Competitive Neural Trees for Pattern Classification

Sven Behnke and Nicolaos B. Karayiannis, *Member, IEEE*

***Abstract*—This paper presents *competitive neural trees* (CNeT's) for pattern classification. The CNeT contains m-ary nodes and grows during learning by using inheritance to initialize new nodes. At the node level, the CNeT employs unsupervised competitive learning. The CNeT performs hierarchical clustering of the feature vectors presented to it as examples, while its growth is controlled by forward pruning. Because of the tree structure, the prototype in the CNeT close to any example can be determined by searching only a fraction of the tree. This paper introduces different search methods for the CNeT, which are utilized for training as well as for recall. The CNeT is evaluated and compared with existing classifiers on a variety of pattern classification problems.**

***Index Terms*—Classification, competitive learning, competitive neural tree, decision tree, neural tree, search method, splitting criterion, stopping criterion, tree pruning.**

I. Introduction

DECISION trees have extensively been used to perform decision making in pattern recognition [24]. By applying the decision tree methodology, one difficult decision can be split into a sequence of less difficult decisions. The first decision determines which decision has to be made next by indicating which node of the tree should be visited. Because of the tree structure, only some of all possible questions are asked in the process of making the final decision. In fact, the final decision is made at a terminal node of the tree, which is reached by traversing the tree starting from the root as indicated by the decisions made at internal nodes.

The design of decision trees is frequently performed in a *top-down* fashion. The nodes are split during the design process according to some criterion. The existing spitting criteria include the impurity measure used in *classification and regression trees* (CART's) [5], [6] and the mutual information measure employed by the *average mutual information gain* (AMIG) algorithm [31]. The terminal nodes are determined during the construction of the tree by freezing some of the nodes according to some stopping criterion or by growing a large tree and performing selective backward pruning. After the final tree structure is determined, the terminal nodes are frequently assigned class labels by using a majority rule.

Decision trees and multilayer feedforward neural networks are essentially competing methodologies for pattern classification. Atlas *et al.* [2] presented a performance comparison of multilayer neural networks and classification and regression trees used for load forecasting, power security, and vowel recognition. Neural trees were recently introduced for pattern classification in an attempt to combine advantages of neural networks and decision trees. The neural tree architectures reported in the literature can be grouped according to the learning paradigm employed for their training. Most of the existing neural tree architectures were directly or indirectly related to feedforward neural networks, although they differ in terms of the design methodology. In fact, the characterization "neural tree" was indistinguishably used to describe approaches employing decision trees as tools for building and training feedforward neural networks as well as approaches using feedforward neural networks as building elements in order to improve the design of decision trees.

The first family of approaches attempt to build neural networks either by developing tree structured neural networks for function approximation or by mapping decision trees to multilayer neural networks. Sanger [25] proposed a tree-structured adaptive network for function approximation in high-dimensional spaces. This approach is based on the hypothesis that only few dimensions of the input data are necessary to compute the desired output function. A learning procedure based on gradient descent grows a neural tree whose structure depends on the input data and the function to be approximated. Sethi [29], [30] proposed a procedure for mapping a decision tree into a multilayer feedforward neural network. This approach can be used for the systematic design of a class of multilayer neural networks, called *entropy networks.* A two-step methodology for designing entropy networks was proposed along with a rule for incremental learning. This methodology specifies the number of neurons needed in each layer of the network and leads to a learning procedure that allows each layer to be trained separately.

The second family of approaches attempt to develop tree structures containing feedforward neural networks in their nodes. Sankar and Mammone [26]–[28] introduced the *neural tree network* (NTN), a classifier consisting of single-layered neural networks connected in a tree architecture. These networks are used to recursively partition the feature space into subregions. The NTN grows by a heuristic learning procedure based on the $\mathcal{L}_1$ norm of the classification error. The generalization ability of the NTN is enhanced after training by an optimal pruning algorithm. Sankar and Mammone [27] used the NTN for speaker independent vowel recognition. Rahim used variations of the NTN for phoneme classification [22] and recognition of speech features [23]. Farrell *et al.* [8] modified the learning rule and pruning criteria employed by the NTN and compared the resulting modified NTN with various neural and conventional classifiers on speech recognition.

Manuscript received January 8, 1996; revised April 24, 1997.

S. Behnke is with the Institute of Computer Science, Free University of Berlin, 14195 Berlin, Germany.

N. B. Karayiannis is with the Department of Electrical and Computer Engineering, University of Houston, Houston, TX 77204-4793 USA.

Publisher Item Identifier S 1045-9227(98)08347-7.

Guo and Gelfand [10] used multilayer neural networks at the decision nodes of a binary classification tree to extract nonlinear features. They employed a gradient-type learning algorithm in conjunction with a class-aggregation algorithm to train the networks and grow the tree. This approach overcomes the problem of selecting the size of feedforward neural networks for classification applications by building a tree of an appropriate size and architecture. This is accomplished by a tree pruning algorithm. Guo and Gelfand evaluated their tree structure and compared it with the CART method on waveform recognition and a handwritten character recognition problem.

Most of the approaches mentioned above were motivated by the lack of a reliable procedure for determining the appropriate size of feedforward neural networks in practical classification applications. These approaches replace unstructured feedforward neural networks by structured neural architectures in order to facilitate learning and/or improve generalization by controlling the number of neurons and connections. An alternative approach to the development of neural trees was motivated by competitive learning. Li *et al.* [18], [19] developed adaptive neural trees for classification and vector quantization by combining competitive learning principles with structural adaptation during learning. The adaptive neural tree is a multilevel competitive neural network the nodes of which are organized in a tree topology. During training, all nodes of each level compete for each input. The connection weights of the winner are then updated using gradient descent learning. New nodes are added to the tree when the error rate exceeds a certain threshold and some nodes are deleted if they remain inactive for a long period. The inactive nodes are deleted during periodic traversals of the tree. This architecture splits the input space at each node until the problem is easy enough to be solved by a simple node [7], [18].

Neural trees are grown and pruned. Some algorithms grow a perfect tree that classifies all examples correctly [28]. Then a set of pruned subtrees is checked for performance on an independent testing set of examples and the best performing subtree is selected. This method is called *backward pruning*. Other algorithms perform *forward pruning* [7], [8], [18]. While the tree is growing, its performance on an independent testing set is checked. The tree stops growing when a stopping criterion triggers. Usually the growing of the tree is terminated when its performance on the testing set begins degrading. Forward pruning avoids the growth of large branches that will be pruned later.

Competitive learning is the key ingredient of several approaches to vector quantization design implemented through a neural model often referred to as *learning vector quantization* (LVQ) [11]. The combination of competitive learning and fuzzy-theoretic concepts resulted in a variety of learning vector quantization algorithms that can effectively deal with the uncertainty associated with the representation of the feature vectors by a finite set of prototypes [11]–[15]. Such LVQ models overcome the problems often associated with hard or crisp LVQ algorithms, i.e., learning algorithms that allow only the update of the prototype that is the closest to the feature vector presented to the network. An alternative approach to learning vector quantization resulted in *competitive neural trees* (CNeT's), which perform hierarchical clustering of the feature vectors and employ competitive learning at the node level [3], [4]. The CNeT employs the same learning rule that is associated with crisp LVQ algorithms. Nevertheless, the CNeT is capable of resolving the uncertainty associated with the representation of the feature vectors by the prototypes by creating a structured partition of the feature space. This partition depends on the structure of the corresponding tree, which is determined by the strategy employed for creating and pruning nodes during learning as indicated by the structure of the feature space.

This paper presents CNeT's that can be grown and trained by a supervised learning procedure to perform pattern classification. This paper is organized as follows: Section II introduces the CNeT architecture and presents a generic learning algorithm. Section III describes several search methods. Splitting and stopping criteria are presented in Section IV. Recall procedures are described in Section V. Section VI benchmarks the CNeT using the double spiral problem, the IRIS data set, a vowel recognition problem, and a handwritten digit recognition task. Finally, Section VII summarizes the results and draws conclusions.

II. CNeT Architecture and Learning

CNeT's are self-organizing neural architectures that combine the advantages of competitive neural networks and decision trees. Among other applications, CNeT's can be used to perform *pattern classification*. Consider a set $\mathcal{X} \subset \mathbb{R}^n$ of feature vectors from an n-dimensional Euclidean space which belong to s distinct classes $\mathcal{C}_1, \mathcal{C}_2, \cdots, \mathcal{C}_s$. The classes $\mathcal{C}_1, \mathcal{C}_2, \cdots, \mathcal{C}_s$ form a partition of $\mathcal{X}$ such that $\mathcal{C}_i \cap \mathcal{C}_j = \varnothing, \forall\, i \neq j$, and $\cup_{j=1}^{s} \mathcal{C}_j = \mathcal{X}$. Let $\mathcal{L} = \{1, 2, \cdots, s\}$ be the set of class labels. Each $\mathbf{x} \in \mathcal{X}$ is assigned a class label $\ell(\mathbf{x}) \in \mathcal{L}$ in such a way that $\ell(\mathbf{x}) = j$ if $\mathbf{x} \in \mathcal{C}_j$. The objective of a pattern classification scheme is to find a representation of the examples such that the class labels $\ell(\mathbf{x})$ can be reproduced for the examples. An efficient pattern classification scheme must be capable of producing appropriate class labels for input vectors that do not belong to the training set. This is called *generalization*.

A. Architecture

The CNeT has a structured architecture. A hierarchy of identical *nodes* forms a m-ary tree as shown in Fig. 1(a). Fig. 1(b) shows a node in detail. Each node contains m *slots* $\mathtt{s}_1, \mathtt{s}_2, \cdots, \mathtt{s}_m$ and a counter `age` that is incremented each time an example is presented to that node. The behavior of the node changes as the counter `age` increases. Each slot $\mathtt{s}_i$ stores a *prototype* $\mathbf{v}_i \in \mathcal{V} \subset \mathbb{R}^n$, a counter `count`, and a pointer to a node. The prototypes are updated to represent clusters of examples. The slot counter `count` is incremented each time the prototype of that slot is updated to match an example. Finally, the pointer contained in each slot may point to a child-node assigned to that slot. A `NULL` pointer indicates that no node was created as a child so far. In this case, the slot is called *terminal slot* or *leaf*. *Internal slots* are slots with an assigned child-node. The slots shown in Fig. 1(b) also have class

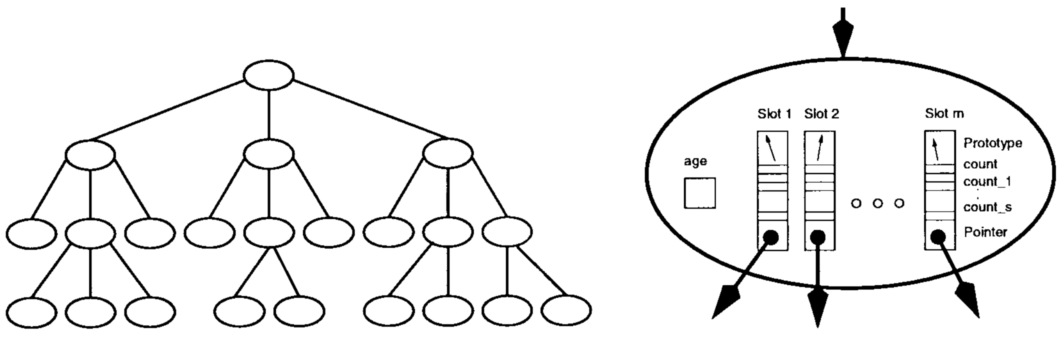


Fig. 1. The architecture of the CNeT: (a) the tree structure and (b) a node in detail.

counters. The class counters $\mathtt{count}_1, \mathtt{count}_2, \cdots, \mathtt{count}_s$ keep track of the number of examples belonging to a particular class that the prototype responds to. There is a counter for each class in every slot.

B. Learning

In the learning phase, the tree grows starting from a single node, the root. The prototypes of each node form a minuscule competitive network. When an example $\mathbf{x} \in \mathcal{X}$ arrives at a node, all of its prototypes $\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_m$ compete to match it. If $d(\mathbf{x}, \mathbf{v}_j)$ denotes the distance between $\mathbf{x}$ and $\mathbf{v}_j$, the prototype $\mathbf{v}_k$ is the winner if $d(\mathbf{x}, \mathbf{v}_k) < d(\mathbf{x}, \mathbf{v}_j), \forall j \neq k$. The distance measure used in this paper is the squared Euclidean norm, defined as

$$d(\mathbf{x}, \mathbf{v}_j) = \|\mathbf{x} - \mathbf{v}_j\|^2. \tag{1}$$

The competitive learning scheme used at the node level resembles that employed by the (unlabeled data) LVQ, an unsupervised learning algorithm proposed to generate crisp c-partitions of a set of unlabeled data vectors [16], [17]. According to this scheme, the winner $\mathbf{v}_k$ is the only prototype that is attracted by the input $\mathbf{x}$ arriving at the node. More specifically, the winner $\mathbf{v}_k$ is updated according to the equation

$$\mathbf{v}_k^{new} = \mathbf{v}_k^{old} + \alpha\,(\mathbf{x} - \mathbf{v}_k^{old}) \tag{2}$$

where α is the learning rate. The learning rate α decreases exponentially with the age of a node according to the equation

$$\alpha = \alpha_0 \exp(-\alpha_d\, \mathsf{age}) \tag{3}$$

where α_0 is the initial value of the learning rate and α_d determines how fast α decreases.

