A Neural-Fuzzy Clustering Approach to Rule Extraction: the role of self-spawning competition

(Invited Paper)

Zhi-Qiang Liu
Centre for Media Technology, School of Creative Media
City University of Hong Kong, Hong Kong, SAR, CHINA
E-mail: smzliu@cityu.edu.hk

***Abstract*— Cluster analysis has been a popular and effective tool in dada analysis. Learning rules from data will enhance traditional clustering methods. In this paper, we present a neural-fuzzy approach to rule extraction from using the self-spawning competition, which is based on a generic definition of incremental perceptron and a new competitive learning algorithm we recently developed. It extracts a correct number of rule patches and their positions and shapes in the input space. Initially the rule base consists of only a single fuzzy rule; during the iterative learning process the rule base expands according to a supervised spawning-validity measure. To demonstrate the effectiveness and applicability of our algorithm, we present some experimental results.**

I. Introduction

Clustering has been major paradigm in data analysis. Traditional approaches often overfit data as more data samples are available due to the lack of prior knowledge in performing clustering. Fuzzy inference systems are able to model the continuous input/output relationships by means of fuzzy IF-THEN rules without employing precise quantitative analysis. This fuzzy modeling has been successfully used in many applications, e.g., to build function approximators, fuzzy controllers, fuzzy classifiers, and decision making.

The basic components in a fuzzy inference system are fuzzy rules. It is desirable that the rule base covers all the states of the system that are important to the intended application [18]. In general, fuzzy rules can be obtained by either human experts or a data-driven extraction scheme from measured input-output data pairs. The latter case is currently a growing research topic [3], since in most cases people or decision makers in a data-mining project or industry applications are usually not trained statisticians, mathematicians, or AI experts[4]. Moreover, in many commercial areas there are a huge number of data collections. Therefore, it is important to learn knowledge and derive fuzzy rules from the data itself [2].

Over the last few years, research on combining neural networks and fuzzy systems, neuro-fuzzy systems, has gained considerable importance and attention. The learning capabilities of neural networks give fuzzy systems the ability to tune the paramaters and shapes of fuzzy membership functions. Among the proposed methods, the hybrid neuro-fuzzy approach has recently become most popular, which can be viewed as a special n-layer feedforward neural network [16] with sampled fuzzy memberships as the input/output [8], [9], or with parameterized membership functions stored in the neurons [7], [1], or with fuzzy sets as the link weights [16]. With learning capability, we are able to determine the fuzzy sets or fuzzy rules. Learning in a neuro-fuzzy system normally involves two phases: structural learning and parametric learning. The structure learning tunes a number of rules; thus it is also often referred to as *rule learning*, whereas the parametric learning tunes the positions of fuzzy sets. Therefore, once the neural network architecture has been determined, the number of rules are assumed fixed during the learning process. Such a process is often not effective for many real-world problems (i.e., small and interpretable) rule bases. Researchers have proposed pruning techniques to reduce the number of rules and variables in the neuro-fuzzy system [16], [19]. Such approaches usually start from a *large* number of rules; during the learning process some of the rules are *pruned* (rule deduction) resulting in a set of rules that are most relevant to the problem at hand. However, it is difficult in most cases to choose a reasonably *large* number of rules, because we simply do not have enough prior knowledge about the data. This can lead to poor system performance in real-world applications. In the last few years incremental learning (rule induction) have been used [15]. Unfortunately, A serious problem with such learning algorithms is that they do not have a criterion to terminate the growth process of the neural network structure: the termination is determinde subjectively and based on a pre-defined maximum network size. Furthermore, the shape adaption of a membership function is a simple calculation between two closest rule patches in the input space, e.g., half of their distance, which is not suitable for rule patches with different sizes, as in this case the memberships for smaller rule patches may overlap significantly with parts of larger rule patches, leading to misclassification in data analysis.

In this paper, we introduce a new incremental, hybrid neuro-fuzzy system using the self-spawning competitive learning paradigm, *Self-Spawning Neuro-Fuzzy System* (SSNFS). It is a scatter-partitioning fuzzy inference system that allows the IF-parts of the fuzzy rules to be positioned at arbitrary locations in the input space. A major problem with scatter partitions is to find a suitable number of rules, suitable positions and suitable width of the rule patches in the input space. Our neuro-fuzzy model is able to incrementally and adaptively build up a network structure during the training process. It requires only a single rule prototype randomly initialized in the input space; during the training process the rule base expands adaptively according to a spawning validity measure to discover more rule patches. The shape width of the rule patch is indicated by an on-line property vector. The rule induction terminates when a stop criterion is satisfied. The output weights are trained in an on-line fashion, which is more appropriate (computationally less complex) than that in batched learning. To extract rules from data, we assume that a limited number of input-output pairs are provided on-line for single-pass processing.

II. SCATTER-PARTITIONING FUZZY SYSTEMS

A. Propositional Fuzzy Logic

The basic component of a fuzzy inference system is the fuzzy rule which is expressed using linguistic labels, for instance,

$$\begin{aligned} &\text{IF } (x_1 \text{ is low}) \text{ AND } (x_2 \text{ is medium}) \\ &\text{THEN } (y_1 \text{ is } 0.2) \text{ AND } (y_2 \text{ is } 0.4), \end{aligned} \tag{1}$$

where x_1, $x_2 \in \mathcal{R}$ is the input variables (antecedent) and y_1, $y_2 \in \mathcal{R}$ is the output (consequent) of this rule. The linguistic labels (e.g., low) are usually modeled as parameterized membership functions (MFs) within a particular area of the input space. AND is the T-norm fuzzy operator and in some cases OR (T-conorm) is used in fuzzy rules. In the example shown above, the fuzzy sets involved only in the premise part (IF-part). Fuzzy systems of this kind are referred to as *zeroth-order fuzzy systems*. Generally, in an nth-order fuzzy system, the THEN-part of each rule consists of a polynomial of degree n in the input variables.

