the reference vector stored at each active node. Whether a first-layer node is active or not is based on the value of its corresponding second-layer node. If a first-layer node is inactive, a maximum value is output. The active nodes in the fourth layer than compete to find the minimum Euclidean distance for the current feature vector. The fifth layer gates the Euclidean distances from layer one with the results of the competitive fourth layer so that only the winning distance survives to be passed to both the activation node and the output layer. The activation node tests the Euclidean distance against a vigilance parameter. If the distance is less than the vigilance parameter value, the activation node sends no activation signal, which allows the output layer to update its estimates of the *a posteriori*

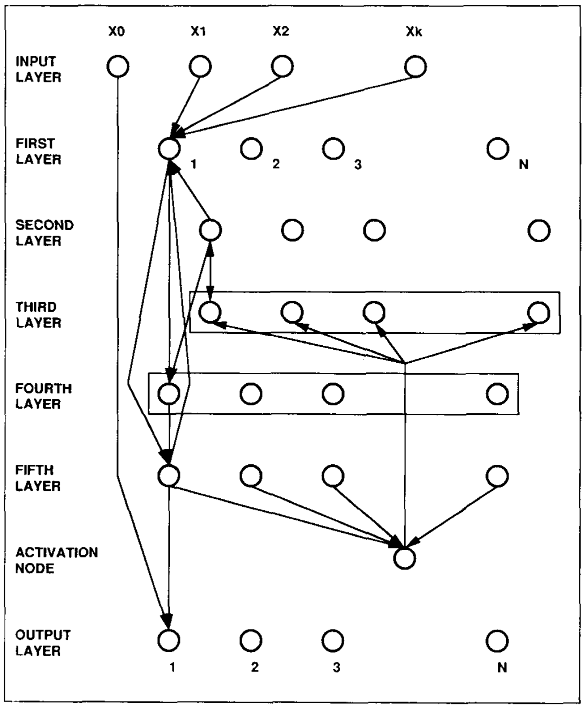


Fig. 7. Network architecture for adaptive computation of the conditional class probability function.

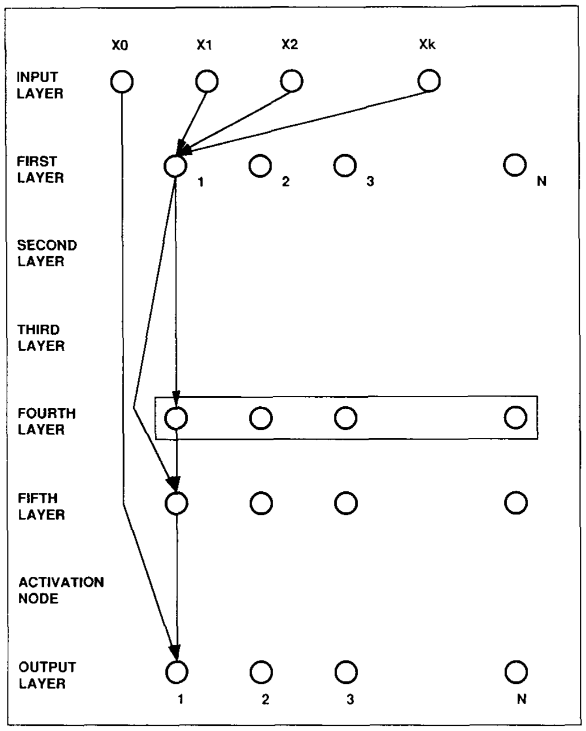


Fig. 8. Network architecture for fixed computation of the conditional class probability function.

conditional class probability. If the minimum Euclidean distance exceeds the vigilance parameter, the activation node sends a signal back up to the third layer, which triggers a competitive process to determine which previously uncommitted node will be committed. The winning node causes the state of its corresponding node in the second layer to flip to the active state. The first-layer computation is now repeated, followed by the fourth through output layers. If there is a new node free to be committed, the activation node will remain inactive the second time through, so that the output layer will update its estimates. If all nodes have been committed, no update will occur for that input feature vector. In any event, after the second cycle, the network is ready for a new input vector.

The detailed neural network description below is broken down into description of the individual layers and the processing done in each layer. The network is assumed to be clocked so that each layer operates synchronously with the exception of the Maxnet modules, which are assumed to converge within one of the synchronizing clock cycles. The processing done in each layer is given as a function of the clock cycle.