The update (2) moves the winner $\mathbf{v}_k$ closer to the example $\mathbf{x}$ and, therefore, decreases the distance between the two. After a sequence of example presentations and updates, each of the prototypes will respond to examples from a particular subregion of the input space. Each prototype $\mathbf{v}_j$ attracts a cluster of examples $\mathcal{R}_j$. Hence the prototypes split the region of the input space that the node sees into subregions. The examples that are located in a subregion constitute the input for a node on the next level of the tree that may be created after the node is mature. A new node will only be created if a splitting criterion is TRUE.

1) Life Cycle: Each node goes through a life cycle. The node is created and ages with the exposure to examples. When a node is mature, new nodes can be assigned as children to it. A child-node is created by copying properties of the slot that is split to the slots of the new node. More specifically, the child will inherit the prototype of the parent slot as well as fractions of its class counters. Right after the creation of a node, all its slots are identical. They will differentiate with the exposure to examples. As soon as a child is assigned to a node, that node is frozen. Its prototypes are no longer updated in order to keep the partition of the input space for the child-nodes constant. A node may be destroyed after all of its children have been destroyed. The life cycle of a node may be partitioned into the following phases.

1) **Creation** (at age 0):
   a) the node is initialized;
   b) the node inherits properties from the parent slot such as the prototype and a fraction of the class counters.
2) **Youth** (before the maturity age τ is reached):
   a) the prototypes compete to respond to the examples;
   b) the winning prototype is updated;
   c) the prototypes split the region of the input space that the node sees into subregions.
3) **Maturity** (after the maturity age τ has been reached):
   a) the prototypes still compete for the examples and they are updated;
   b) if a splitting criterion is TRUE, then a new child is created and is assigned to a slot.
4) **Frozen** (as soon as a child is assigned):
   a) the prototypes compete for the inputs but they are not updated;
   b) if the winner has a child-node assigned, then it sends the example to the child.
5) **Destruction** (after all children have been destroyed).

2) Training Procedure: If the CNeT is used for pattern classification, its goal is to partition the input space into regions that are *pure* or almost pure. A pure region contains only examples from the same class. The general learning scheme works as follows.

Do while stopping criterion is FALSE:

1) Select randomly an example $\mathbf{x}$. Let $\mathcal{C}_j$ be class that $\mathbf{x}$ belongs to.
2) Traverse the tree starting from the root to find a terminal prototype $\mathbf{v}_k$ that is close to $\mathbf{x}$. Let $\mathbf{n}_\ell$ and $\mathbf{s}_k$ be the node and the slot that $\mathbf{v}_k$ belongs to, respectively.
3) If the node $\mathbf{n}_\ell$ is not frozen, then update the prototype $\mathbf{v}_k$ according to (2).
4) If a splitting criterion for the slot $\mathbf{s}_k$ is TRUE, then assign a new node as child to $\mathbf{s}_k$ and freeze the node $\mathbf{n}_\ell$.
5) Increment the counter $\texttt{count}_j$ for class $\mathcal{C}_j$, the counter count in slot $\mathbf{s}_k$, and the counter age in node $\mathbf{n}_\ell$.

Depending on how the search in Step 2) is implemented, various learning algorithms can be developed. The search is the only operation in the learning algorithm that depends on the size of the tree. Hence, the speed of the learning process is mainly determined by the computational complexity of the search method. Different search methods are described in the next section. Given a search method, the training process can be accelerated through some simple modifications of the learning algorithm outlined below.

3) Acceleration of Training: If there is a large number of examples, a high maturity age may be needed to ensure proper training of the tree. This affects the time required for training. The higher the maturity age τ is chosen, the longer the learning algorithm runs. Therefore, it might be useful to introduce a maturity age τ_i that is specific to a node $\mathbf{n}_i$, instead of using the same maturity age τ for all nodes. Since the root is presented with more examples than any other node of the tree, it is necessary to assign the highest maturity age τ_0 to the root $\mathbf{n}_0$ of the tree. The initial high maturity age can decrease as the training proceeds. Since fewer examples arrive at the nodes deeper in the tree, the assignment of lower maturity ages to these nodes does not have a negative impact on the clustering of the examples. In fact, decreasing the maturity age of these nodes can accelerate significantly the learning process. The obvious reason is that most of the learning time is spent training the nodes at the deepest levels of the tree.

In order to grow balanced trees of reasonable depth, the maturity age of each node should not be directly dependent on the depth of that node. Such a scheme would allow branches that grow deep to split more frequently than shorter branches. Frequent splitting accelerates the creation of new nodes, thus resulting in degenerate tree structures. The following scheme can be used to balance the tradeoff between the speed of training and the preservation of a balanced tree structure: let N_{tree}^i be the number of terminal prototypes in the tree when the node $\mathbf{n}_i$ is created. At creation, the node $\mathbf{n}_i$ is assigned the maturity age $\tau_i = \tau_0/\log_2(N_{tree}^i)$. To ensure sufficient training, a node $\mathbf{n}_i$ with low maturity age τ_i must be assigned a high learning rate α_0^i and decay parameter α_d^i. If α_0^0 and α_d^0 denote the initial values of the learning rate and the decay parameter of the root, respectively, sufficient training is guaranteed by choosing $\alpha_0^i = (\tau_0/\tau_i)\alpha_0^0$ and $\alpha_d^i = (\tau_0/\tau_i)\alpha_d^0$.

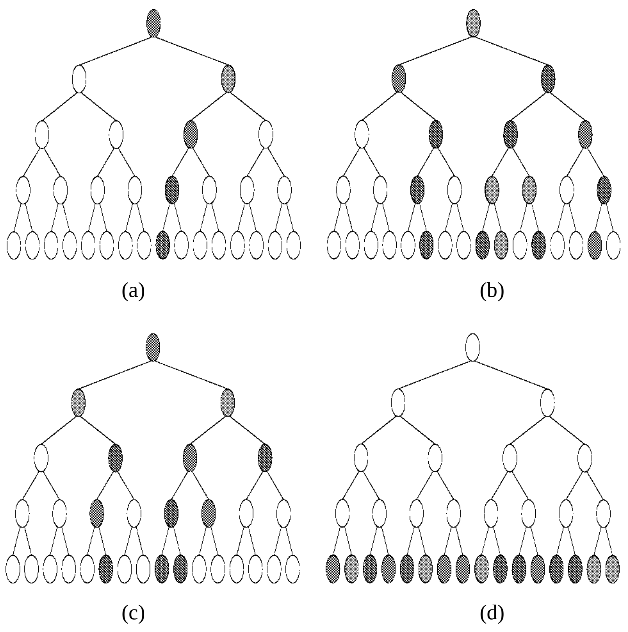

(a) (b) (c) (d)

Fig. 2. Shaded nodes visited by (a) the greedy search method, (b) the local search method, (c) the global(3) search method, and (d) the full search method.

III. Search Methods

The search method determines the speed of learning and recall as well as the generalization ability of the trained CNeT. A feature vector $\mathbf{x}$ constitutes the input for the search. An exhaustive search of the tree is guaranteed to return the closest prototype $\mathbf{v}_k$ to the input vector $\mathbf{x}$. Because of the computational and time requirements associated with an exhaustive search, alternative search methods can be employed for determining a terminal prototype $\mathbf{v}_k$ that is close, but not necessarily the closest, to the input $\mathbf{x}$. During learning, any terminal prototype $\mathbf{v}_j \in \mathcal{V}$ is a candidate to be selected by the search method. In contrast, only the prototypes that responded during learning to at least one example are candidates to be selected in the recall phase. Fig. 2 shows which nodes are *visited* and *expanded* in a complete binary tree by the search methods described in this section. Expanding an m-ary node means computing the distances between all of its prototypes $\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_m$ and the feature vector $\mathbf{x}$. Throughout this paper, N_{tree} represents the number of terminal prototypes of the tree and D_{tree} represents the depth of the tree, that is, the maximum number of edges on the path from a terminal node to the root.

A. Full Search Method

The full search method is based on conservative exhaustive search. To guarantee that the prototype $\mathbf{v}_k$ with the minimum distance to the given feature vector $\mathbf{x}$ is returned, it is necessary to compute the distances $d(\mathbf{x}, \mathbf{v}_j)$ between the input vector $\mathbf{x}$ and each of the terminal prototypes $\mathbf{v}_j \in \mathcal{V}$. The prototype $\mathbf{v}_k$ with the minimum distance is returned. Table I shows the full search method in pseudocode.

TABLE I
THE FULL SEARCH METHOD IN PSEUDOCODE

```
full(x)
  • $d_{min}$ := MAXDISTANCE
  • $\mathbf{v}_{win} := \mathbf{v}_1$
  • for all $\mathbf{v}_j \in \mathcal{V}$
      do  if $(d(\mathbf{x}, \mathbf{v}_j) < d_{min})$ then $d_{min} := d(\mathbf{x}, \mathbf{v}_j)$, $\mathbf{v}_{win} := \mathbf{v}_j$
  • return $(\mathbf{v}_{win})$
```

The full search method is the slowest among the search methods described in this section, since it does not take advantage of the tree structure to find the closest prototype. As a result, the full search method runs in $O(N_{tree})$. On the other hand, this is the only search method that guarantees the return of the closest prototype to the input vector.

B. Greedy Search Method

The greedy search method starts at the root of the tree and proceeds in a greedy fashion. When the search method arrives at a certain node, the distances between the input vector and all prototypes $\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_m$ of the node are computed. The prototype $\mathbf{v}_k$ with the minimum distance to the presented feature vector $\mathbf{x}$ is selected and called the winner. If a child-node is assigned to the slot $\mathbf{s}_k$ that contains the winner, then the greedy method is applied recursively to that node. Otherwise, the winner $\mathbf{v}_k$ is returned to the calling function. Table II shows the greedy search method in pseudocode.

The greedy method expands only one slot per level. Therefore the searched subtree is only a simple path from the root to the returned prototype. Since the time needed to expand each node is constant, the greedy search runs in $O(D_{tree})$. For trees that are balanced to some extent, the running time is $O(\log N_{tree})$. The greedy search method is the fastest among the search methods described in this section and local. However, whenever a prototype is not the winner, the subtree whose root is the corresponding node is not searched. As a result, the greedy method will not always return the terminal prototype with the minimum distance to the presented feature vector $\mathbf{x}$. If the greedy search method is employed in the learning phase, the generalization ability of the resulting CNeT will be inferior to that of a CNeT trained using the full search method.

C. Local Search Method

The local search method is designed to find the terminal prototype $\mathbf{v}_k$ that is close to the input vector even though a prototype on the path from the root to $\mathbf{v}_k$ has only been the second best prototype in its node. The local search method searches a subtree that has at most i nodes on the ith level ($i = 1, 2, \cdots, D_{tree}$). The terminal prototypes in this subtree are those which have at most one prototype on their path to the root that was only the second best. The local search method expands the nodes of the tree starting from the root. When the search method arrives at a certain node, the distances between the input vector $\mathbf{x}$ and all prototypes $\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_m$ in the node are calculated. Then, the local search method is called recursively for the winning prototype. In the subtree that is below the winner, the prototypes are still allowed to loose. The greedy search method is called for the second best prototype. In its subtree prototypes are not allowed to loose once more. They have to win in order to be expanded by the search method. If a winning prototype has no child-node assigned, it cannot be expanded. Instead it is returned to the calling function. The two recursive calls to the local and the greedy search methods will both return a prototype that is close to the input. The closest of these two prototypes to the input is then determined and returned. Table III shows the local search method in pseudocode.

The local search method uses only information that is locally available to make decisions. At most i slots are expanded on the ith level of the tree. Therefore, the time required by the local search method to return a terminal prototype is $O(D_{tree}^2)$, which is usually $O((\log N_{tree})^2)$. Note that the method is more efficient for trees with a high fan-out m. The local search method returns the closest terminal prototype more often than the greedy method. Thus, the use of the local search method instead of the greedy search method is expected to improve the generalization ability of the CNeT. Although the local search method is easy to implement and fast for small trees, its asymptotic running time is quadratic in the depth of the tree D_{tree}. If the use of a local method is not necessary for a given application, the time required for the search can be moderated by using the global search method.