In this paper we will concentrate on zeroth-order fuzzy systems. Different shapes of the MFs have been proposed such as the Gaussian, triangular, or trapezoidal. In the following discussions we assume that the MFs have the form of the Gaussian function and that the fuzzy system consists of m fuzzy rules each with n input variables to classify data into k fuzzy or crisp classes. Thus, the fuzzy system can be described as $\{R_i\}_{i\in\{1,\ldots,m\}}$. For the ith rule R_i,

$$\begin{aligned} \mathcal{R}_i:\ &\text{IF } (x_1 \text{ is } g_{i1}(x_1)) \text{ AND}(x_2 \text{ is } g_{i2}(x_2)) \\ &\text{AND} \cdots \text{ AND } (x_n \text{ is } g_{in}(x_n)) \\ &\text{THEN } (y_1 \text{ is } t_{i1}) \text{ AND } (y_2 \text{ is } t_{i2}) \\ &\text{AND} \cdots \text{ AND } (y_k \text{ is } t_{ik}), \end{aligned} \tag{2}$$

where $g_{ij}(x_j)$ is membership function denoting the lingusitic label associated with the jth input variable in the ith fuzzy rule,

$$g_{ij}(x_j) = exp\left[-\frac{(x_j - p_{ij})^2}{2\sigma_{ij}^2}\right], \tag{3}$$

where the variable p_{ij} and σ_{ij} are the center and variance of the Gaussian membership function g_{ij}, respectively. $\vec{T_i} = [t_{i1},\ldots,t_{ik}]^T$ is the output vector for the ith fuzzy rule, $t_{ij} \in \{0,1\}$ for partitioning crisp clusters or function approximation and $t_{ij} \in [0,1]$ for partitioning fuzzy clusters.

Usually only a moderate number of MFs are defined for each input variable. It is always desirable that the membership values for each input variable add to unity everywhere. This may be achieved by dividing the membership values $g_{ij}(x_j)$ by the sum of all memberships with respect to x_j, leading to *normalized* MFs,

$$\hat{g}_{ij}(x_j) = g_{ij}(x_j)/\sum_{l=1}^{m} g_{lj}(x_j). \tag{4}$$

We focus our attention to rules that combine their sub-expressions by fuzzy AND. In the case of Guassian MFs, the fuzzy AND can be realized by the arithmetic product. Let G_i denote the membership value for the IF-part of R_i in (2),

$$G_i = \prod_{j=1}^{n} g_{ij}(x_j). \tag{5}$$

In this case the IF-part of each fuzzy rule can be described by an n-dimensional Guassian membership function,

$$G_i(\vec{X}) = exp(-\frac{1}{2}(\vec{X}-\vec{P_i})^T\Lambda^{-1}(\vec{X}-\vec{P_i})), \tag{6}$$

where $\vec{X}$ is the multivariate input vector $[x_1, x_2, \ldots, x_n]^T$, $\vec{P_i} = [p_{i1}, p_{i2}, \ldots, p_{in}]^T$ is the n-dimensional Gaussian center that has the corresponding centers of the one-dimensional factor Gaussians as components. $\Lambda^{-1} = diag(1/\sigma_{i1}^2, 1/\sigma_{i2}^2, \ldots, 1/\sigma_{in}^2)$ is the inverse of the covariance matrix Λ corresponding to the product of n one-dimensional Guassian memberships. Similarly, we can normalize the multivariate Gaussian membership functions as follows:

$$\hat{G}_i(\vec{X}) = G_i(\vec{X})/\sum_{l=1}^{m} G_l(\vec{X}). \tag{7}$$

For the input pattern $\vec{X}$, the output of the ith component of a fuzzy system with normaliztion is given by

$$O^{(i)}(\vec{X}) = \sum_{j=1}^{m} t_{ji}\hat{G}_j(\vec{X}) = \sum_{j=1}^{m} t_{ji}G_j(\vec{X})/\sum_{j=1}^{m} G_j(\vec{X}). \tag{8}$$

B. Scatter-Partitioning Fuzzy Systems

Based on (6), we may simplify the fuzzy rule in (2) as

$$\mathcal{R}_i:\ \text{if } \vec{X} \text{ is } G_i(\vec{X}) \text{ then } \vec{Y} \text{ is } \vec{T_i}, \tag{9}$$

where $\vec{Y} = \{y_1, y_2, \ldots, y_k\}$ is the multivariate output variable and T_i is the consequent output vector for ith rule.

Since each Gaussian membership function G_i, $i \in \{1,\ldots,m\}$ covers a particular area in the input space, each rule is considerably activated only in this area. One can think of the input space as being covered by some small patches, each of them corresponding to the IF-part of one fuzzy rule. We denote these small patches as *rule patches* in this paper. Building a meaningful fuzzy inference system to a large extent is to establish a series of fuzzy rules that cover all the rule patches in the input space. At the same time, we must keep the number of rules as low as possible in order to maintain the generalizing ability of the model and to ensure a compact and transparent model [18]. Therefore, our task is reduced to seeking an optimal number of rule patches, locating their positions, and computing their shape widths in the input space. This is known as *scatter-partitioning* distinguished from *grid-partitioning* approaches in which the rule patches are assumed on a regular predefined grid distribution. In a scatter-partitioning fuzzy system the fuzzy memberships are not confined to the corners of a rectangular grid. Rather, they can activate anywhere with any width in the input space as long as the $\vec{P}$ and Λ in (6) can be detected. If the convariance matrices Λ_i of the Gaussians are diagonal, then they can still be thought of being generated as a product of n one-dimensional Gaussian MFs. Fig.1 shows an example of grid-partitioning and scatter-partitioning fuzzy systems. Once the IF-parts are solved, the output vector $\vec{T}$ for each rule may be achieved by minimizing the summed squared error,

$$E = \frac{1}{2}\sum_{i=1}^{M}\sum_{j=1}^{m}\left[\vec{Y}_i - \hat{G}_j(\vec{X}_i)\vec{T}_j\right]^2, \tag{10}$$

where $(\vec{X}_i, \vec{Y}_i)$ is the training pattern, and M is the total number of patterns presented for training. This approach is usually computationally complex. In fact, the output vector can be learned on-line concurrently in the output space to reduce the complexity, such as training by the delta-rule [15].

Now, the problem is how to correctly classify the rule clusters in the input space. The conventional learning algorithms using neural networks or fuzzy clustering are very sensitive to the number of prototypes initialized. We, however, do not usually have sufficient prior knowledge about the data set. Therefore it is desirable to develop an algorithm that is able to adaptively detect the number of rule patches and their locations.

In this paper we consider an incremental learning paradigm which first constructs a fuzzy system by seeking one fuzzy rule, and conducting rule induction in an iterative learning process according to a self-spawning validity measure. The rule growth is terminated when all the rule patches have been detected based on a stop criterion.