A. Input Layer

The input to the network is assumed to occur at time t_m and consists of two distinct parts, $x_0(t_m)$ and $\boldsymbol{x}(t_m) = \{x_i(t_m);\ i = 1, n\}$, where $x_0(t_m)$ has the encoded class membership, while $\{x_i(t_m);\ i = 1, n\}$ is the feature vector. The class membership is encoded in the x_0 node according to

$$x_0(t_m) = 1, \qquad \text{if } \boldsymbol{x} \in \text{class 1}$$

and

$$x_0(t_m) = 0, \qquad \text{if } \boldsymbol{x} \in \text{class 2}.$$

It is assumed that there are only two *a priori* classes. Each of the components $x_i(t_m)$ of $\boldsymbol{x}(t_m)$ represent an independent feature out of a total of n features yielding an n-dimensional feature space. Thus, the input to the network is a feature vector plus *a priori* class membership. The input layer does no active processing, it just passes the input values to the other layers. As described here, the network takes 11 time steps to complete its forward pass and node commitment cycle so that the next input can occur at t_{m+12}.

B. First Layer

The primary purpose of the first layer is to form Euclidean distances between the active nodes and the input feature vector, and to output the results to the fourth and fifth layers.

Each active node in this layer forms the Euclidean distance based on the feature vector inputs and its stored offset weights w_{ji}:

$$y_{1j}(t_{m+k+1}) = y_{2j}(t_{m+k}) \sum [w_{ji}(t_{m+k}) - x_i(t_{m+k})]^2 + \{1 - y_{2j}(t_{m+k})\}\, y_1^{\max} \qquad k = 0, 6$$

where $y_{1j}(t_{m+k+1})$ is the output produced by node j in this layer at time t_{m+k+1} and $y_1^{\max}$ is a constant with $y_1^{\max} > \rho$. This occurs on clock cycles $k = 0$, which corresponds to the downward pass and $k = 6$, which corresponds to the node commitment cycle. This result for node j is sent via the connections to the jth node in both the fourth and fifth layers. The node retains the feature vector component values until it receives a new input vector. Upon receiving a signal from the fifth layer (clock cycle 2), the jth node updates its offsets by the following update rule:

$$w_{ji}(n_j + 1) = w_{ji}(n_j)[n_j/(n_j + 1)] + x_i[1/(n_j + 1)]$$

and

$$n_j \rightarrow n_j + 1.$$

Each node in this layer is inactive until its corresponding node in the second layer turns on. When this signal first changes, the offsets are set to $w_{ji}(1) = x_i$ and $n_j = 1$, and $y_{ij}(t_{m+k+1})$ is recomputed.

C. Second Layer

The nodes in the second layer keep track of which nodes in the first, third, and fourth layers are active. All nodes start in the inactive state with an output value of zero.

$$y_{2j}(t = 0) = 0.$$

Each node activates when the inhibitory signal from its corresponding third layer node ceases. Starting at activation, each node sends out an output value of one to the corresponding nodes in the first, third, and fourth layers. This is done according to the update law

$$y_{2y}(t_k) = \theta\{ y_{2j}(t_{k-1}) + 1 - y_{3j}(t_{k-1}) - \epsilon\}, \qquad k > t_m.$$

This occurs on every clock cycle.

D. Third Layer

All third layer nodes start with offset biases of

$$o_{3j} = (1 - j\epsilon) \qquad \text{where } \epsilon < 1/N,$$

and N is the total number of nodes in each layer. The biases are updated according to

$$o_{3j}(t_k) = \{1 - y_{2j}(t_{k-1})\}[1 - j\epsilon], \qquad k > 0.$$

This occurs on every clock cycle. On a signal from the activation node ($\delta_a = 1$), a Maxnet process is initiated that determines the minimum (in j) node that is active. The winning node j' has a value of

$$z_{3j'}(t_k) = 1$$

for Maxnet initiated and completed during time step t_k and for all losing nodes

$$z_{3j}(t_k) = 0, \qquad j \neq j'.$$

The output for the third layer is determined by

$$y_{3j}(t = 0) = 1$$

$$y_{3j}(t_k) = \theta\{ y_{3j}(t_{k-1}) - \epsilon - z_{3j}(t_k)\} \qquad k > 0.$$

This output is sent to the corresponding second-layer node. Once the jth node wins the Maxnet competition, $z_{3j}(t_k) = 1$, the output $y_{3j}(t_k) = 0$, so that the jth node has deactivated. The winning node has committed its corresponding nodes in layers 1, 2, 4, 5, and 6 to the formation of a new cluster.