D. Global(ω) Search Method

The global(ω) search method expands the nodes of the tree level by level, starting at the root. After this is done for all the nodes that are to be expanded at this level of the tree, the ω prototypes with the smallest distances are selected. If a selected prototype has a child-node assigned, this child-node will be expanded during the next expansion step. Suppose a selected prototype is a terminal prototype. If its distance to the given feature vector is the smallest so far, then the smallest distance is updated and the prototype is the new candidate to be selected for return. When no more prototypes are to be expanded, the global(ω) search method terminates and returns the best terminal prototype seen. Table IV shows the global(ω) search method in pseudocode.

The global(ω) search method searches a subtree that has a width of at most ω. Hence, the time required by the global(ω) search method to return a terminal prototype is $O(\omega D_{tree})$. Clearly, the speed of this search method depends on the choice of the search width ω. If $\omega = 1$, then the global(ω) and the greedy search methods are equivalent in terms of their time requirements. If $\omega > 1$, then the search by the global(ω) method takes longer but the probability that the search returns the closest prototype increases. Thus, the selection of ω allows the user to balance the tradeoff between the time required for

TABLE II
THE GREEDY SEARCH METHOD IN PSEUDOCODE

```
greedy(x)
  • $\mathbf{v}_{win} := \mathbf{v}_k \in \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m\}$ such that $d(\mathbf{x}, \mathbf{v}_k) \leq d(\mathbf{x}, \mathbf{v}_j) \forall j \neq k$
  • $\mathbf{s}_{win}$ := the slot $\mathbf{v}_{win}$ belongs to
  • if ($\mathbf{s}_{win}$.child ≠ NULL)  then return ($\mathbf{s}_{win}$.child → greedy(x))
                                        else return ($\mathbf{v}_{win}$)
```

TABLE III
THE LOCAL SEARCH METHOD IN PSEUDOCODE

```
local(x)
  • $\mathbf{v}_{win} := \mathbf{v}_{k_1} \in \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m\}$ such that $d(\mathbf{x}, \mathbf{v}_{k_1}) \leq d(\mathbf{x}, \mathbf{v}_j) \forall j \neq k_1$
  • $\mathbf{v}_{sec} := \mathbf{v}_{k_2} \in \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m\}$ such that $k_2 \neq k_1$ and $d(\mathbf{x}, \mathbf{v}_{k_2}) \leq d(\mathbf{x}, \mathbf{v}_j) \forall j \neq$
    $k_1, j \neq k_2$
  • $\mathbf{s}_{win}$ := the slot $\mathbf{v}_{win}$ belongs to
  • if ($\mathbf{s}_{win}$.child ≠ NULL)  then $\mathbf{v}_{win}$ := $\mathbf{s}_{win}$.child → local(x)
  • $\mathbf{s}_{sec}$ := the slot $\mathbf{v}_{sec}$ belongs to
  • if ($\mathbf{s}_{sec}$.child ≠ NULL)  then let $\mathbf{v}_{sec}$ be $\mathbf{s}_{sec}$.child → greedy(x)
  • if ($d(\mathbf{x}, \mathbf{v}_{sec}) < d(\mathbf{x}, \mathbf{v}_{win})$)  then return ($\mathbf{v}_{sec}$)
                                           else return ($\mathbf{v}_{win}$)
```

the search and the generalization ability of the CNeT. Since ω is a parameter that is not growing with the problem size, $O(\omega D_{tree})$ is not higher than $O(D_{tree})$. In other words, the time required for the global(ω) search method grows with the problem size only as fast as the time required for the greedy search method.

IV. SPLITTING AND STOPPING CRITERIA

The splitting criterion must be TRUE when the proposed leaning algorithm splits a slot and creates a new node. Meaningful splitting criteria are needed for growing trees that have desired features such as balance and small size. It is also important to give each node enough time to find a good partition of its input region. For this reason, splitting of immature nodes must be avoided. The splitting criterion will be FALSE for nodes below the maturity age τ. The stopping criterion is used to terminate the training. Splitting and stopping criteria work together to prevent the tree from growing too large. Extensive growth is not desirable for the following reasons.

- Large trees need more resources than small trees.
- Large trees tend to memorize the examples. This has a negative impact on generalization.

A. Splitting

The task of the splitting criterion is to split a slot, if this improves local classification performance, or not to split it, if the local classification is satisfactory. This is achieved by testing the slot for purity. After the maturity age τ is reached, a slot $\mathbf{s}_k$ may be split if it is not pure enough. Whether or not $\mathbf{s}_k$ is pure enough is determined by its class counters $\texttt{count}_1, \texttt{count}_2, \cdots, \texttt{count}_s$. The class counter $\texttt{count}_\ell$ with the highest value indicates that the prototype $\mathbf{v}_k$ from the slot $\mathbf{s}_k$ has responded most of the times to feature vectors belonging to the class $\mathcal{C}_\ell$. The slot $\mathbf{s}_k$ is not pure enough if a significant fraction of its prototypes responds to feature vectors belonging to classes other than $\mathcal{C}_\ell$. In this case, further splitting is required. The *purity parameter* β determines the fraction of acceptable local misclassification and typically varies from 0 to 1/2. The recommended default setting is $\beta = 1/8$.

TABLE IV
THE GLOBAL(ω) SEARCH METHOD IN PSEUDOCODE

```
global($\mathbf{x}$, $\omega$)

• $d_{min}$ := MAXDISTANCE

• $\mathbf{v}_{win}$ := $\mathbf{v}_1$

• active_nodes := $n_0$

• while (active_nodes $\neq \emptyset$)
    do  o active_slots := $\emptyset$
        o for all nodes $n_i \in$ active_nodes
            do  for all slots $s_j$ in $n_i$
                do  $\mathbf{v}_j$ := the prototype in $s_j$
                    if ($s_j$.child $\neq$ NULL)
                    then active_slots := active_slots $\cup \{s_j\}$
                    else if ($d(\mathbf{x}, \mathbf{v}_j) < d_{min}$) then $d_{min} := d(\mathbf{x}, \mathbf{v}_j)$, $\mathbf{v}_{win} := \mathbf{v}_j$

        o active_nodes := $\emptyset$
        o for the $\omega$ slots $s_j \in$ active_slots that have the smallest $d(\mathbf{x}, \mathbf{v}_j)$
            do  active_nodes := active_nodes $\cup$ {$s_j$.child}

• return ($\mathbf{v}_{win}$)
```

B. Stopping

The CNeT can easily be trained until all examples $\mathbf{x} \in \mathcal{X}$ are classified correctly by choosing the purity parameter $\beta = 0$. However, for the majority of problems this choice will result in memorization of examples and degradation of the generalization ability. The generalization ability of the trained tree can be evaluated using a testing set of examples $\mathcal{X}'$, selected such that $\mathcal{X}' \cap \mathcal{X} = \emptyset$.

- **Maturity Stopping Criterion**: This stopping criterion is based on a good choice of the purity parameter β, that will prevent the tree from growing too large. The maturity stop criterion becomes TRUE as soon as all nodes are mature. That means the tree does not grow any longer and all nodes have reached the maturity age τ. To achieve satisfactory generalization, it is important to combine the maturity stopping criterion with a splitting criterion that does not split nodes which are almost pure. The purity parameter β influences the size of the grown tree and decides when the maturity stopping criterion triggers.
- **Testing-Set Stopping Criterion**: This stopping criterion utilizes an independent testing set $\mathcal{X}'$ of examples. According to this criterion, training is terminated if the performance of the tree on the testing set $\mathcal{X}'$ does not improve for $\tau/4$ adaptation cycles, where τ is the maturity age. This criterion attempts to minimize the number of incorrect classifications on the testing set. This is accomplished by maintaining a counter $\mathtt{min}_{\mathrm{err}}$ that stores the minimum number of incorrect classifications for the testing set so far. This counter will be updated frequently in the beginning of the training process, since every time a slot is split the performance of the tree most likely improves on the training set $\mathcal{X}$ as well as on the testing set $\mathcal{X}'$. These updates of $\mathtt{min}_{\mathrm{err}}$ will become less frequent as training progresses. If the number of incorrect classifications on the testing set $\mathcal{X}'$ is not decreasing below $\mathtt{min}_{\mathrm{err}}$ for a long time, the tree is most likely too large due to overtraining. Therefore the testing-set stopping criterion triggers. When the last decrease of incorrect classifications on the testing set occurred, the tree had a better performance and a smaller size. Hence the tree must be pruned. After the testing-set stopping criterion triggers, all nodes that did not exist at the time the last update of $\mathtt{min}_{\mathrm{err}}$ occurred are deleted.

V. RECALL PROCEDURES

The trained CNeT contains a representation of the examples $\mathcal{X}$ that have been presented to it. Once the training process is complete, the terminal prototypes $\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_t$ of the CNeT represent the example clusters $\mathcal{R}_1, \mathcal{R}_2, \cdots, \mathcal{R}_t$. For this particular application, the tree is grown in such a way that each cluster of examples $\mathcal{R}_j$ contains examples $\mathbf{x} \in \mathcal{X}$ which may belong to one or more classes. The objective of the recall procedure is to produce a class label $i = \ell(\mathbf{x})$, $i \in \mathcal{L}$, for each input vector $\mathbf{x}$. The function $\ell(\mathbf{x})$ can be approximated by the composition $q(v(\mathbf{x}))$, where $q(\cdot)$ is a function that assigns class labels to the prototypes $\mathbf{v}_k = v(\mathbf{x}) \in \mathcal{V}$. The approximation of $\ell(\cdot)$ by $q(v(\cdot))$ is efficient if the number of clusters inherent in the example set is small compared to the number of examples.

Presentation of examples in the recall phase begins at the root of the tree and proceeds down to the leaves. The prototypes belonging to internal slots are used as signposts for the search. More specifically, these prototypes guide the search algorithm to find a terminal prototype $\mathbf{v}_j = v(\mathbf{x})$ that is close to the input vector $\mathbf{x}$ without looking at all terminal prototypes. The same search method used during training is also called for recall with the input vector $\mathbf{x}$ as the argument. The search will return a terminal prototype $\mathbf{v}_k = v(\mathbf{x}) \in \mathcal{V}$, close to $\mathbf{x}$. A class label $i = \ell(\mathbf{x})$ is then assigned to the input vector $\mathbf{x}$ by composing $q(\cdot)$ and $v(\cdot)$ as $\ell(\mathbf{x}) = q(\mathbf{v}_k) = q(v(\mathbf{x}))$. If $\mathbf{v}_k$ belongs to the slot $\mathbf{s}_k$, then $q(\mathbf{v}_k)$ is computed on the basis of the class counters $\mathtt{count}_1, \mathtt{count}_2, \cdots, \mathtt{count}_s$ stored in the slot $\mathbf{s}_k$. These class counters indicate how often the prototypes in the slot $\mathbf{s}_k$ responded to examples from each class. Depending on the specific objectives, the classification function $q(\cdot)$ can be evaluated according to the following two methods.

- **Crisp Classification**: The class label $\ell(\mathbf{x}) = q(\mathbf{v}_k)$ is produced by majority vote. The feature vector $\mathbf{x}$ is classified to belong to the class $\mathcal{C}_i$ with the highest class counter $\mathtt{count}_i$ in the slot $\mathbf{s}_k$ that responded to the input vector $\mathbf{x}$.
- **Likelihood Estimation**: This method does not return a class label but an estimate of the likelihood that a certain input vector belongs to the classes $\mathcal{C}_1, \mathcal{C}_2, \cdots, \mathcal{C}_s$. The likelihood that an input vector belongs to the class $\mathcal{C}_j$ is determined as the ratio between the corresponding class counter $\mathtt{count}_j$ and the sum $\mathtt{count}$ of all class counters. Furthermore the distance between the input vector $\mathbf{x}$ and the responding prototype $\mathbf{v}_k$ can be utilized as a measure of confidence, with a smaller distance corresponding to higher confidence.