III. The SSNFS Neuro-Fuzzy Modeling

Many neuro-fuzzy systems use parameterized membership functions stored in the “neurons” of a multilayered feed-forward architecture, e.g., GARIC [1] and ANFIS [7]. The links in these kind of networks indicate only the data flow directions between nodes and no weights are associated with the links. The membership functions are usually parameterized Gaussians, trapezoidal, or triangular functions, or they are constructed by superposing sigmoids. NEFCLASS [16] builds a different structure which uses sampled memberships (fuzzy sets) as link weights.

It must be clear that the above mentioned neuro-fuzzy models (and in fact almost all neuro-fuzzy models in the literature) have little to do with fuzzy logic in the narrow sense [11] but to do with the parameterized membership functions in the fuzzy IF-THEN rules that are associated with linguistic labels.

A. SSNFS Architecture

As discussed in Section II, each fuzzy rule is associated with one parameterized Gaussian membership with multivariate inputs and one output vector, as seen in (9). The Gaussian membership functions can be identified by detecting rule patches in the input space, whereas the output vectors can be updated on-line in the output space. In this perspective, we propose SSNFS based on a generic incremental perceptron and a new learning algorithm, *self-spawning competitive learning*, to incrementally search the rule patches.

Similar to that in the ANFIS, the membership functions are stored in the hidden nodes. However, the difference is that the new model uses a multivariate membership function for each rule, thus the node with this membership function is fully connected to all the input variables. In addition, in SSNFS, we model the parameters for each membership function as the link weights between the input nodes and membership nodes and the consequent vector of each rule as the weights between the input nodes and rule nodes. The benefit is that the weight vectors can be adaptively updated given input training patterns.

To build a neuro-fuzzy system for extracting fuzzy linguistic rules, we present a generic 5-layer incremental perceptron that will provide a basis for SSNFS architecture and self-spawning learning algorithm.

Definition 1: A generic 5-layer incremental perceptron is a 5-layer feedforward neural network (U,W,P,T,NET,A,O,ex,Inc) with the following specifications:

1) $U = \bigcup_{i\in I} U_i$ is a non-empty set of neurons and $I = \{1,\ldots,5\}$ is the index set of U. For all $i, j \in I$, $U_i \neq \emptyset$ and $U_i \cap U_j = \emptyset$ for $i \neq j$. $U_1 = U_1^{(\vec{\mathcal{X}})} \bigcup U_1^{(\vec{X})} \bigcup U_1^{(\vec{Y})}$ is the input layer, U_2, U_3 and U_4 are the hidden layers, and U_5 is the output layer.
2) Let $M = \{2m_1 + m_5, m_2, m_3, m_4, m_5\}$ define the number of neurons for layer 1 to layer 5, respectively; m_1 is the number of neurons for either $U_1^{(\vec{\mathcal{X}})}$ or $U_1^{(\vec{X})}$; $m_2 = m_3 = m_4$ is the number of neurons for each hidden layer; m_5 is the number of neurons for $U_1^{(\vec{Y})}$ or U_5.
3) W defines the link connectedness in the perceptron as follows:
   (a) for $u_i \in U_1^{(\vec{\mathcal{X}})} \bigcup U_1^{(\vec{X})}$ and $v_j \in U_2$, $W(u_i, v_j) = 1$;
   (b) for $u_i \in U_1^{(\vec{Y})}$ and $v_j \in U_4$, $W(u_i, v_j) = 1$;
   (c) for $u_i \in U_2$ and $v_j \in U_3$, $W(u_i, v_j) = 1$;
   (d) for $u_i \in U_3$ and $v_j \in U_4$ and $i = j$, $W(u_i, v_j) = 1$;
   (e) for $u_i \in U_4$ and $v_j \in U_5$, $W(u_i, v_j) = 1$; otherwise,
   (f) $W(u_i, v_j) = 0$.
4) $P = \{\vec{P}_1, \ldots, \vec{P}_{m_2}\}$ defines the weight vectors (prototypes) on the connections $W(U_1^{(\vec{X})}, U_2)$. The ith prototype can be given by $\vec{P}_i = [p_{i1}, \ldots, p_{im_1}]^T$ where p_{ij} is the weight on the connection $W(u_j, v_i)$ with