E. Fourth Layer

The fourth layer is a Maxnet layer that competitively finds the active node with the minimum Euclidean distance between the input feature vector and the cluster center. The output of each node in this layer is given by

$$y_{4j}(t_{m+k+1}) = \text{Maxnet\_Result}\,\{ y_{2j}(t_{m+k})/[1 + y_{1j}(t_{m+k})]\} \qquad k = 1, 7$$

where $\{ y_{2j}(t_{m+k})/[1 + y_{1j}(t_{m+k})]\}$ is the starting value of the jth node in the Maxnet module. After the winning node has been found, the results (1 for the winning node, 0 for all others) are output to the fifth layer. For the first pattern, no nodes are committed and $y_{4j}(t_{m+k+1}) = 0$ for all nodes. This occurs on clock cycles $k = 1$, which corresponds to the forward pass and $k = 7$, which corresponds to the node commitment cycle.

F. Fifth Layer

The results from layer four are used to gate the first-layer results. The result is one node with an activation equal to the minimum Euclidean distance. Each fifth-layer node performs the operation

$$y_{5j}(t_{m+k+1}) = y_{4j}(t_{m+k})\theta\{ \rho - y_{1j}(t_{m+k})\} \qquad k = 2, 8$$

which tests to see if the winning cluster is within a squared distance ρ of the input vector. If so, the output is one. If not, the activation node activates, triggering a new round of competition in the third layer. This occurs on clock cycles $k = 2$, which corresponds to the forward pass and $k = 8$, which corresponds to the node commitment cycle.

G. The Activation Node

The activation node forms the quantity

$$\delta_a(t_{m+k+1}) = \left\{ 1 - \sum y_{5j}(t_{m+k}) \right\}$$

where

$$\delta_a(t_{m+k+1}) = 1 \text{ if a) no active nodes } (y_{2j}(t_{m+k}) = 0 \text{ for all } j) \text{ or b) for } y_{2j}(t_{m+k}) = 1; \quad y_{1j} > \rho$$

$$\delta_a(t_{m+k+1}) = 0 \text{ otherwise.}$$

This occurs on clock cycles $k = 3$, which corresponds to the forward pass and $k = 9$, which corresponds to the node commitment cycle. On all other clock cycles, $\delta_a = 0$.

H. Output Layer

The output layer nodes are activated by the fifth-layer nodes and when activated, update offsets and compute output as

$$y_{oj}(t_{m+k+1}) = \{(n_j/[n_j + 1])\, y_{oj}(t_{m+k}) + (1/[n_j + 1])x_0(t_m)\}\, y_j(t_{m+k-1}) + y_{oj}(t_{m+k})\{1 - y_{5j}(t_{m+k-1})\};$$

$$n_j \geq 0.$$

$$n_j = n_j + y_{5j}(t_{m+k-1})$$

This occurs on clock cycle $k = 9$, which gives the network ample time to commit a new node if necessary. On all other clock cycles

$$y_{oj}(t_{m+k+1}) = y_{oj}(t_{m+k})$$

that is, y_{oj} retains its current value.

Clocking Summary

Layer	Completion Time
Input	m
First	$m + 1/m + 7$
Second	every clock cycle
Third	every clock cycle/Maxnet on signal from activation $(m + 5)$
Fourth	$m + 2/m + 8$
Fifth	$m + 3/m + 9$
Activation	$m + 4/m + 10$
Output	$m + 11$

VII. Analysis and Comparison with Classical Box Counting

The *a posteriori* probability estimates of the neural network have been compared with classical box counting estimates and the known true values for three two-dimensional, two-class test cases. In the first test case, observations for the two classes are drawn from two separate uniform distributions. The second test case consists of a Gaussian-distributed class versus a uniformly-distributed class, and the third case uses observations drawn from two different Gaussian distributions. In each case, *a priori* class probabilities of

$$P(c_1) = 0.6$$

and

$$P(c_2) = 0.4$$

are used. In all three cases the distributions are defined on the unit square. (This requires a slight alteration of the Gaussian distributions, with negligible effect.) The error measure used for comparison is an estimate of the standard root integrated-square error (STD ERROR) between the theoretical probability f and the estimate f'. Thus

$$\text{STD\_ERROR } (f, f') = \left| \sum (f - f')^2\, dx \right|^{0.5}$$

where the sum ranges over a discrete two-dimensional grid.