The recall strategy for the CNeT can be modified to allow the rejection of some examples in order to improve classification accuracy. Suppose $\mathbf{x}$ is the input example to the CNeT and the search method returns a terminal prototype $\mathbf{v}_k \in \mathcal{V}$, which belongs to the slot $\mathbf{s}_k$ that stores the class counters $\mathtt{count}_1, \mathtt{count}_2, \cdots, \mathtt{count}_s$. Let $\mathtt{count}$ be the sum of all class counters $\mathtt{count}_1, \mathtt{count}_2, \cdots, \mathtt{count}_s$. According to the recall strategy based on likelihood estimation, the ratio $\mathtt{count}_i/\mathtt{count}$ provides an estimate of the likelihood that the example $\mathbf{x}$ belongs to the ith class, that is, $\mathbf{x} \in \mathcal{C}_i$. Let k be the class label such that $\mathtt{count}_k \geq \mathtt{count}_i, \forall\, i \neq k$. The decision about the example $\mathbf{x}$ is made by comparing the likelihood ratio $\mathtt{count}_k/\mathtt{count}$ with the *rejection parameter* $r \in [0, 1]$ as follows: if $\mathtt{count}_k/\mathtt{count} \geq r$, then the example $\mathbf{x}$ is assigned the class label k. The input example is *recognized* if k is its actual label and *substituted* otherwise. If $\mathtt{count}_k/\mathtt{count} < r$, then the example $\mathbf{x}$ is *rejected*. The role of the rejection parameter $r \in [0, 1]$ becomes clear by considering the two extreme cases $r = 0$ and $r = 1$. If $r = 0$, none of the examples is rejected since $\mathtt{count}_k/\mathtt{count} \geq 0$. If $r = 1$, the example is rejected unless $\mathtt{count}_k/\mathtt{count} = 1$, which implies that the corresponding slot is perfectly pure. Thus, for $r = 1$ this strategy rejects all examples which correspond to a slot that is not perfectly pure. Any other value of $r \in (0, 1)$ determines the degree of impurity of a slot that is tolerated before the corresponding example is rejected.

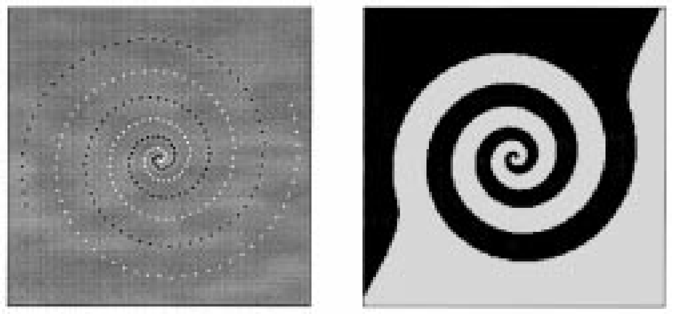


Fig. 3. The double spiral problem: (a) the 256 training samples and (b) the desired generalization.

VI. Experimental Results

This section evaluates the performance of the CNeT and existing classifiers on a variety of pattern classification problems.

A. The Double Spiral Problem

The CNeT was trained to separate two nested spirals. This is a nontrivial benchmark problem used extensively for evaluating neural architectures and supervised learning schemes. A nice feature of the double spiral problem is that its difficulty can be adjusted easily. The partition of the input space becomes an increasingly difficult problem as the spirals are extended by more rotations. Another advantage of the double spiral problem is that the partition of the input space can be visualized on paper since the input space is two-dimensional (2-D). Fig. 3(a) shows two nested spirals, each containing 128 points. The two spirals have the same center but do not intersect. The white and black points in Fig. 3(a) represent two classes. These points were used as training examples in the experiments. Fig. 3(b) shows the desired generalization. The desired generalization was computed by classifying each point $\mathbf{x}$ of the plane as belonging to the class $\ell(\mathbf{x}_i)$, where $\mathbf{x}_i \in \mathcal{X}$ is the example closest to $\mathbf{x}$.

A CNeT was trained with the examples using the global(3) search method. In these experiments, the tree was binary ($m = 2$), the maturity age was set to $\tau = 512$, the initial learning rate was $\alpha_0 = 0.1$ and the decay of the learning rate was $\alpha_d = 0.01$. Fig. 4(a)–(d) show the response of the tree as it grows. In the beginning, the input space is split by the root node into two regions (not shown). Splitting again yields four regions shown in Fig. 4(a). In the left part of the figure the regions are grayed as follows: For every point $\mathbf{x}$ in the input space, the recall procedure finds a prototype $\mathbf{v}_k$ close to it. The gray level of the pixel at the position $\mathbf{x}$ is proportional to the ratio between the examples corresponding to white and black points used to update $\mathbf{v}_k$ during learning.

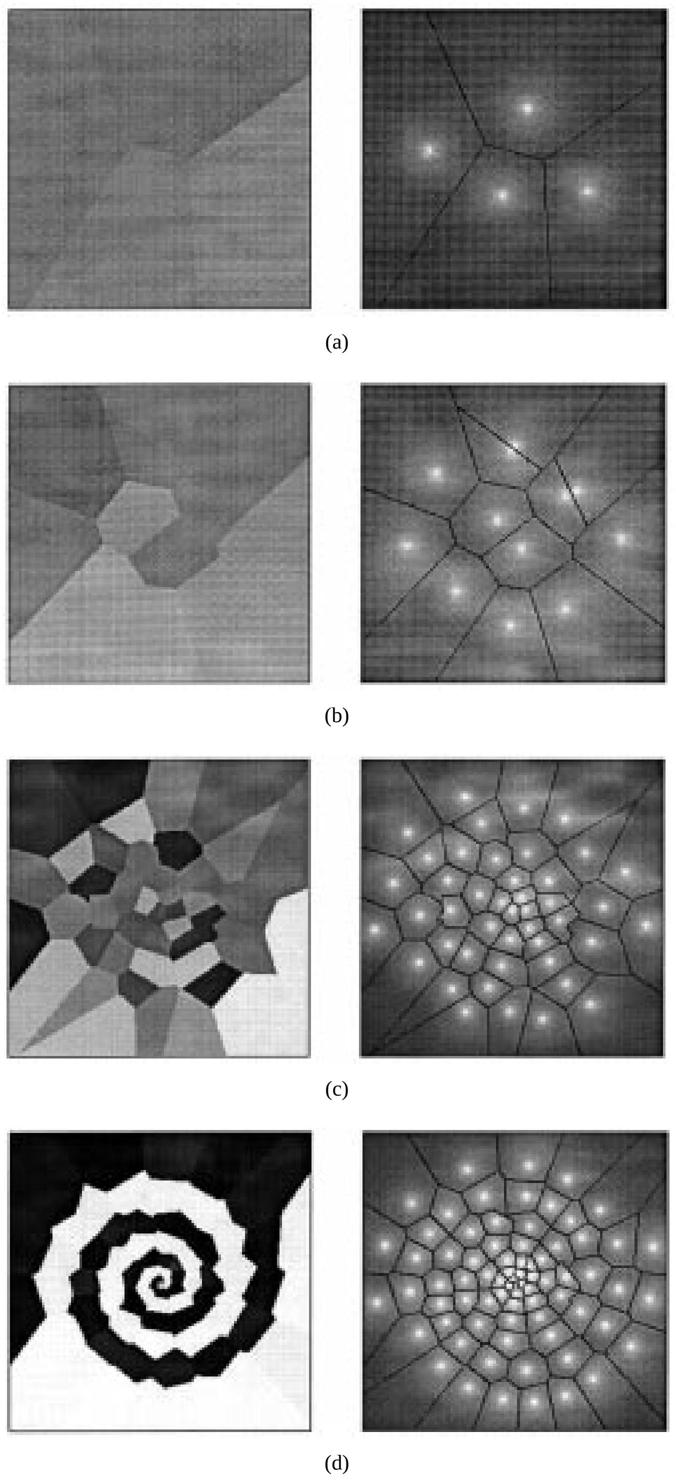


Fig. 4. Progress of training: Response of the CNeT after (a) 4, (b) 16, (c) 64, and (d) 256 adaptation cycles.

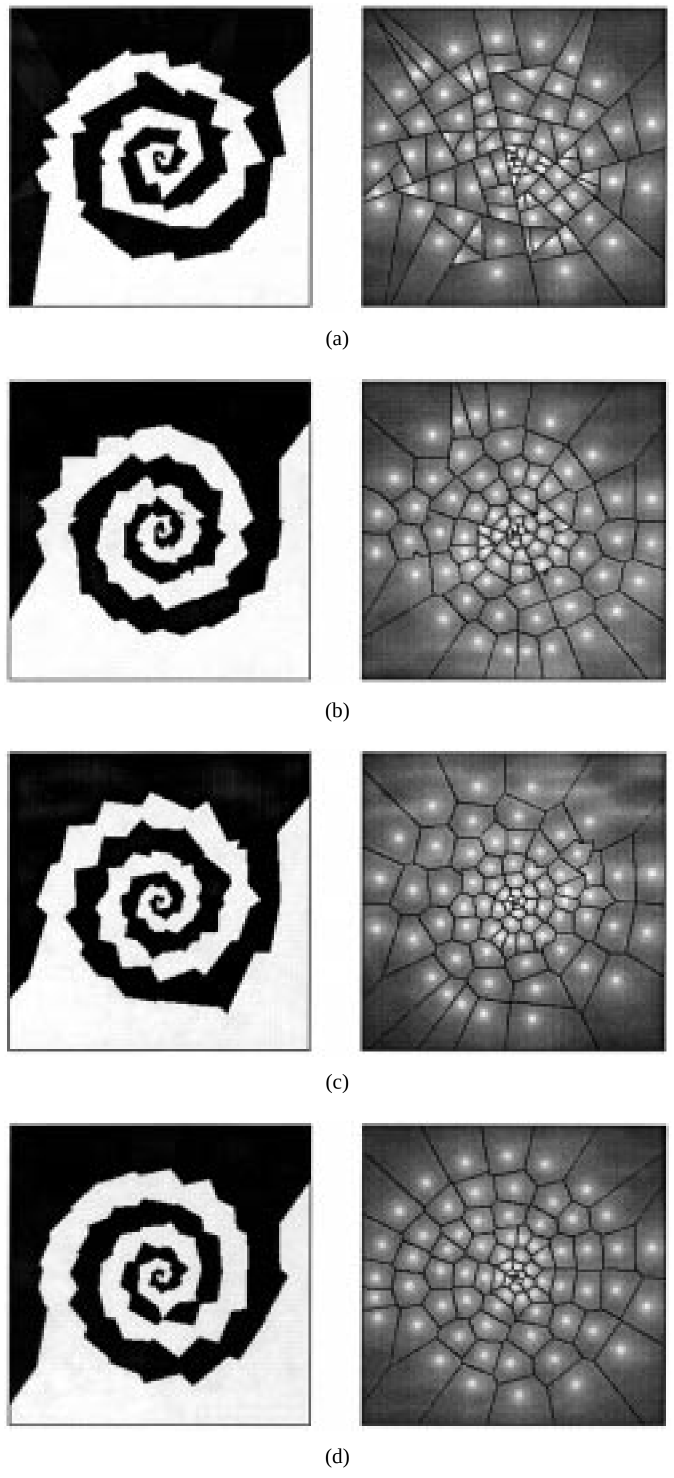
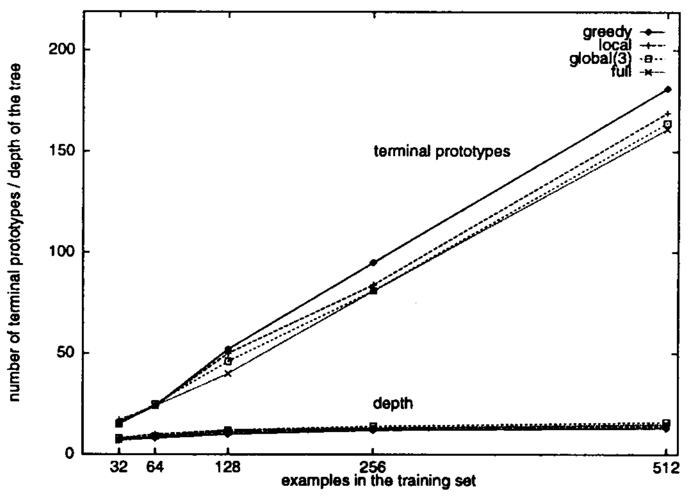


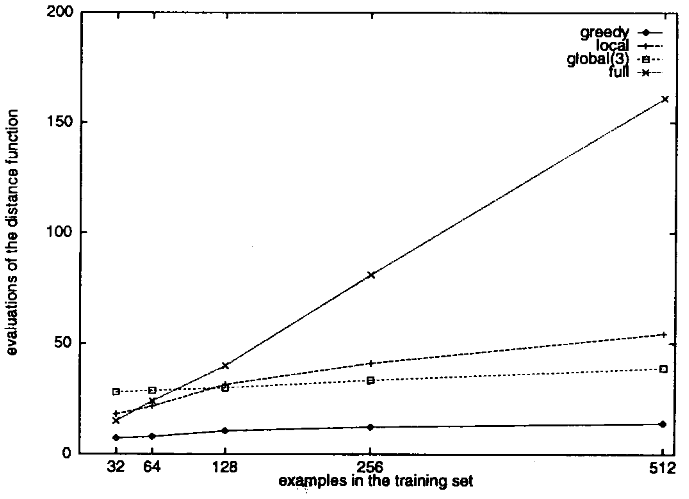
Fig. 5. Response of the CNeT trained using: (a) the greedy search method, (b) the local search method, (c) the global(3) search method, and (d) the full search method.