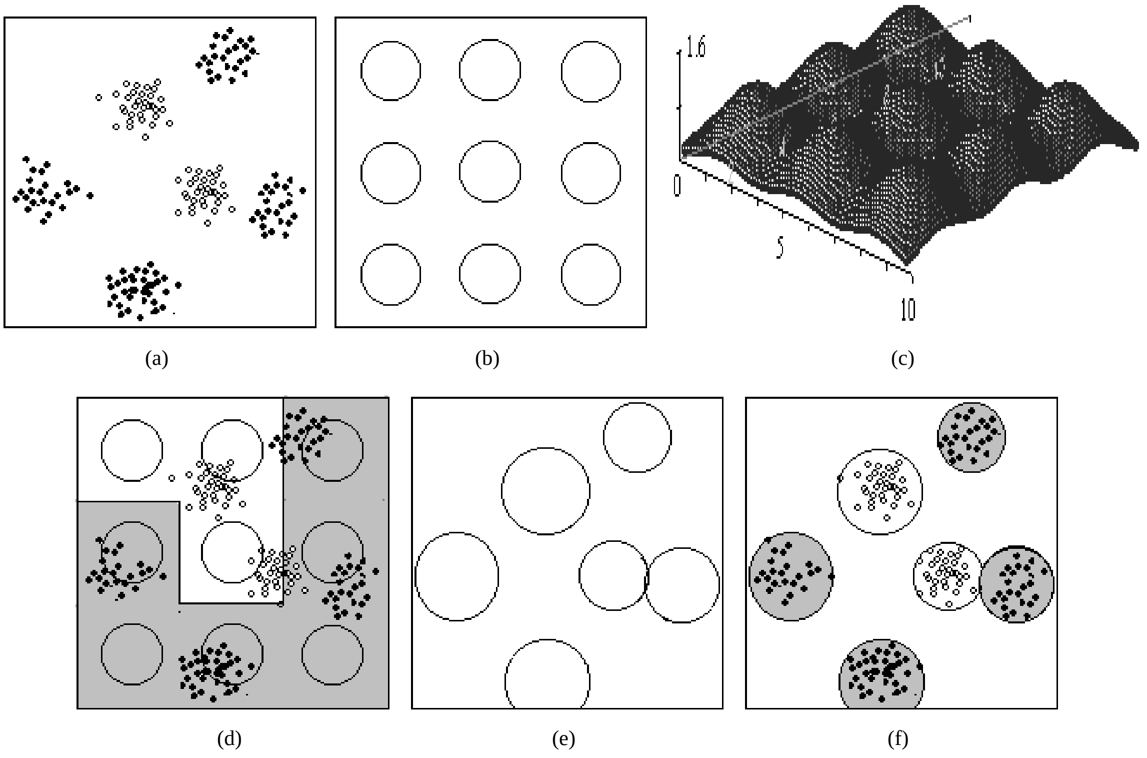

Fig. 1. Classfication example with a grid-partitioning fuzzy system and a scatter-partitioning fuzzy system. (a) Two dimensional dataset in the unit square consisting of two classes (white and black). (b) The grid-partitioning system initializes 3×3 grid to arrange 9 fuzzy rules. (c) The non-normalized Gaussian view in the input space for this grid-partitioning system. (d) The classification result by the grid-partitioning system. (e) The scatter-partitioning system captured 6 rule patches in the input space. (f) The classification result by the scatter-partitioning system.

$u \in U_1^{(\vec{X})}$ and $v \in U_2$. $T = \{\vec{T}_1, \ldots, \vec{T}_{m_2}\}$ defines the weight vectors on the connections $W(U_1^{(\vec{Y})}, U_4)$. $\vec{T}_i = [t_{i1}, \ldots, t_{im_5}]^T$ where t_{ij} is the weight on the connection $W(u_j, v_i)$ with $u \in U_1^{(\vec{Y})}$ and $v \in U_4$.

5) NET defines the propagation function for each unit $u \in U$ to calculate the net input net_u.
   (a) for $u \in U_1$:
   $NET_{u_i} : \mathcal{R} \mapsto \mathcal{R},\ net_{u_i} = ex_{u_i};$
   (b) for $u \in U_2$:
   $NET_{u_i} : \mathcal{R}^{m_1} \mapsto \mathcal{R}^{m_1},$
   $net_{u_i} = O(U_1^{(\vec{X})}) - [o_{v_1}p_{i1}, o_{v_2}p_{i2}, \ldots, o_{v_{m_1}}p_{im_1}]^T,\ \ v \in U_1^{(\vec{X})};$
   (c) for $u \in U_3$:
   $NET_{u_i} : \mathcal{R}^{m_2} \mapsto \mathcal{R},$
   $net_{u_i} = \frac{o_{v_i}}{\sum_{j=1}^{m_2} o_{v_j}},\ \ v \in U_2;$
   (d) for $u \in U_4$:
   $NET_{u_i} : \mathcal{R} \times \mathcal{R}^{m_5} \mapsto \mathcal{R}^{m_5},$
   $net_{u_i} = [o_{v_i}t_{i1}, o_{v_i}t_{i2}, \ldots, o_{v_i}t_{im_5}]^T,\ \ v \in U_3;$
   (e) for $u \in U_5$:
   $NET_{u_i} : \mathcal{R}^{m_4} \mapsto \mathcal{R},$
   $net_{u_i} = \sum_{j=1}^{m_4} o_{v_j},\ \ v \in U_4.$

6) A defines the activation function for each $u \in U$ to calculate the activation a_u.
   (a) for $u \in U_1 \bigcup U_3 \bigcup U_4 \bigcup U_5$:
   $A_{u_i} : a_{u_i} = A_{u_i}(net_{u_i}) = net_{u_i};$
   (b) for $u \in U_2$:
   $A_u : \mathcal{R}^{m_1} \mapsto \mathcal{R},\ \ a_u = A_u(net_u).$
   where A_u is the parameterized activation function.

7) O defines for each $u \in U$ an output function O_u to calculate the output o_u.
   (a) for $u \in U_1 \bigcup U_2 \bigcup U_3 \bigcup U_4$:
   $o_{u_i} = O_{u_i}(a_{u_i}) = a_{u_i};$
   (b) for $u \in U_5$:
   $O_{u_i} : \mathcal{R} \mapsto \{0, 1\}$ (for fuzzy partitioning or function approximation), $o_u = O_u(a_u) = a_u;$
   $O_{u_i} : \mathcal{R} \mapsto [0, 1]$ (for crisp partitioning), $o_u = O_u(a_u) = DF(a_u)$, where DF is a suitable defuzzification function.

8) ex defines for each input unit $u \in U_1$ its external input $ex(u) = ex_u$. For all other units ex is not defined.

9) Inc defines the policy for the perceptron increment. A spawning request will be carried out by taking the following actions:
   (a) One neuron is spawned for each hidden layer, while the input and output layers remain unchanged. Let u, v and w indicate the new added neuron for layer 2, 3, and 4, respectively.
   $U_2 := U_2 \bigcup \{u\},\quad m_2 := m_2 + 1;$
   $U_3 := U_3 \bigcup \{v\},\quad m_3 := m_3 + 1;$

$U_4 := U_4 \bigcup \{w\}, \quad m_4 := m_4 + 1;$
$m_2 = m_3 = m_4$ holds anytime.

(b) for all U_1, U_2, U_3, U_4, and U_5, the structure are reconstructed following the definition of (U,W,P,T,NET,A,O,ex).

Given the definition of the generic perceptron, we can describe the SSNFS as follows.

Definition 2: A SSNFS system is a generic 5-layer incremental perceptron employing supervised self-spawning competitive learning algorithm with the following specifications:

1) In SSNFS, $U_1^{(\vec{\mathcal{X}})}$ is the input layer for data classification; $U_1^{(\vec{X})}$ and $U_1^{(\vec{Y})}$ are the input layers for training; U_2 is the membership layer; U_3 is the normalized membership layer; U_4 is the rule layer and U_5 is the output layer.
2) SSNFS starts from a single neuron for each hidden layer, $m_2 = m_3 = m_4 = 1$ holds initially.
3) SSNFS assigns each prototype $\vec{P_i}$ with three property vectors $(\vec{A_i}, \vec{C_i}, \vec{R_i})$, called Asymptotic Property Vector (APV), Center Property Vector (CPV), and Distant Property Vector (DPV), respectively. In a simulated neural network, it is achieved by assigning each p_{ij} three other weights, while for a real neural network we can simply set $W(u_i, v_j) = 4$ where $u \in U_1^{(\vec{X})}$ and $v \in U_2$.
4) SSNFS applies Gaussian membership functions or triangular membership functions as the activation functions for layer 2; $\vec{P}$ and its property vectors are the parameter vectors for membership functions and $\vec{T}$ is the consequent output vector for a rule.
5) SSNFS switches between the training process and classification task as following:
   (a) for training process, $ex_u = N/A$, for $u \in U_1^{(\vec{\mathcal{X}})}$; $U_1^{(\vec{X})} \bigcup U_1^{(\vec{Y})}$ is the active input layer;
   (b) for classification task, $ex_v = 1$, for $v \in U_1^{(\vec{X})} \bigcup U_1^{(\vec{Y})}$; $U_1^{(\vec{\mathcal{X}})}$ is the active input layer.