A uniform distribution pseudorandom number generator is used for Case 1 to first generate the class membership with a 60 percent chance of membership in class 1. It is then used to successively generate the two coordinates of the point. The theoretical *a posteriori* probability for class 1 is easily found using the results of Section III to be 0.6. In this case, the box counting histogram performs slightly better than the self-organizing neural network (SONN) approach in the standard error measure (see Table I). As this case is explicitly designed to allow for good performance from the box counting method, this is as expected. When analyzing these results, the slightly superior results for box counting in this simple case should be compared with the more significantly superior results of the SONN in cases 2 and 3.

The second test case uses a Gaussian distribution for class 1 with *a priori* probability 0.6. The distribution used is given by

$$p(\boldsymbol{x}|c_1) = \{k^2/(2\pi\sigma_x\sigma_y)\} \exp\{-0.5[(x - \mu_x)/\sigma_x]^2 - 0.5[(y - \mu_y)/\sigma_y]^2\}$$

where $\mu_x = \mu_y = \sigma_x = \sigma_y = 0.5$, $\boldsymbol{x} = (x, y)$, and k is a normalization constant used to normalize the distribution over the unit square instead of the normal real plane normalization. Class 2 in this case is again uniformly-distributed. Applying the results of Section III, we find the theoretical *a posteriori* probability for class 1 to be

$$P(c_1|\boldsymbol{x}) = 0.6p(\boldsymbol{x}|c_1)/[0.6p(\boldsymbol{x}|c_1) + 0.4].$$

This expression is evaluated at the center of each node for the neural network at the center of each box for the classical box counting and compared with the respective experimental results. Table I shows that, for this slightly less ideal case, the SONN outperforms the box counting method in the standard error measure by more than an order of magnitude.

Case 3 uses Gaussian distributions for both classes. The *a priori* probabilities of 0.6 and 0.4 are again used. The distribution used is the Gaussian distribution from the Case 2 with $\mu_x = \mu_y = \sigma_x = \sigma_y = 0.5$, and $k = 1/0.68268$ for class 1 and $\mu_x = \mu_y = 0$, $\sigma_x = \sigma_y = 1.0$, and $k = 1/0.34134$ for class 2. Applying the results of Section II, we find the theoretical *a posteriori* probability for class 1 to be

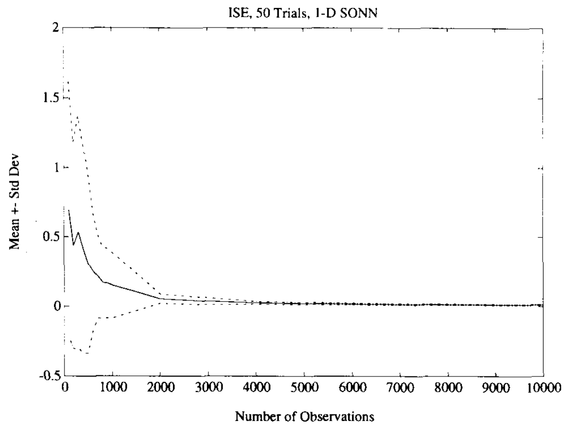
$$P(c_1|\boldsymbol{x}) = 0.6p(\boldsymbol{x}|c_1)/[0.6p(\boldsymbol{x}|c_1) + 0.4p(\boldsymbol{x}|c_2)].$$

The two Gaussian expressions are evaluated at the center of each mode for the neural network and at the center of each box for the classical box counting and used in the above expression to obtain the theoretical *a posteriori* values for comparison with the respective experimental results. Again, Table I indicates that the SONN approach outperforms the classical approach by more than an order of magnitude.