The right part of the figure shows the centroids of the clusters as bright spots and the borders between regions as black lines. In this case, the pixels are grayed according to the distance $d(\mathbf{v}_k, \mathbf{x})$. Fig. 4(b)–(d) show the progress of the training. The clusters are split to cover the input space in more and more detail. According to Fig. 4(d), all examples from the training set are classified correctly after 256 adaptation cycles. Further training will only enhance contrast.

The second set of experiments investigated the influence of the search method on the performance of the CNeT. The parameter setting for this series of experiments was the same as above. Fig. 5(a), (b), (c), and (d) shows the separation of the 2-D input space produced by four CNeT's trained using the greedy, local, global(3), and full search methods, respectively. Clearly, all trained CNeT's were able to classify all the training examples correctly. The major performance difference due to



(a)



(b)

Fig. 6. Effect of the problem size (number of examples in the training set) on (a) the size of the tree (number of treminal prototypes/depth) grown by different search methods and (b) the computational complexity (evaluations of the distance function) of different search methods.

the search method was the generalization ability of the trained CNeT's, that is, their response to feature vectors not included in the training set. According to Fig. 5, the CNeT trained using the full search method achieved the closest generalization to the desired one. The search methods can be rated in terms of their generalization ability as: greedy, global(3), local, and full. Given the computational complexity of the search methods, there is a tradeoff between computational complexity of the search method used for training the CNeT and the resulting generalization ability.

The third set of experiments investigated the effect of the problem size on the size of the grown tree and the computational complexity of different search methods. The parameter setting for this series of experiments was the same as above. Fig. 6(a) shows the depth of the fully grown tree and the number of terminal prototypes as the number of training examples increased from 32 to 512. The size of the fully grown CNeT depends on the complexity of the problem. The more difficult the problem is, the bigger the tree grows. According to Fig. 6(a), the number of terminal prototypes grows linearly with the problem size while the depth of the tree grows only logarithmically. This means that the grown tree is balanced to some extent. Fig. 6(a) also indicates that the search method has only a slight influence on the size of the tree. Nevertheless, the greedy search method grows slightly bigger trees than the other three search methods. The influence of the different search methods on the computational complexity is shown in Fig. 6(b) for different problem sizes. The computational complexity is measured in terms of the average number of evaluations of the distance function $d(.,.)$ for recall on a fully grown tree. This is justified by the fact that the evaluation of the distance, which is the only vector operator, is computationally more expensive than the remaining scalar operators. The number of scalar operations needed is linear in the number of evaluations of the distance function. As expected, the recall times grow with the problem size. For the full search method the growth is linear. The local method grows sublinearly. The greedy method has the smallest recall times, since its growth is only logarithmic. The recall time for the global(3) method also grows logarithmically. However, the recall times of the global(3) method are longer by a constant factor than the recall times for the greedy method.

B. The IRIS Data Set

The CNeT was also tested using Anderson's IRIS data set [1], which has extensively been used for evaluating the performance of pattern classification algorithms. This data set contains 150 feature vectors of dimension four which belong to three physical classes representing different IRIS subspecies. Each class contains 50 feature vectors. One of the three classes is well separated from the other two, which are not easily separable due to the overlapping of their convex hulls. The performance of the algorithms is evaluated by counting the number of classification errors, i.e., the number of feature vectors that are assigned to a wrong physical cluster. The available 150 examples were split into two sets $\mathcal{X}$ and $\mathcal{X}'$, each containing 75 examples. The examples were randomly assigned to a training set and a testing set. This assignment was different for each experiment. A CNeT was grown with the training set $\mathcal{X}$ and its generalization ability was evaluated using the testing set $\mathcal{X}'$.

A binary tree was grown using the global(3) search method and a maturity age of $\tau = 256$. Fig. 7 shows the number of feature vectors from the training and testing sets classified to a wrong class while the tree was grown. Not surprisingly, the CNeT performed better on the training set after the completion of almost every adaptation cycle. After a certain adaptation cycle, the number of incorrect classifications on the testing set remained almost constant with some fluctuations. On the other hand, the number of classification errors on the training set reduced further as the tree kept growing. This is an indication of overtraining, which can be avoided by using the testing-set stopping criterion to terminate the training as shown in Fig. 7.

Because of its properties, the IRIS data set can be used to illustrate the learning and pruning procedures described in this paper. Fig. 8 shows a grown and pruned CNeT. The bars

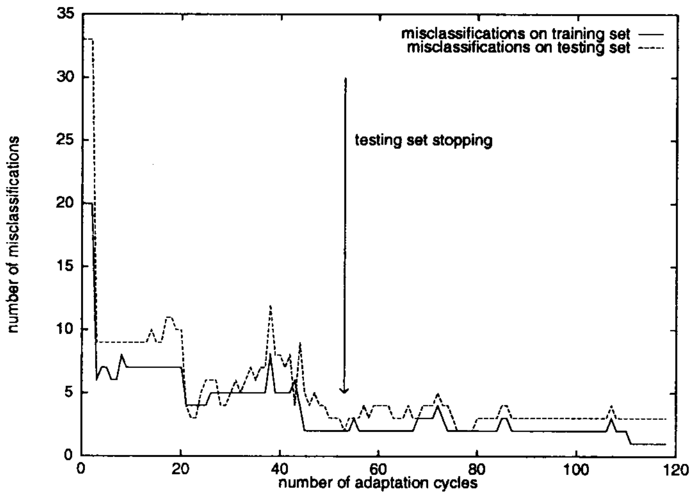


Fig. 7. Training a CNeT on the IRIS data set: Number of feature vectors from the training and testing sets classified incorrectly by a CNeT as a function of the number of adaptation cycles.

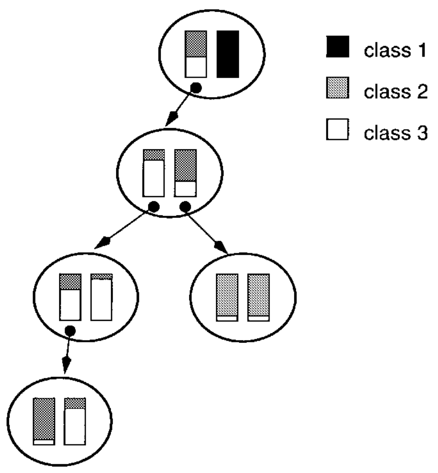
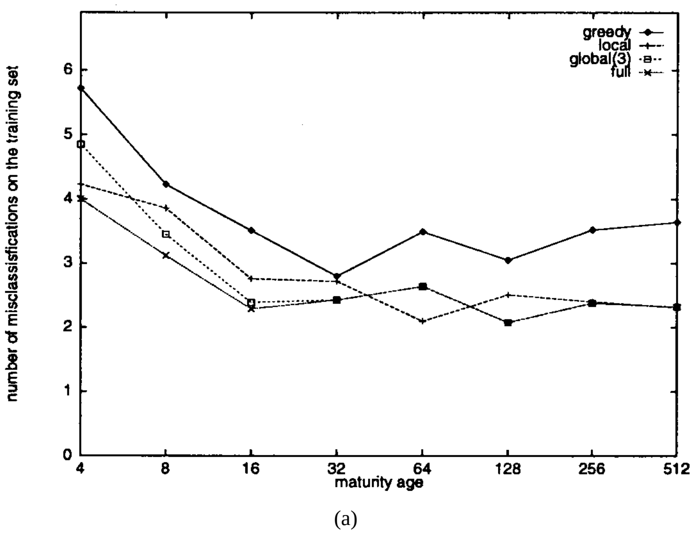


Fig. 8. Grown CNeT trained to classify the IRIS data: The bars show how often the prototypes in each node were updated to match feature vectors from the three classes.



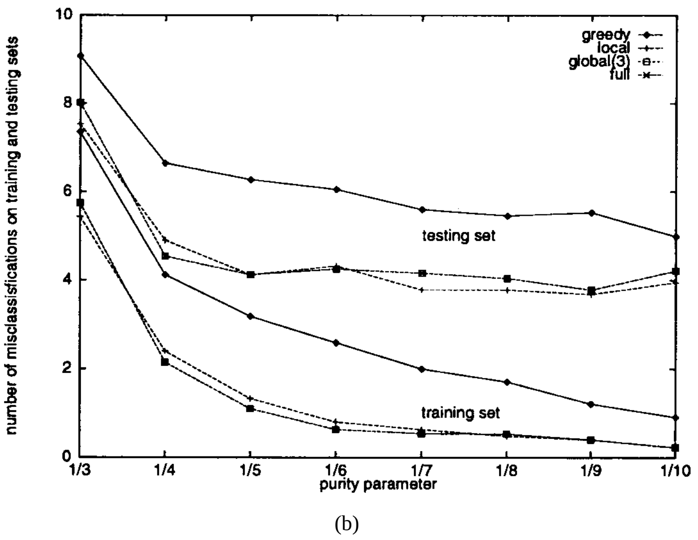


Fig. 9. Effect of the maturity age τ and the purity parameter β on the performance of a CNeT trained to classify the IRIS data: Number of feature vectors classisfied incorrectly by a CNeT as a function of (a) the maturity age τ of the nodes and (b) the purity parameter β.

within the nodes correspond to the prototypes of the tree and show how often each prototype was updated to match input vectors belonging to the three classes. In full consistency with the nature of the IRIS data, the first class was separated from the other two by the root of the tree. The rest of the tree was grown to separate the feature vectors belonging to the other two classes.

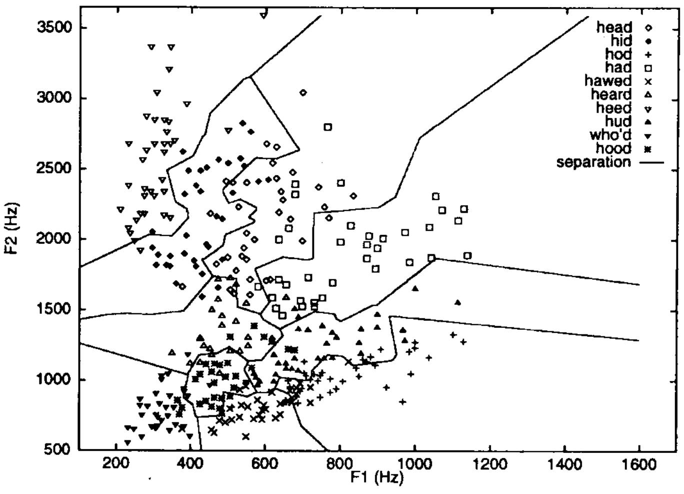
The last set of experiments investigated the effect of the maturity age τ and the purity parameter β on the performance of the CNeT trained on the IRIS data. Each experiment was repeated 100 times in order to minimize the effect of the random selection of the training and testing examples. The number of incorrect classifications shown in Fig. 9 was obtained by averaging the results of all these experiments. Fig. 9(a) shows the number of feature vectors from the training set classified incorrectly by various CNeT's trained using different search methods. The maturity age τ varied from 4 to 512, while the training was terminated according to the testing-set stopping criterion. Although the maturity age affected significantly the speed of training, it had no significant influence on the performance of the trained CNeT. In contrast, the performance of the CNeT was affected by the search method employed. Compared with other search methods, the greedy search method resulted in a relatively large number of classification errors. For high values of the maturity age, there was no significant differences between the performance of the CNeT's trained using the other three search methods. Fig. 9(b) shows the number of feature vectors from the training and testing sets that were classified incorrectly by various CNeT's trained using different search methods. The training was terminated using the maturity stopping criterion while the purity parameter β decreased from 1/3 to 1/10. Regardless of the search method used, the performance of the CNeT's on the training set improved as the value of the purity parameter decreased. In contrast, the number of feature vectors from the testing set assigned by the CNeT to a wrong class remained almost constant for values of β lower than 1/4. In some cases, there was even an increase of the number of incorrect classifications on the testing set. This is an indication of overtraining. Note that because of the tree architecture used, the number of incorrect classifications on the training set can easily be reduced to zero by choosing smaller values of β. However, such a choice would allow the creation of new nodes and degrade the generalization ability of the CNeT.

C. Two-Dimensional Vowel Data

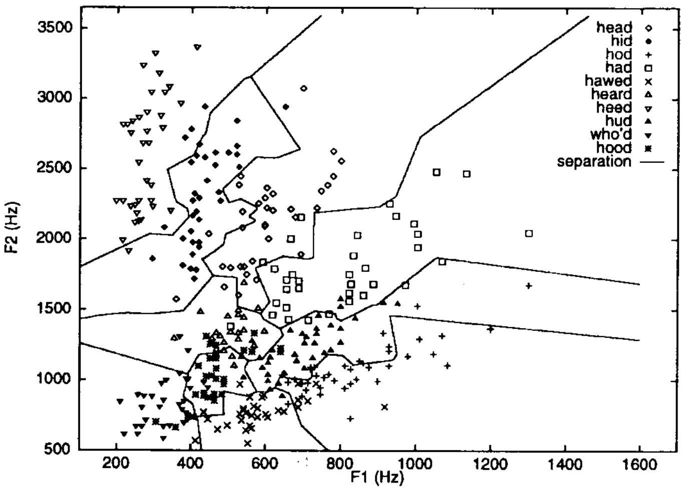
The performance of the CNeT was evaluated using a set of 2-D vowel data formed by computing the first two formants F1 and F2 from samples of ten vowels spoken by 67 speakers [20], [21]. This data set has been extensively used to compare different pattern classification approaches because there is significant overlapping between the points corresponding to different vowels in the F1-F2 plane [20], [21]. The available 671 feature vectors were divided into a training set, containing 338 vectors, and a testing set, containing 333 vectors.