In the SSNFS model, we model fuzzy rule extraction as a task to spawn the structure and to establish the premise and consequent parameters in the training process.

B. Supervised Self-Spawning Competitive Learning

As defined above, SSNFS employs supervised self-spawning competitive learning (SSSCL) algorithm to tune the structure and parameters. However, we face a usual problem of the One Prototype Takes Multi-Clusters (OPTMC). As shown in Fig.2(a), suppose there are three rule patches in the input space and one prototype is initialized to detect them, by the conventional competitive learning paradigm, this single prototype will move to the center of the training patterns. As a result, the extracted fuzzy rule covers a wrong rule patch leading to misclassification. In fuzzy systems, or for that matter any rule-based systems, it is desirable that the extracted fuzzy rules represent the true rule patches.

The SSSCL tackles the $OPTMC$ problem by designing a new learning scheme in which one prototype takes one cluster ($OPTOC$) and ignores the others when the number of prototypes is less than that of the natural clusters, or in this case, the true rule patches. Fig.2(b) shows the same example as that in (a) but applying the $OPTOC$ learning paradigm. In this case, one rule patch (cluster) is correctly detected, and the rest rule patches can be detected by spawning new prototypes in subsequent learning rounds. In this way we will be able to extract a correct set of rules from the data set.

In the neural-fuzzy network, the spawning of new prototypes is accomplished by expanding the SSNFS structure according to a spawning validity measure and the increment policy. The newly spawned prototype is to extract one more fuzzy rule in the next learning cycle. This growth process terminates when a stop criterion is satisfied. The process is *supervised* in that the spawning validity measure is based on the desired output patterns for the given training patterns.

Let $\vec{P_i}(\vec{A_i}, \vec{C_i}, \vec{R_i})$ denote the ith prototype in terms of its three property vectors; $(\vec{X}, \vec{Y})$ denote the pair of training patterns for input layer, where $\vec{X}$ is a m_1-dimensional pattern in the input space, and $\vec{Y}$ is a desired m_5-dimensional pattern in the output space; E_i denote the regression error and $\vec{T_i}$ the desired output vector for the fuzzy rule extracted by $\vec{P_i}$. Each time when a pair of patterns, $(\vec{X}, \vec{Y})$, is randomly picked from the training set, the competition occurs among all the current prototype vectors. The winning prototype is judged by the nearest neighbor criterion.

The SSSCL OPTOC learning algorithm

STEP 1.

1) SSNFS starts from one fuzzy rule, therefore, $m_2 = m_3 = m_4 = 1$ holds initially. The only single prototype vector, $\vec{P_1}$, is initialized randomly in the input space. Its $APV(\vec{A_1})$ and $CPV(\vec{C_1})$ are initialized at a random location far from $\vec{P_1}$, whereas its $DPV(\vec{R_1})$ is set at the same place as $\vec{P_1}$. $\vec{T_i}$ is randomly initialized in the output space and $\vec{E_i}$ is initialized to 0.
2) For the input $(\vec{X}, \vec{Y})$, if $\vec{P_i}$ is the winning prototype for $\vec{X}$, its APV ($\vec{A_i}$)is updated in the input space by
$$\vec{A_i^*} = \vec{A_i} + \frac{1}{n_{\vec{A_i}}} \cdot \delta_i \cdot (\vec{X} - \vec{A_i}) \cdot \Theta(\vec{P_i}, \vec{A_i}, \vec{X}), \quad (11)$$
where
$$\Theta(\vec{\mu}, \vec{\nu}, \vec{\omega}) = \begin{cases} 1 & \text{if } |\vec{\mu}\vec{\nu}| \geq |\vec{\mu}\vec{\omega}|, \\ 0 & \text{otherwise.} \end{cases} \quad (12)$$
and
$$\delta_i = \left(\frac{|\vec{P_i}\vec{A_i}|}{|\vec{P_i}\vec{X}| + |\vec{P_i}\vec{A_i}|} \right)^2 . \quad (13)$$
$n_{\vec{A_i}}$ is the winning counter and $|\vec{u}\vec{v}|$ is the Euclidean distance between a vector $\vec{u}$ and a vector $\vec{v}$. δ is the adaptively updated learning rate which satisfies $0 < \delta_i \leq 1$.
3) The DPV ($\vec{R_i}$) always follows the farthest pattern for which $\vec{P_i}$ has been the winner in the input space so far. For the input $(\vec{X}, \vec{Y})$, if $\vec{P_i}$ is the winning prototype for $\vec{X}$, $\vec{R_i}$ is updated by
$$\vec{R_i^*} = \vec{R_i} \cdot (1 - \Theta(\vec{P_i}, \vec{X}, \vec{R_i})) + \vec{X} \cdot \Theta(\vec{P_i}, \vec{X}, \vec{R_i}). \quad (14)$$

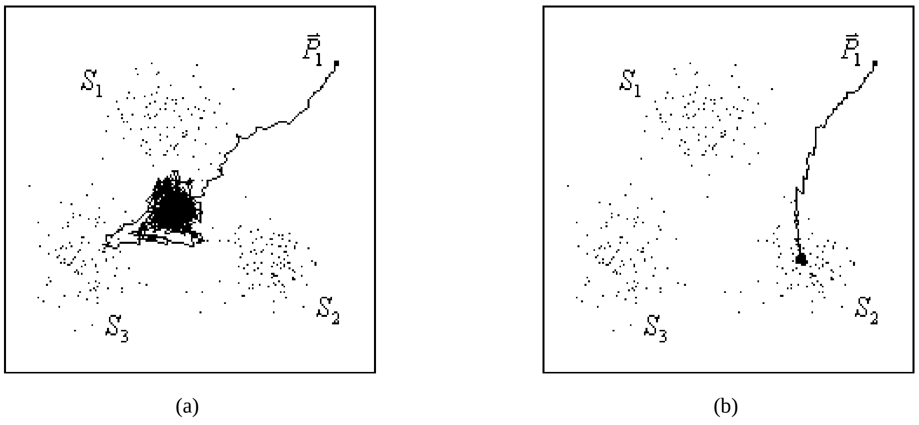

Fig. 2. Two learning behaviors: OPTMC vs OPTOC. (a) OPTMC: One prototype $\vec{P}_1$ is trying to take all three patches $\{S_1, S_2, S_3\}$, resulting in oscillation phenomenon. (b) OPTOC: This prototype detects only one rule patch S_2 and ignores the other two.