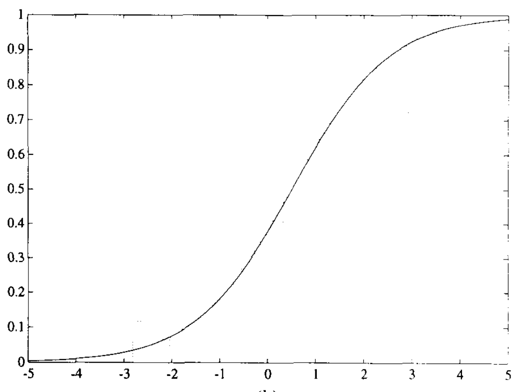
TABLE I
TRAINING AND TESTING RESULTS FOR THE NEURAL NETWORK VERSUS BOX COUNTING[a]

		SONN STD_ERROR	BOX STD_ERROR
CASE 1	TRAIN	.000164	.000106
	TEST	.000159	.000106
CASE 2	TRAIN	.000165	.002030
	TEST	.000162	.002030
CASE 3	TRAIN	.000164	.002146
	TEST	.000145	.002135

[a]Case 1: Uniform versus uniform, p (class 1) = 0.6 Case 2: Gaussian versus uniform, p (class 1) = 0.6. Case 3: Gaussian versus Gaussian p (class 1) = 0.6. SONN = =self-organizing neural network. BOX = =classical box counting method. STD_ERROR = =estimate of standard root-integrated-square error. Separate sets of $10e7$ two-dimensional variates used for training and testing. Errors are in comparison with known theoretical values for class 1 posterior probabilities.

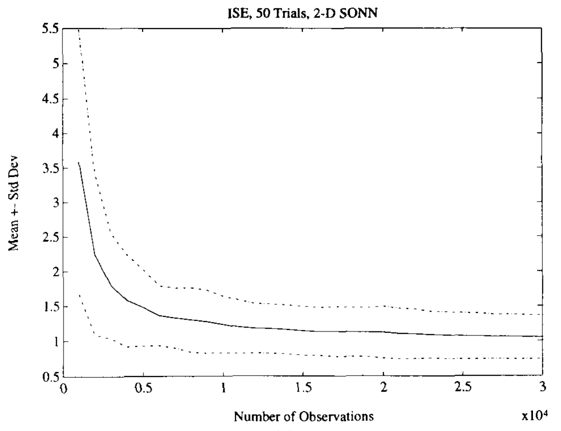


(a)

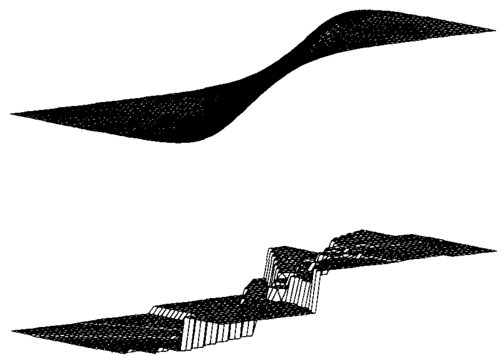


(b)

Fig. 9. (a) Mean integrated square error as a function of number of observations for the case of one-dimensional, normal distributions. (b) Example estimated posterior probability (dashed) versus true posterior probability for the case of one-dimensional, normal distributions.



(a)



(b)

Fig. 10. (a) Mean integrated square error as a function of number of observations for the case of two-dimensional, normal distributions. (b) Example estimated posterior probability (bottom) versus true posterior probability for the case of two-dimensional, normal distributions.

Further simulations are now presented to elucidate the network's development of posterior probability estimates. In the following we compare the performance of the SONN, versus the true optimal performance. Fig. 9(a) and (b) indicate the performance of the network on a one-dimensional input in which $P(c_1) = P(c_2) = 0.5$ and c_1 is normally distributed with mean zero and unit variance, while c_2 is normal with unit mean and unit variance. Fig. 9(a) shows the mean integrated square error and its standard deviation (based on 50 trials) as a function of the number of observations, and Fig. 9(b) compares the typical posterior estimate produced by the network with the true posterior probability function. Fig. 10(a) and (b) indicate performance on an analogous two-dimensional problem. Again, $P(c_1) = P(c_2) = 0.5$ and each class is normally distributed with a covariance matrix equal to the 2×2 identity matrix I_2. c_1 has a mean vector of $(0, 0)^T$ while c_2 has a mean vector of $(1, 1)^T$.