The training set formed from the 2-D vowel data was used to train a CNeT to classify the ten vowels. The CNeT was grown using the global(3) search method and training was terminated according to the testing-set stopping criterion. The purity parameter was $\beta = 1/4$, the maturity age was $\tau = 1024$, and the learning parameters involved in the update of the prototypes were $\alpha_0 = 0.02$ and $\alpha_d = 0.002$. Fig. 10 shows the partition of the feature space into regions that correspond to one vowel each that was produced by the trained CNeT. In addition to the regions formed by the CNeT, Fig. 10(a) and (b) also show the training and testing data, respectively. The regions in Fig. 10 are composed of the cells of the terminal prototypes that correspond to the same class. Therefore, the boundaries of the regions are piecewise linear. Due to the extensive overlapping of the vowel classes, it is not possible to find a partition that classifies correctly all examples from the training and testing sets [20], [21]. It is clear from Fig. 10 that the CNeT attempts to find the best possible compromise in regions of the input space with extensive overlapping between the classes. This is accomplished through the stopping criterion employed, which forces the CNeT to accept some incorrect classifications of examples from the training set in order to produce a partition of the feature space that performs well on the testing set. This improves the generalization ability of the CNeT, as indicated by Fig. 10(a) and (b).

The recall strategy employed in the previous experiment assigned a class label to each input example using only its closest terminal prototype. The next experiment was based on the same trained CNeT but a slightly modified recall strategy. More specifically, a class label was assigned to each input example by interpolating between its two closest terminal prototypes. The partition of the feature space produced by this modified recall strategy is shown in Fig. 11. It is clear from Fig. 11 that the resulting partition is similar to that shown in Fig. 10 but the boundaries of the regions produced by the modified recall strategy are not necessarily piecewise linear. Since this modified strategy resulted in improved performance on both training and testing sets, it was also used in the experiments which follow.



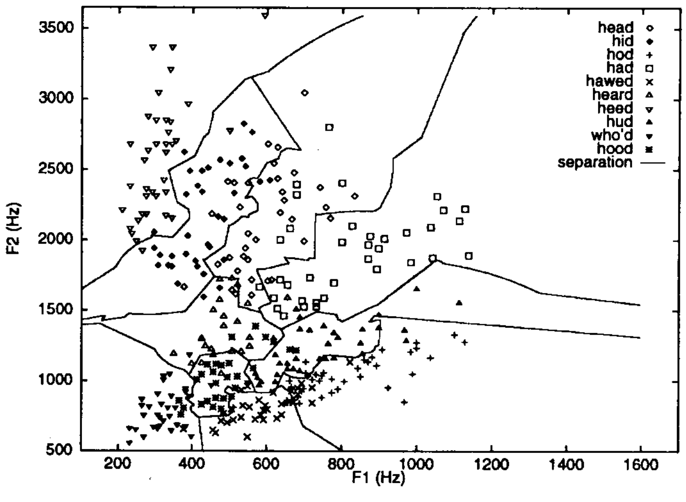
(a)



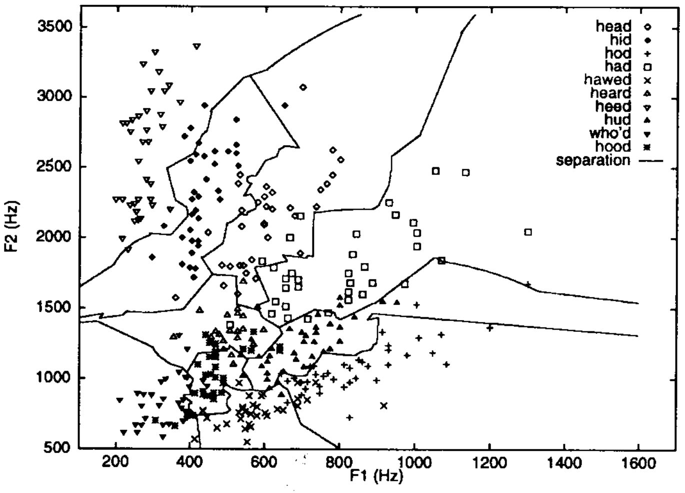
(b)

Fig. 10. Classification of the 2-D vowel data produced by the CNeT shown together with (a) the training data set and (b) the testing data set. The recall of the CNeT was based only on the best prototype found.

The next set of experiments evaluated the effect of the combination of splitting and stopping criteria on the performance and size of the CNeT grown using different search methods. Fig. 12(a) shows the number of feature vectors from the training and testing sets classified incorrectly by the CNeT's trained and grown using different values of the purity parameter β. The number of terminal prototypes and the depth of the CNeT's grown in these experiments are shown as a function of β in Fig. 12(b). The curves shown in Fig. 12 were obtained when the CNeT was grown until all of its nodes were mature (maturity stopping criterion). Since the purity parameter β determines the impurity of a slot that is tolerated before the slot is split, the acceptable number of incorrect classifications on the training set decreases as the value of β decreases. According to Fig. 12(a), the misclassification rate on the training set decreased to 0 as β decreased from 0.5 to 0.1. However, the performance of the CNeT on the testing set did not improve or even degraded as β decreased below 0.4. This indicates the beginning of overtraining. For low values

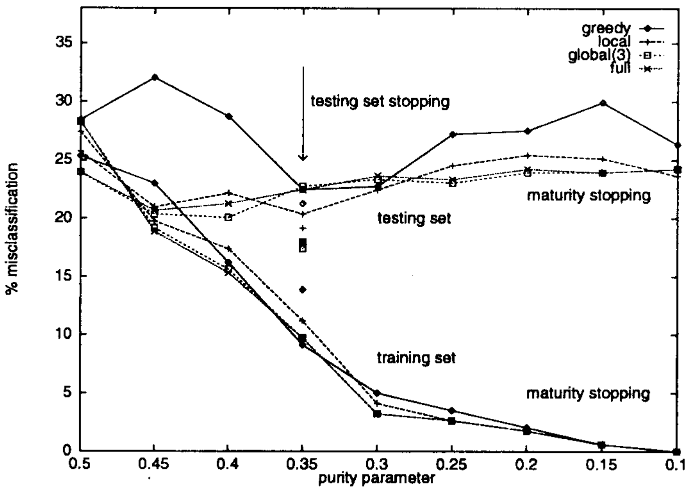


(a)

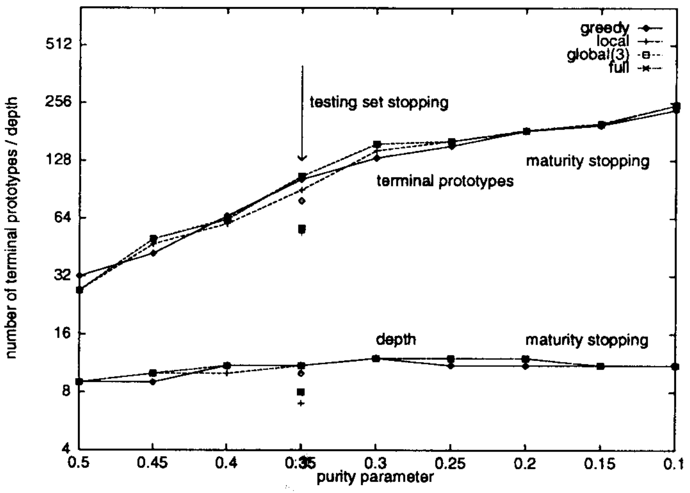


(b)

Fig. 11. Classification of the 2-D vowel data produced by the CNeT shown together with (a) the training data set and (b) the testing data set. The recall of the CneT was based on the interpolation between the two best prototypes found.



(a)



(b)

Fig. 12. Effect of the purity parameter β on the classification of the 2-D vowel data: (a) percentage of misclassified examples and (b) size of the CNeT (number of terminal prototypes/depth).

of β the CNeT memorizes the examples from the training set, which degrades its generalization ability. Comparison of the performance of different search methods indicates that only the greedy search method deviated significantly from their average behavior. In fact, the performance of the greedy search method on the testing set was close to the average performance of all four search methods for values of β between 0.35 and 0.3. As the value of β decreased, the size of the trees increased. According to Fig. 12(b), the depth of the tree increased only slightly despite the substantial growth in the number of terminal prototypes. This indicates that the tree is well balanced. The CNeT was also grown using the same search method but a different stopping criterion. In this case, the testing-set stopping criterion was used to prevent the tree from growing too large. It was experimentally verified that the selection of the purity parameter β did not affect the growth of the CNeT when the tree was pruned based on its performance on the testing set. Fig. 12 shows the performance

TABLE V
PERFORMANCE OF DIFFERENT CLASSIFIERS ON THE 2-D VOWEL DATA: PERCENTAGE OF EXAMPLES FROM THE TRAINING AND TESTING SETS CLASSIFIED INCORRECTLY BY THE CNeT, A k-NN CLASSIFIER, AND TWO FFNN'S

	% misclassified	
Classifier	training set	testing set
CNeT	17.40	17.96
k-NN		24.55
FFNN (5 hidden units)	20.41	23.42
FFNN (10 hidden units)	20.12	19.82

and size of the CNeT grown for $\beta = 0.35$. Fig. 12(a) indicates that the CNeT grown using the testing-set stopping criterion resulted in fewer misclassifications on the testing set, which implies improved generalization ability. According to Fig. 12(b), this stopping criterion resulted in a pruned tree which has a smaller depth and contains a smaller number of

TABLE VI
PERFORMANCE OF THE CNeT ON THE 2-D VOWEL DATA WHEN REJECTIONS WERE ALLOWED: PERCENTAGE OF RECOGNIZED, SUBSTITUTED, AND REJECTED EXAMPLES FROM THE TRAINING AND TESTING SETS FOR DIFFERENT VALUES OF THE REJECTION PARAMETER r

	training set			testing set		
r	%recognized	%substituted	%rejected	%recognized	%substituted	%rejected
0.5	79.05	16.81	4.12	79.04	17.06	3.89
0.6	76.99	10.02	12.97	71.55	12.27	16.16
0.7	62.24	5.01	32.74	60.77	7.78	31.43
0.8	42.18	1.76	56.04	43.11	3.89	52.99
0.9	35.69	0	64.30	34.73	1.19	64.07

terminal prototypes compared to that grown using the same value of β and the maturity stopping criterion.

Table V shows the percentage of the examples from the training and testing sets misclassified by the CNeT grown by the global(3) search method, the k-*nearest neighbor* (k-NN) classifier, and two *feedforward neural networks* (FFNN's) with five and ten hidden units trained using gradient descent. The k-NN classifier uses feature vectors from the training set as a reference to classify examples from the testing set. Given an input example from the testing set, the k-NN computes its Euclidean distance from all the examples included in the training set. The closest k examples from the training set are selected and the input example is classified on the basis of their labels as follows: if all the labels are not identical, then the example is *rejected*. If all the labels are identical, then the example is classified accordingly. More specifically, the example is *recognized* if the label assigned by the classifier and the actual label are identical or *substituted* if the assigned and actual labels are different. The k-NN classifier was tested in these experiments with $k = 1$, which implies that none of the examples was rejected. For the k-NN classifier, Table V shows only the misclassifications on the testing set since the training set is only used as a reference. The CNeT performed better than the other classifiers on both training and testing sets. It is also remarkable that the CNeT classified incorrectly the same percentage of examples from the training and testing sets, which implies that pruning resulted in a tree of very satisfactory generalization ability. Both trained FFNN's classified incorrectly the same percentage of examples from the training set but increasing the number of hidden units from five to ten improved the performance on the testing set. Nevertheless, it was experimentally verified that increasing the number of hidden units above ten degraded the ability of the trained FFNN's to generalize.

The CNeT grown using the global(3) search method was allowed to reject some ambiguous examples in order to improve its classification accuracy. Table VI shows the percentage of recognized, substituted, and rejected examples from the training and testing sets for different values of the rejection parameter r. As the value of r increased from 0.5 to 0.9, the percentage of substituted examples from both training and testing sets decreased with a simultaneous increase in the percentage of rejected examples. It is clear from Table VI that there is a tradeoff between the reliability of the classifier (measured by the percentage of substituted examples) and its confidence (measured by the percentage of accepted examples). For example, when $r = 0.9$ the percentage of substituted examples from the training and testing sets reduced to 0% and almost 1%, respectively. However, in this case the classifier rejects more than 2/3 of the examples from both sets.