4) For the input $(\vec{X}, \vec{Y})$, if $\vec{P}_i$ is the winning prototype for $\vec{X}$, the following update scheme guarantees $\vec{P}_i$ to cluster one rule patch and ignore the others in the input space.

$$\vec{P}_i^* = \vec{P}_i + \alpha_i \cdot \beta_i \cdot (\vec{X} - \vec{P}_i), \qquad (15)$$

where α_i is computed with

$$\alpha_i = \left(\frac{|\vec{P}_i \vec{A}_i|}{|\vec{P}_i \vec{X}| + |\vec{P}_i \vec{A}_i|} \right)^2 \quad (0 < \alpha_i \leq 1), \qquad (16)$$

and β_i is given by

$$\beta_i = \left(\frac{|\vec{P}_i \vec{R}_i|}{|\vec{P}_i \vec{X}| + |\vec{P}_i \vec{R}_i|} \right)^2 \quad (0 < \beta_i \leq 1). \qquad (17)$$

The OPTOC scheme enables each prototype to find only one natural rule patch in the input space when the patches is more than the prototypes. This in itself is a major improvement over other competitive learning algorithms in the literature. However, at this stage we are still not sure whether there are other rule patches that have not been detected yet. For this we introduce a spawning validity measure to determine if all the rule patches have been properly discovered. If not yet, SSNFS expands its structure by spawning and appending one neuron for each hidden layer, then reconstruct the architecture. The prototype vector of the newly spawned neuron in U_2 is hence to join the competition in the next iterative learning process. Based on the $OPTOC$ learning scheme, it is therefore able to extract one more fuzzy rule.

Definition 3: **(The SSSCL Spawning Validity Measure and Stop Criteria)**

1) For the input $(\vec{X}, \vec{Y})$, if $\vec{P}_i$ is the winning prototype for $\vec{X}$, $\vec{C}_i$ and $\vec{T}_i$ are updated on-line by the k-Means learning scheme [12],

$$\vec{C}_i^* = \vec{C}_i + \frac{1}{n_{\vec{C}_i}}(\vec{X} - \vec{C}_i); \quad \vec{T}_i^* = \vec{T}_i + \frac{1}{n_{\vec{T}_i}}(\vec{X} - \vec{T}_i). \qquad (18)$$

Just as the name k-Means indicates, the CPV $(\vec{C}_i)$ indicates the arithmetic means (centroid) of all the X's and $\vec{T}_i$ indicates the arithmetic centroid of all the Y's, where $(\vec{X}, \vec{Y})$'s are the presented patterns for which $\vec{P}_i$ has been the winner.

2) E_i denotes the regression error for the fuzzy rule extracted by $\vec{P}_i$. Therefore, it is given by

$$E_i^* = E_i + (O_i^{(\vec{X})} - \vec{Y})^2, \qquad (19)$$

where O_i represents the output vector of the fuzzy rule extracted by $\vec{P}_i$ as the if-parts and $\vec{T}_i$ as the then-parts.

3) For all $i \in \{1, \ldots, m_2\}$, if $|\vec{P}_i \vec{A}_i| < \epsilon$ and there is at least one prototype $\vec{P}_j$ satisfies $|\vec{P}_j \vec{C}_j| > \epsilon$, then SSNFS is suitable for spawning. ϵ is a small positive constant which is theoretically 0. The self-spawning process is carried by:

(a) SSNFS expands its structure in terms of the policy Inc defined in the Definition (1).

(b) Let $\vec{P}_{m_2}$ denote the prototype vector for the newly spawned neuron in U_2. It is initialized with

$$\vec{P}_{m_2} = \vec{R}_s$$

where the index s satisfies

$$s = \arg min_j E_j|_{|\vec{P}_j \vec{C}_j| > \epsilon}$$

set $\vec{R}_{m_2} = \vec{P}_{m_2}$, $\vec{A}_{m_2} = \vec{C}_{m_2}$ at a random location in the input space, $E_{m_2} = 0$, and T_{m_2} at a random location in the output space.

4) For all $i \in \{1, \ldots, m_2\}$, if $|\vec{P}_i \vec{A}_i| < \epsilon$ and $|\vec{P}_i \vec{C}_i| < \epsilon$, SSNFS finish its structural and parametric learning and can be interpreted as a set of established fuzzy rules.

SSNFS adaptively updates its structure and weight parameters to extract fuzzy rules. In the next section, we present our experimental results.

IV. Experimental Results

To demonstrate our system and its applicability, in this section we present the experimental results for two classical problems. First, we consider the reconstruction of a known rule base from data to verify the ground truth. The purpose is to identify the partition in the data from the rule base. Second, we describe an application in pattern recognition using a well-known benchmark data, namely, the Iris data set.

A. Modeling a Known Rule-Base

We consider an existing fuzzy system which partitions a two-dimensional data into three classes: A, B, and C. Let $Tr(x, a, b, c)$ denote the triangular membership function specified by three parameters (a, b, c),

$$Tr(x, a, b, c) = \begin{cases} \frac{x-a}{b-a}, & \text{if } a \leq x \leq b \\ \frac{c-x}{c-b}, & \text{if } b < x \leq c \\ 0, & \text{if } x < a \text{ or } x > c. \end{cases} \tag{20}$$

The rule base consists of seven fuzzy rules as shown in Table I. The input variables take values in the domain $\{0, 10\}$. The output vector y indicates the class an input pattern belongs. The three classes are labeled as $A = [100]$, $B = [010]$, and $C = [001]$.