Figs. 11 and 12 indicate one- and two-dimensional performance in which the individual classes are distributed uniformly. In each case we have $P(c_1) = 0.3$ and $P(c_2) = 0.7$. For the one-dimensional example, $c_1 \sim U(0.2)$ and $c_2 \sim (0.5, 1.5)$. Fig. 11(a) shows the mean integrated square error as a function of the number of observations, Fig. 11(b) shows the number of nodes used in the network (dynamically allocated), also as a function of the number of observations, and Fig. 11(c) plots the true versus estimated posterior probability. For the two-dimensional example, c_1 is uniformly distributed on the square defined

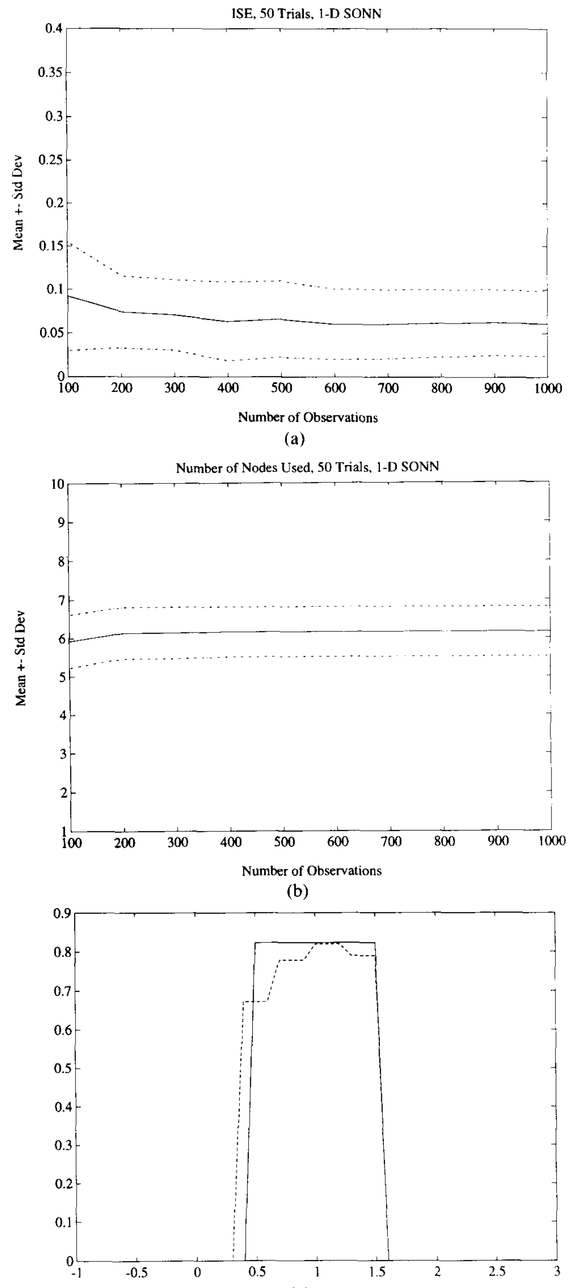


Fig. 11. (a) Mean integrated square error as a function of number of observations for the case of one-dimensional, uniform distributions. (b) Number of terms allocated in the network as a function of number of observations for the case of one-dimensional, uniform distributions. (c) Example estimated posterior probability (dashed) versus true posterior probability for the case of one-dimensional, uniform distributions.