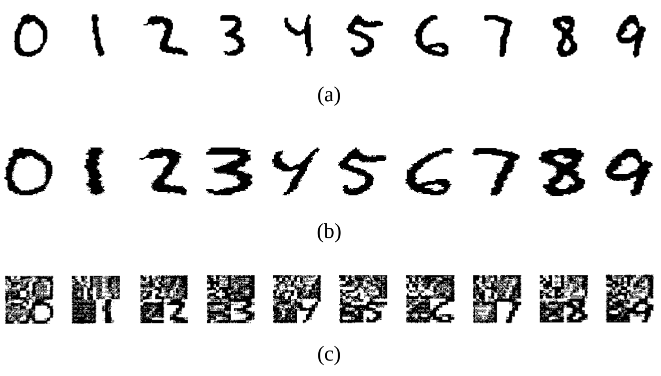


Fig. 13. Digits from the NIST database: (a) ordinary binary images, (b) 32 × 32 binary images after one stage of preprocessing (slant and size normalization), and (c) 16 × 16 images of the digits after two stages of perprocessing (slant and size normalization followed by wavelet decomposition).

D. NIST Digits

In the last set of experiments, the CNeT was tested and compared with competing techniques on a large-scale handwritten digit recognition problem. The objective of a classifier in this application is the recognition of the digit represented by a binary image of a handwritten numeral. Recognition of handwritten digits is the key component of automated systems developed for a great variety of real-world applications, including mail sorting and check processing. Automated recognition of handwritten digits is not a trivial task due to the high variance of handwritten digits caused by different writing styles, pens, etc. Thus, the development of a reliable system for handwritten digit recognition requires large databases containing a great variety of samples. Such a collection of handwritten digits is contained in the *NIST Special Databases 3*, which was used as a data source in these experiments. The NIST database contains 120 000 isolated binary digits that have been extracted from handwriting sample forms. These digits were handwritten by about 2100 field representatives of the United States Census Bureau. The isolated digits were scanned to produce binary images of size 40 × 60 pixels, which are centered in a 128 × 128 box. Fig. 13(a) shows some sample digits from 0 to 9

TABLE VII
PERFORMANCE OF THE k-NN CLASSIFIER ON THE TESTING SET FORMED FROM THE NIST DATA FOR DIFFERENT DIGIT REPRESENTATIONS

	32×32 digit representation (one preprocessing stage)			16×16 digit representation (two preprocessing stages)		
k	%recognized	%substituted	%rejected	%recognized	%substituted	%rejected
1	97.35	2.64	0.00	97.71	2.28	0.00
2	95.80	1.05	3.14	96.47	1.13	2.39
4	93.35	0.57	6.06	94.38	0.60	5.01
8	89.68	0.33	9.98	91.27	0.31	8.40
16	84.01	0.16	15.81	86.27	0.17	13.54

from the NIST database used in these experiments. The data set was partitioned in three subsets as follows: 58 646 digits were used for training, 30 367 digits were used for testing, and the remaining 30 727 digits constituted the validation set.

The raw data from the NIST database were preprocessed in order to reduce the variance of the images that is not relevant to classification. The first stage of the preprocessing scheme produced a slant and size normalized version of each digit. The slant of each digit was found by first determining the center of gravity of each digit, which defines an upper and lower half of it. The centers of gravity of each half were subsequently computed and provided an estimate of the vertical main axis of the digit. This axis was then made exactly vertical using a horizontal shear transformation. In the next step, the minimal bounding box was determined and the digit was scaled into a 32×32 box. This scaling may slightly distort the aspect ratio of the digits by centering, if necessary, the digits in the box. Fig. 13(b) shows the same digits shown in Fig. 13(a) after slant and size normalization.

The second preprocessing stage involved a four-level wavelet decomposition of the 32×32 digit representation produced by the first preprocessing stage. Each decomposition level includes the application of a 2-D Haar wavelet filter in the decomposed image, followed by downsampling by two along the horizontal and vertical directions. Because of downsampling, each decomposition level produces four subbands of lower resolution, namely a subband that carries background information (containing the low-low frequency components of the original subband), two subbands that carry diagonal details (containing low-high and high-low frequency components of the original subband), and a subband that carries the nondiagonal details (containing the high-high frequency components of the original subband). As a result, the four-level decomposition of the original 32×32 image produced three subbands of sizes 16×16, 8×8, and 4×4, and four subbands of size 2×2. The 32×32 image produced by wavelet decomposition was subsequently reduced to an image of size 16×16 by representing each 2×2 window by the average of the four pixels contained in it. This step reduces the amount of data by 3/4 and has a smoothing effect that suppresses the noise present in the 32×32 image. Fig. 13(c) shows the images representing the digits shown in Fig. 13(a) and (b), resulting after the second preprocessing stage described above.

The representation of the digits produced by the wavelet decomposition involved in the second preprocessing stage was evaluated by using a standard classification method. More specifically, the k-NN classifier was tested on the 32×32 digit representation resulting from the first preprocessing stage and the 16×16 digit representation resulting after wavelet decomposition and averaging. Table VII summarizes the results of the k-NN classifier used on the results of the two stages of preprocessing for different values of k. Table VII indicates that there is a tradeoff between the reliability of classification and the rejection rates, which mainly depends on the value of k used. As the value of k increases, the substitution rate decreases but more digits are rejected. It is clear from Table VII that the k-NN classifier performed better when classification was based on the results of the wavelet decomposition instead of the 32×32 representation of the digits produced by the first preprocessing stage. An additional advantage of using the second preprocessing stage is that the resulting feature vectors need roughly 1/4 of the memory and computational time required when the feature vectors are the result of the first preprocessing stage.

The next experiment illustrates the operation of a search method applied to a CNeT trained to recognize the NIST digits using the feature vectors obtained after the two preprocessing stages. The CNeT was trained and grown using the global(2) search method. The purity parameter was $\beta = 1/32$ and the maturity age was $\tau = 512$. The operation of the search method is described by Fig. 14. Fig. 14(e) shows the 8×8 lower-right quadrant of the 16×16 representation of the digit "1" produced after two stages of preprocessing. Fig. 14(a) (b), (c), and (d) shows the parts of the tree visited by the global(2) search method after initialization, one training cycle, two training cycles, and four training cycles, respectively. Fig. 14(a)–(d) shows only the prototypes whose Euclidean distances from the input example $\mathbf{x}$ are calculated. As a result, Fig. 14(a)–(d) shows four prototypes per level except of the two prototypes shown in the first level (initialization). The small plots next to some of the prototypes represent their normalized histograms. The horizontal axis of each histogram corresponds to the digits $0, 1, \cdots, 9$. The locations of the vertical bars correspond to the digits represented by each prototype while their heights represent the likelihood of this representation. The distribution of the bars over the horizontal axis reveals the purity of each prototype. The three dots shown

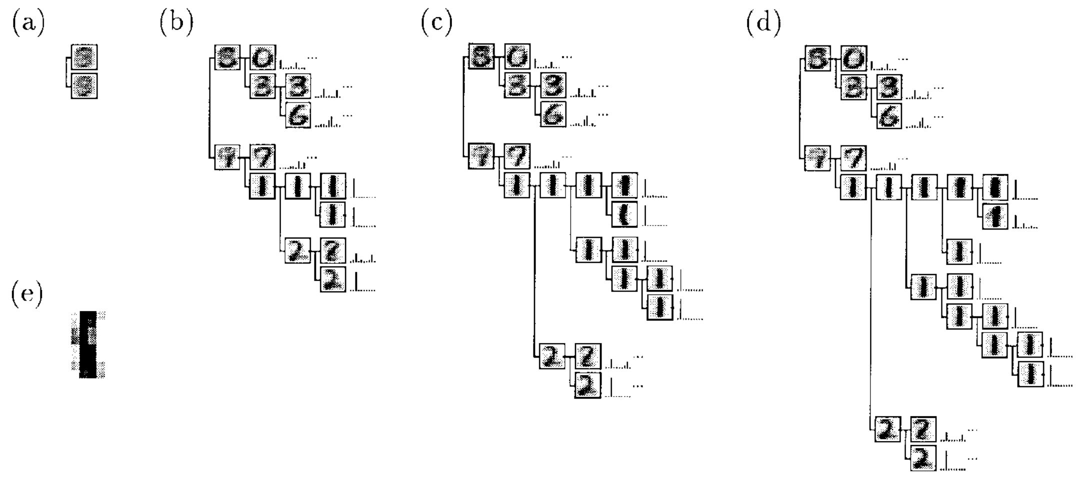


Fig. 14. The parts of the CNeT visited by the global(2) earch method after (a) initialization, (b) one training cycle, (c) two training cycles, and (d) four training cycles when the example (e) was presented to the tree.

TABLE VIII
PERFORMANCE OF THE CNeT ON THE NIST DIGITS FOR DIFFERENT SEARCH WIDTHS OF THE GLOBAL (ω) SEARCH METHOD WHEN NO REJECTIONS WERE ALLOWED

	training set		testing set		validation set	
ω	%recognized	%substituted	%recognized	%substituted	%recognized	%substituted
8	99.71	0.29	97.53	2.47	97.72	2.28
16	99.73	0.27	97.63	2.37	97.83	2.17
32	99.74	0.26	97.69	2.31	97.90	2.10

in the right-hand side of some histograms indicate that there is a subtree (not shown in the figure) whose root is the prototype next to the histogram. Fig. 14 describes the growth and training of the CNeT by showing how the prototypes become tuned to some examples as training progresses. Moreover, Fig. 14 shows that the search method visits only the parts of the tree that contain prototypes similar to the input example.

A CNeT was trained to classify the feature vectors resulting from the NIST digits after the two stages of preprocessing digits using the global(ω) search method. The purity parameter was $\beta = 1/32$, the maturity age was $\tau = 256$, and the learning parameters involved in the update of the prototypes were $\alpha_0 = 0.05$ and $\alpha_d = 0.005$. The CNeT was pruned for improved performance on the testing set by terminating its growth and training using the testing-set stopping criterion. Recall was performed by interpolating between the two closest prototypes to the input example. For $\omega = 8$, the maximum depth of the grown CNeT was 24. The CNeT consisted of 10 059 terminal prototypes, which indicates that the tree was well balanced. The effect of the search method on the success rate of the CNeT was evaluated in the case where the classifier was not allowed to reject any input examples. Table VIII summarizes the performance of the CNeT trained and grown using the global(ω) search method when the search width varied from $\omega = 8$ to $\omega = 32$. As ω increases, the global(ω) method searches a larger subtree and is expected to return more often a terminal prototype that is the closest to the input example. This is consistent with the results summarized in Table VIII, which indicates that the performance of the CNeT improved consistently as the value of ω increased from 8 to 32. For $\omega = 32$, the CNeT classified correctly 99.71% of the digits from the training set, 97.69% of the digits from the testing set, and 97.90% of the digits from the validation set. The CNeT performs better than various classification schemes tested on the same data set [9], none of which exceeded the recognition rate of 97.5%. The CNeT is also a strong competitor to FFNN's tested on this data set. Table IX shows the percentage from the training, testing, and validation sets recognized or substituted by three FFNN's with 16, 32, and 64 hidden units trained using gradient descent. The best performance was achieved by the FFNN with 64 hidden units. This network performed better than the CNeT on the training set. However, its performance was slightly inferior to that of the CNeT on the testing and validation sets. After their training, the FFNN and CNeT required roughly the same time for recall. However, the training of the FFNN's was approximately ten times longer than the growth and training of the CNeT's. The performance of the CNeT is also close to that of the k-NN classifier, which classified correctly 97.79% of the digits from the testing set and 97.98% of the digits from the validation set. In addition to its competitive classification accuracy, the CNeT was computationally more efficient than the k-NN. More specifically, the computational time required by the CNeT to perform this classification task was two orders of magnitude lower than that required for the same task by the k-NN. This is due to the fact that the CNeT computes only

TABLE IX
PERFORMANCE OF FFNN'S WITH VARIOUS NUMBERS OF HIDDEN UNITS (N_h) ON THE NIST DIGITS

	training set		testing set		validation set	
n_h	%recognized	%substituted	%recognized	%substituted	%recognized	%substituted
16	84.14	15.86	86.46	13.54	86.88	13.12
32	96.45	3.55	95.95	4.05	96.35	3.65
64	99.99	0.01	97.33	2.67	97.54	2.46

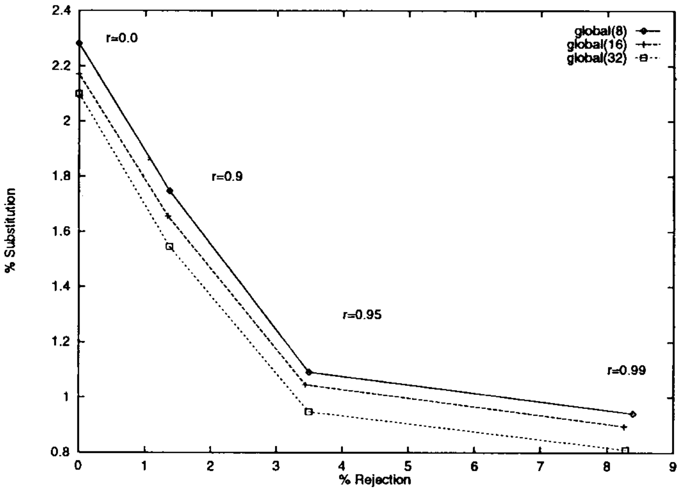


Fig. 15. Recall of the CNeT using the global (ω) for the validation set formed from the NIST digits: The substitution rates are plotted against the rejection rates for different values of the rejection parameter r and different search widths.

the Euclidean distances between the input example and a small portion of the prototypes stored in the CNeT. The recall speed of the CNeT employing the global(8) search method was 85 digits per CPU-second on a SUN SPARC-20 workstation. This high recall speed makes the implementation of the CNeT on serial hardware feasible. On the other hand, the application of the k-NN in real-time applications requires special purpose parallel hardware.