Fig.3(a) shows the training data set obtained by the data-generating rule base. It contains 200 data points partitioned into three classes. We know that the data-generating system consists of seven rules, but it is not straightforward to tell how many clusters are actually present in the input space [18]. Given this data set, we apply the SSNFS model to extract the fuzzy rules with two-dimensional Gaussian membership functions. Our objective is to verify, from the given data set, whether the SSNFS is able to extract a set of rules that are similar to those in the original, known rule base. It is a network of two input neurons, one neuron for each hidden layer and three output neurons. The constant parameter ϵ in the spawning validity measure was initialized to 2% of the domain scale. Initially, as expected, the only single prototype extracted one fuzzy rule by detecting one rule patch. During the self-spawning learning process, the spawning action occurred seven times. Consequently the rule base expanded to eights rules finally. In Fig.3(b) the learning trajectories and spawning actions were drawn in detail in the input space to show the effectiveness of the SSNFS. The number in each circle indicates the spawning order and the initial position of the newly spawned prototype; the thin curves are the learning trajectories of prototype vectors. Fig.3(c) shows the partition of the data after the rules have been extracted by SSNFS. Table II lists the extracted rules with their parameters using SSNFS, where $\vec{P}_i$ $\vec{R}_i$, and $\vec{T}_i$ are defined in Definition 3.

Fig.4(a) shows the classification map with respect to class B data (filled circles) obtained by the original, known rule base. Fig.4(b) shows the corresponding classification map obtained by using the rules extracted by the SSNFS from the input data set (the 200 data points). Since the training data set does not reflect the global distribution of the data-generating rules, the extracted rules mostly perform well on these training data points and thereabouts. One can easily extract the global optimal fuzzy rules by uniformly sampling the input space with small step. Thus this is not a problem of the proposed technique but that of the computational complexity.

From the known rule set in Table I, the extract rules in Table II, and classification maps in Fig.4, we can see that the SSNFS is able to reconstruct the existing rule base very accurately.

B. Rule Extraction from Iris Data

We applied the SSNFS to the Iris data set [1] to extract a set of fuzzy rules. The objectives in this experiment are threefold: (1) to extract a set of fuzzy rules; (2) to classify the data set using the extracted rules; and (3) to compare the class centroids with the actual centroids. The SSNFS network consists of four input neurons, one hidden neuron for each hidden layer initially, and three output nodes. Throughout the test, we set $\epsilon = 0.02$. During the learning process, the spawning action occurred 4 times, therefore 5 rules were extracted by SSNFS. Table III lists the final rules with their parameters. To test the robustness of the proposed model, we carried out Monte Carlo tests and run the SSNFS 10 times. Each time, SSNFS was able to extract a very consistent set of rules; in fact, the rules were almost identical among the tests. Fig. 5 shows a classification result for attribute 2 against attribute 4 by using the extracted fuzzy rules to classify data points as being Iris Setosa, Versicolour, or Virginica.

The spawning constant ϵ is a constant that is used to tune the quality of rule extraction. The smaller the ϵ, the more rule patches in the input space, therefore the better accuracy of the rule extraction. Typically it is sufficient that $\epsilon < 0.05$. As shown in Table.IV, by setting $\epsilon = 0.02$ or $\epsilon = 0.05$, we can obtain the centroids of classes that are very close to the actual centroids of the three classes.

V. Conclusions

Cluster analysis is a powerful paradigm in dada analysis, especially when we are dealing with *dynamically* generated dada sets increasing found in the Internet. Many techniques are available in the literature. However, due to the insufficient prior knowledge, we are at the mercy of data samples in clustering which may lead to unreliable results. As an initial attempt, we have proposed a novel fuzzy-neural approach to learning rules from data that will have many data clustering applications. We first discussed the scatter-partitioning fuzzy system, based on which we then presented the SSNFS model, a neuro-fuzzy system for rule extraction. It is derived from a generic 5-layer incremental perceptron model and employs the supervised self-spawning competitive learning algorithm (SSSCL). When initialized with a single rule prototype, the SSNFS is able to adapt its structure to reach a suitable number of rules by the self-spawning learning algorithm. The considered synthetic and real-world examples demonstrated the effectiveness and applicability of the new rule generating system.

In most real-world applications, rarely do we have adequate prior knowledge to specify the shapes, locations, and the number of rules. Therefore, extracting rules using the self-spawning competitive learning algorithm is effective and

[1]The Iris data was obtained from *http://www.ics.uci.edu/ mlearn/ MLRepository.html* via a free, public ftp site.

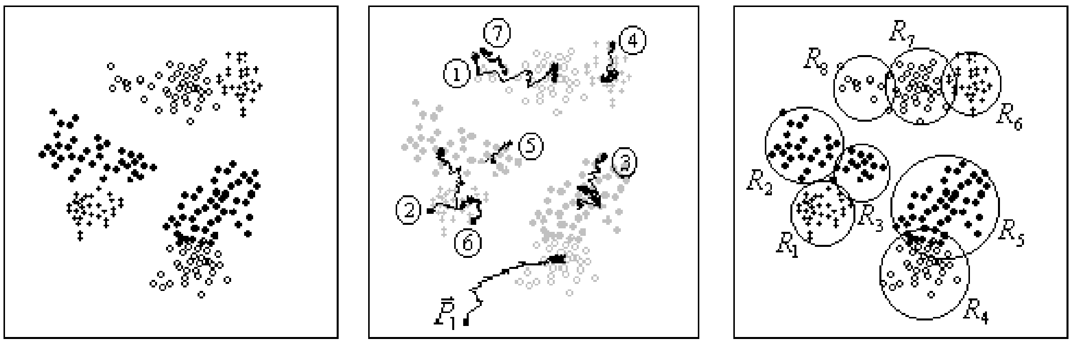

Fig. 3. (a) The data set contains 200 data points obtained by the data-generating rule base; they were partitioned into 3 classes: A (plus signs), B (filled circles), and C (hollow circles). (b) The OPTOC learning trajectories and spawning actions of SSNFS; the spawning occurred 7 times and finally 8 fuzzy rules were extracted. (c) The classification result obtained by our neural-fuzzy system.