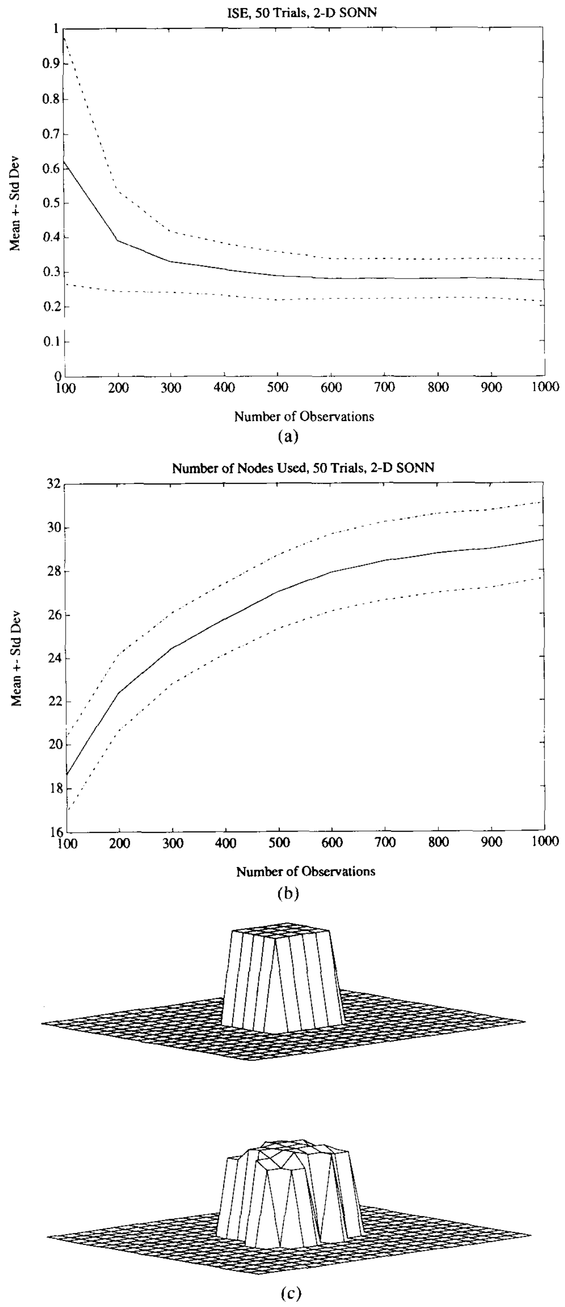


Fig. 12. (a) Mean integrated square error as a function of number of observations for the case of two-dimensional, uniform distributions. (b) Number of terms allocated in the network as a function of number of observations for the case of two-dimensional, uniform distributions. (c) Example estimated posterior probability (bottom) vs. true posterior probability for the case of two-dimensional, uniform distributions.

by vertices $a = (0, 0)$ and $b = (2, 2)$ and c_2 likewise with $a' = (0.5, 0.5)$ and $b' = (1.5, 1.5)$. Again, $P(c_1) = 0.3$ and $P(c_2) = 0.7$, and Fig. 12 shows (a) mean integrated square error, (b) number of nodes, and (c) posterior probability estimate.

The simulations considered in Figs. 9-12 indicate the fundamental trade-off between the number of nodes used in the estimation procedure, the number of observations available, and the performance (in, say, square-error) that one can expect. Analysis of this trade-off will be the focus of an asymptotic analysis of the network. Taken together with the tabulated performance given in Table I, our simulations give initial results indicating that the connectionist approach detailed herein is a viable approach to developing *a posteriori* probabilities for use in pattern recognition tasks.

VIII. Conclusion

We have presented a connectionist model for both supervised and unsupervised learning and probabilistic discrimination based on the Voronoi partitioning scheme used in vector quantization. This model is equivalent to a serial algorithm with strong foundations in classical theory. The approach is a Euclidean connectionist analog to the maximum likelihood ideas found in mixture models. In addition to presenting classical probability theory in a massively parallel framework, the model yields a connectionist formalization of automatic node creation, and thus has relevance to the many probabilistic neural network paradigms. Finally, the learning performed by the model has been favorably compared with a classical estimation technique in a series of simulation examples.

Acknowledgment

The authors would like to thank Wendy Poston for her help with the computer simulations and graphics that were used to characterize the performance of the network.

References

[1] P. J. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," Ph.D. dissertation, Harvard Univ. Comm. Appl. Math., Nov. 1974.

[2] D. E. Rumelhart, J. L. McClelland, and the PDP research group, *Parallel Distributed Processing Vol. 1: Explorations in the Microstructure of Cognition*. Cambridge, MA: MIT Press, 1986.

[3] D. F. Specht, "Probabilistic neural networks," *Neural Networks*, vol. 3, pp. 109-118, 1990.

[4] D. F. Specht, "A general recognition neural network," *IEEE Trans. Neural Networks*, vol. 2, pp. 568-576, 1991.

[5] L. I. Perlovsky and M. M. McManus, "Maximum likelihood neural networks for sensor fusion and adaptive classification," *Neural Networks*, vol. 4, pp. 89-102, 1991.

[6] H. G. C. Traven, "A neural network approach to statistical pattern classification by 'semiparametric' estimation of probability density functions," *IEEE Trans. Neural Networks*, vol. 2, pp. 366-377, 1991.

[7] S. Lee and R. H. Kil, "A Gaussian potential function network with hierarchically self-organizing learning," *Neural Networks*, vol. 4, pp. 207-224, 1991.