The last set of experiments investigated how the reliability of the CNeT improves if some ambiguous examples are rejected. The digits from the validation set were classified by the CNeT trained and grown using the global(ω) search method with $\omega = 8$, $\omega = 16$, and $\omega = 32$. Fig. 15 plots the substitution rates against the rejection rates obtained for different values of the rejection parameter r. The curves indicate that the development of more reliable classifiers can be accomplished by allowing the CNeT to reject a portion of the examples. As the rejection parameter r increases from zero to 0.99, the substitution rate decreases. The price to be paid for improved reliability is an increase in the rejection rate. The most substantial decrease in the substitution rate was encountered when the value or r decreased from zero to 0.95. For values of r above 0.95 the substitution rate decreased only slightly despite the substantial increase in the rejection rate. Regardless of the value of the rejection parameter r, the performance of the CNeT improved as ω increased from 8 to 32. For all values of r used in these experiments, the three CNeT's rejected practically the same number of examples but the substitution rate decreased as the search width increased from eight to 32.

VII. CONCLUSIONS

This paper introduced competitive neural trees for pattern classification. The CNeT combines the advantages of competitive neural networks and decision trees. The CNeT performs hierarchical clustering by employing competitive unsupervised learning at the node level. The generalization ability of the CNeT is guaranteed by forward pruning, which is an inherent part of the learning process. The main advantage of the CNeT is its structured, self-organizing architecture that allows for short learning and recall times.

The experiments evaluated the proposed learning and recall procedures and different search methods. The experiments also investigated the combined effect of splitting and stopping criteria on the generalization ability of trained CNeT's. The performance of a trained CNeT depends on the search method employed in the learning and recall phases. Among the search methods proposed in this paper, the greedy search method achieves the fastest learning and recall at the expense of performance. On the other hand, the full search method achieves the best performance at the expense of learning and recall speed. The local and global search methods combine the advantages of the greedy and full search methods and allow for tradeoff's between speed and performance. The experiments on the data sets containing overlapping classes revealed that the growth of small and balanced trees with satisfactory generalization ability can be accomplished by combining the testing-set stopping criterion with a value of the purity parameter β allowing the existence of slots that are not perfectly pure. The experiments on the NIST digits indicated that the reliability and classification accuracy of the trained CNeT can be improved by a recall strategy that allows the rejection of some ambiguous examples. The CNeT can be trained in a tiny fraction of the time required for training conventional feedforward neural networks to perform the same classification task. As an example, the CNeT was trained to recognize the NIST digits ten times faster than the FFNN, which required four days for training. On the other hand, the generalization achieved by CNeT's was consistently comparable or even better than that of feedforward neural networks. The classification accuracy of the CNeT was also found to be comparable to that of the k-NN classifier. In fact, the k-NN classifier resulted in slightly higher recognition rates on the NIST data because of the large size of the training set. However, the k-NN classifier was computationally more demanding than the CNeT because of the different search strategy employed. The time required by the k-NN to classify an example increased with the problem size (number of examples in the training set). The trained CNeT classified examples much faster than the k-NN classifier, especially in large-scale classification problems such as the recognition of digits from the NIST database.

REFERENCES

[1] E. Anderson, "The IRISes of the Gaspe Peninsula," *Bull. Amer. IRIS Soc.,* vol. 59, pp. 2–5, 1939.

[2] L. Atlas, R. Cole, Y. Muthusamy, A. Lippman, J. Connor, D. Park, M. El-Sharkawi, and R. J. Marks II, "A performance comparison of trained multilayer perceptrons and trained classification trees," *Proc. IEEE,* vol. 78, no. 10, pp. 1614–1619, 1990.

[3] S. Behnke and N. B. Karayiannis, "Competitive neural trees for vector quantization," *Neural Network World,* vol. 6, no. 3, pp. 263–277, 1996.

[4] ——, "CNeT: Competitive neural trees for pattern classification," in *Proc. IEEE Int. Conf. Neural Networks,* Washington, D.C., June 3–6, 1996, pp. 1439–1444.

[5] L. Brieman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees.* Belmont, CA: Wadsworth, 1984.

[6] P. A. Chou, "Optimal partitioning for classification and regression trees," *IEEE Trans. Pattern Anal. Machine Intell.,* vol. 13, pp. 340–354, 1991.

[7] L. Fang, A. Jennings, W. X. Wen, K. Q.-Q. Li, and T. Li, "Unsupervised learning for neural trees," in *Proc. Int. Joint Conf. Neural Networks,* Seattle, WA, July 8–12, 1991, pp. 2709–2715.

[8] K. R. Farrell, R. J. Mammone, and K. T. Assaleh, "Speaker recognition using neural networks and conventional classifiers," *IEEE Trans. Speech Audio Processing, Pt. II,* vol. 2, pp. 194–205, 1994.

[9] P. J. Grother and G. T. Candela, "Comparison of handprinted digit classifiers," Nat. Inst. Standards Technol., Gaithersburg, MD, Tech. Rep. NISTIR 5209, 1993.

[10] H. Guo and S. B. Gelfand, "Classification trees with neural network feature extraction," *IEEE Trans. Neural Networks,* vol. 3, pp. 923–933, 1992.

[11] N. B. Karayiannis, "Learning vector quantization: A review," *Int. J. Smart Eng. Syst. Design,* vol. 1, pp. 33–58, 1997.

[12] ——, "A methodology for constructing fuzzy algorithms for learning vector quantization," *IEEE Trans. Neural Networks,* vol. 8, pp. 505–518, 1997.

[13] N. B. Karayiannis and J. C. Bezdek, "An integrated approach to fuzzy learning vector quantization and fuzzy c-means clustering," *IEEE Trans. Fuzzy Syst.,* vol. 5, pp. 622–628, 1997.

[14] N. B. Karayiannis and P.-I Pai, "A family of fuzzy algorithms for learning vector quantization," in *Intell. Eng. Syst. Through Artificial Neural Networks,* C. H. Dagli *et al.*, Eds., vol. 4. New York, NY: ASME Press, 1994, pp. 219–224.

[15] ——, "Fuzzy algorithms for learning vector quantization," *IEEE Trans. Neural Networks,* vol. 7, pp. 1196–1211, 1996.

[16] T. Kohonen, *Self-Organization and Associative Memory,* 3rd ed. Berlin, Germany: Springer-Verlag, 1989.

[17] ——, "The self-organizing map," *Proc. IEEE,* vol. 78, no. 9, pp. 1464–1480, 1990.

[18] T. Li, L. Fang, and A. Jennings, "Structurally adaptive self-organizing neural trees," in *Proc. Int. Joint Conf. Neural Networks,* Baltimore, MD, June 7–11, 1992, pp. III-329–III-334.

[19] T. Li, Y. Y. Tang, S. C. Suen, L. Y. Fang, and A. J. Jennings, "A structurally adaptive neural tree for the recognition of large character set," in *Proc. IEEE Int. Conf. Pattern Recognition,* Hague, The Netherlands, Aug. 30–Sept. 4, 1992, pp. 187–190.

[20] R. P. Lippmann, "Pattern classification using neural networks," *IEEE Commun. Mag.,* vol. 27, pp. 47–54, 1989.

[21] K. Ng and R. P. Lippmann, "Practical characteristics of neural network and conventional pattern classifiers," in *Advances in Neural Inform. Processing Syst. 3,* R. P. Lippmann *et al.,* Eds. San Mateo, CA: Morgan Kaufmann, 1991, pp. 970–976.

[22] M. G. Rahim, "A neural tree network for phoneme classification with experiments on the TIMIT database," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing,* San Francisco, CA, Mar. 23–26, 1992, pp. 345–348.

[23] ——, "A self learning neural tree network for recognition of speech features," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing,* Minneapolis, MN, Apr. 27–30, 1993, pp. 517–520.

[24] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE Trans. Syst., Man, Cybern.,* vol. 21, pp. 660–674, 1991.

[25] T. D. Sanger, "A tree-structured adaptive network for function approximation in high-dimensional spaces," *IEEE Trans. Neural Networks,* vol. 2, pp. 285–293, 1991.

[26] A. Sankar and R. J. Mammone, "Optimal pruning of neural tree networks for improved generalization," in *Proc. Int. Joint Conf. Neural Networks,* Seattle, WA, July 8–12, 1991, pp. 219–224.

[27] ——, "Speaker independent vowel recognition using neural tree networks," in *Proc. Int. Joint Conf. Neural Networks,* Seattle, WA, July 8–12, 1991, pp. 809–814.

[28] ——, "Growing and pruning neural tree networks," *IEEE Trans. Computers,* vol. 42, pp. 291–299, 1993.

[29] I. K. Sethi, "Entropy nets: From decision trees to neural networks," *Proc. IEEE,* vol. 78, no. 10, pp. 1605–1613, 1990.

[30] ——, "Decision tree performance enhancement using an artificial neural network implementation," in *Artificial Neural Networks and Statistical Pattern Recognition,* I. K. Sethi and A. K. Jain, Eds. Amsterdam, The Netherlands: Elsevier, 1991, pp. 71–88.

[31] I. K. Sethi and G. P. R. Sarvarayudu, "Hierarchical classifier design using mutual information," *IEEE Trans. Pattern Anal. Machine Intell.,* vol. 4, pp. 441–445, 1982.

Sven Behnke received the Diploma degree in computer science from the Martin-Luther University at Halle-Wittenberg, Germany, in 1997. He is currently working toward the Ph.D. degree at the Computer Science Institute of the Free University of Berlin.

During 1994 and 1995, he was a student at the University of Houston, where he worked in the Image Processing Laboratory of the Department of Electrical and Computer Engineering. During 1997, he did research for Siemens AG. His research interests include growing neural architectures, hierarchical image analysis, and neural control.

Nicolaos B. Karayiannis (S'85–M'91) was born in Greece on January 1, 1960. He received the diploma degree in electrical engineering from the National Technical University of Athens in 1983, and the M.A.Sc. and Ph.D. degrees in electrical engineering from the University of Toronto, Canada, in 1987 and 1991, respectively.

He is currently an Associate Professor in the Department of Electrical and Computer Engineering, University of Houston, TX. From 1984 to 1991, he worked as a Research and Teaching Assistant at the University of Toronto. From 1983 to 1984, he was a Research Assistant at the Nuclear Research Center "Democritos," Athens, Greece, where he was engaged in research on multidimensional signal processing. He has published more than 70 papers, including 28 in technical journals, and is the coauthor of the book *Artificial Neural Networks: Learning Algorithms, Performance Evaluation, and Applications* (Boston, MA: Kluwer, 1993). His current research interests include supervised and unsupervised learning, applications of fuzzy logic in neural modeling, applications of artificial neural networks in image processing and communications, learning vector quantization and its applications in image and video compression.

Dr. Karayiannis is a member of the International Neural Network Society (INNS) and the Technical Chamber of Greece. He is the recipient of the W. T. Kittinger Outstanding Teacher Award. He is also a corecipient of a Theoretical Development Award for a paper presented at the Artificial Neural Networks in Engineering'94 Conference. He is an Associate Editor of the IEEE TRANSACTIONS ON NEURAL NETWORKS and the IEEE TRANSACTIONS ON FUZZY SYSTEMS. He also served as the General Chair of the 1997 International Conference on Neural Networks (ICNN'97), held in Houston, TX, June 9–12, 1997.