[8] D. J. Marchette and C. E. Priebe, "Adaptive kernel neural network," *Mathemat., Comput. Model.*, vol. 14, pp. 328-333, 1990.

[9] H. White, "Consequences and detection of misspecified nonlinear regression models," *J. Amer. Statist. Soc.*, vol. 76, pp. 419-433, 1981.

[10] P. Shoemaker, M. Carlin, R. Shimabukuro, and C. E. Priebe, "Least-squares learning and approximation of posterior probabilities on classification problems by neural networks," *Proc. 2nd Workshop Neural Networks*, WNN-AIND 91, pp. 187-196, 1991.

[11] C. Wolverton and T. Wagner, "Asympotically optimal discriminant functions for pattern classification," *IEEE Trans. Informat. Theory*, vol. 15, pp. 258-265, 1969.

[12] A. K. Krishnamurty, S. C. Ahalt, D. E. Melton, and P. Chen, "Neural networks for vector quantization of speech and images," *IEEE J. Select. Areas Commun.*, vol. 8, pp. 1449-1457, 1990.

[13] T. Lee and A. M. Peterson, "Adaptive vector quantization using a self-development neural network," *IEEE J. Select. Areas Commun.*, vol. 8, pp. 1458-1471, 1990.

[14] F. P. Preparata and M. I. Shamos, *Computational Geometry*. New York: Springer-Verlag, 1985.

[15] D. M. Titterington, A. F. M. Smith, and U. E. Makov, *Statistical Analysis of Finite Mixture Distributions*. New York: Wiley, 1985.

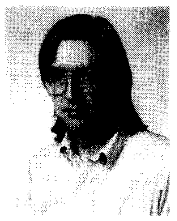
[16] G. Carpenter and S. Grossberg, "A massively parallel architecture for a self-organizing neural pattern recognition machine," *Comput. Vision, Graphics, Image Process.*, vol. 37, pp. 54-115, 1987.

[17] C. E. Priebe and D. J. Marchette, "Adaptive mixtures: Recursive nonparametric pattern recognition," *Pattern Recogn.*, vol. 12, pp. 1197-1209, 1991.

[18] F. J. Pineda, "Generalization of back-propagation to recurrent neural networks," *Phys. Rev. Lett.*, vol. 59, pp. 2229-2232, 1987.

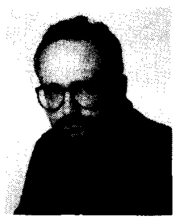
[19] Y. Pao, *Adaptive Pattern Recognition and Neural Networks*. Reading, MA: Addison-Wesley, 1989.

[20] R. P. Lippmann, "An introduction to computing with neural nets," *IEEE ASSP Mag.*, vol. 4, pp. 4-22, 1987.



George W. Rogers received the B.S. degree from Georgia Southern College in 1977 and the Ph.D. degree in theoretical physics from the University of South Carolina in 1984.

Since 1985 he has been employed at the Dahlgren Division of the Naval Surface Warfare Center where he first worked in orbit computation and more recently in the field of artificial neural networks. His current research interests are in composite neuronal dynamics and adaptive pattern recognition.



Jeffrey L. Solka received the B.S. degree in mathematics and chemistry in 1978 and the M.S. degree in mathematics from James Madison University in 1981, and the M.S. degree in physics from Virginia Polytechnic Institute and State University in 1989.

Since 1984, he has been working in the areas of strategic defense, artificial neural systems, and pattern recognition for the Dahlgren Division of the Naval Surface Warfare Center.

Stephen Malyevac received the B.S. and M.S. degrees in mechanical engineering from Virginia Polytechnic Institute and State University in 1986 and 1988, respectively.

Since March 1988 he has worked at the Dahlgren Division of the Naval Surface Warfare Center. His areas of interest include the application of control methods to practical systems, theoretical techniques for robust control design, and neural networks and their applications.

Carey E. Priebe received the B.S. degree in mathematics from Purdue University in 1984, the M.S. degree in computer science from San Diego State University in 1988, and the Ph.D. degree in information technology (computational statistics) from George Mason University in 1993. He also did two years of undergraduate work at the United States Military Academy, West Point, and graduate work in mathematics at the University of California at San Diego.

Since 1985, he has been working in adaptive systems and recursive estimators, first for the Naval Ocean Systems Center, San Diego, CA, and since April 1991, with the Naval Surface Warfare Center, Dahlgren, VA.