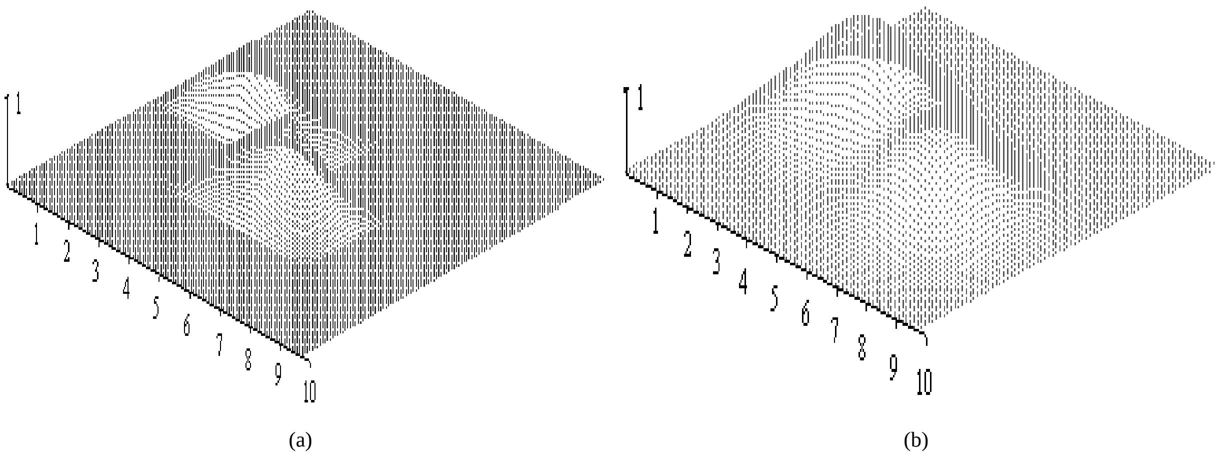

Fig. 4. (a) The class B output surface of the data generating rule base. (b) The corresponding output surface of the simulated fuzzy rules based on a limited data set.

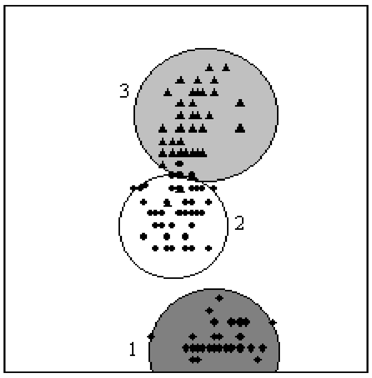

Fig. 5. An example: attribute 2 vs attribute 4 for Iris types 1, 2, and 3 classified by the SSNFS rules.

TABLE I

RULE BASE CONSISTS OF 7 FUZZY RULES FOR GENERATING THE DATA SET.

R_1	If x_1 is $Tr(x_1, 0.0, 2.7, 5.2)$ and x_2 is $Tr(x_2, 2.5, 4.2, 5.4)$ then y is $[1, 0, 0]$
R_2	If x_1 is $Tr(x_1, 0.0, 2.5, 5.0)$ and x_2 is $Tr(x_2, 4.8, 5.6, 7.5)$ then y is $[0, 1, 0]$
R_3	If x_1 is $Tr(x_1, 2.5, 5.2, 7.3)$ and x_2 is $Tr(x_2, 0.0, 1.5, 3.4)$ then y is $[0, 0, 1]$
R_4	If x_1 is $Tr(x_1, 2.5, 5.2, 7.3)$ and x_2 is $Tr(x_2, 2.5, 4.2, 5.4)$ then y is $[0, 1, 0]$
R_5	If x_1 is $Tr(x_1, 5.8, 7.5, 9.4)$ and x_2 is $Tr(x_2, 2.5, 4.2, 5.4)$ then y is $[0, 1, 0]$
R_6	If x_1 is $Tr(x_1, 5.8, 7.5, 9.4)$ and x_2 is $Tr(x_2, 5.6, 7.5, 9.2)$ then y is $[1, 0, 0]$
R_7	If x_1 is $Tr(x_1, 2.5, 5.2, 7.3)$ and x_2 is $Tr(x_2, 5.6, 7.5, 9.2)$ then y is $[0, 0, 1]$

TABLE II

THE RULE PARAMETERS EXTRACTED BY SSNFS BASED ON THE DATA SET IN FIG.3.

	Rule 1	*Rule 2*	*Rule 3*	*Rule 4*	*Rule 5*	*Rule 6*	*Rule 7*	*Rule 8*
$\vec{P}_i$	(2.71, 3.75)	(2.08, 5.83)	(3.85, 5.00)	(5.83, 2.08)	(6.46, 3.96)	(7.08, 7.71)	(5.52, 7.50)	(2.81, 7.60)
$\vec{R}_i$	(3.54, 4.17)	(1.46, 6.88)	(3.02, 4.79)	(4.38, 1.67)	(7.50, 5.00)	(7.19, 6.67)	(5.52, 6.46)	(3.23, 8.23)
$\vec{T}_i$	(0.99, 0.01, 0)	(0, 1, 0)	(0.01, 0.99, 0)	(0, 0.03, 0.97)	(0, 0.97, 0.03)	(1, 0, 0)	(0, 0, 1)	(0, 0, 1)

TABLE III

THE RULE PARAMETERS EXTRACTED BY SSNFS ON THE IRIS DATA.

	Rule 1	*Rule 2*	*Rule 3*	*Rule 4*	*Rule 5*
$\vec{P}_i$	(5.19, 3.54, 1.42, 0.24)	(6.60, 2.64, 3.49, 1.23)	(6.60, 2.64, 4.72, 1.23)	(7.64, 2.64, 5.75, 2.08)	(7.64, 2.64, 6.60, 2.08)
$\vec{R}_i$	(4.50, 2.30, 1.30, 0.30)	(4.90, 2.40, 3.30, 1)	(4.90, 2.50, 4.50, 1.70)	(6.30, 3.40, 5.60, 2.40)	(7.90, 3.80, 6.40, 2.00)
$\vec{T}_i$	(1, 0, 0)	(0, 1, 0)	(0, 0.58, 0.42)	(0, 0, 1)	(0, 0, 1)

TABLE IV

THE CLASS CENTROIDS OBTAINED BY $\epsilon = 0.02$ AND $\epsilon = 0.05$.

	Iris setosa	*Iris versicolour*	*Iris virginica*
Actual	(5.006, 3.428, 1.462, 0.246)	(5.936, 2.770, 4.260, 1.326)	(6.588, 2.974, 5.552, 2.026)
$\epsilon = 0.05$	(5.012, 3.536, 1.460, 0.245)	(5.836, 2.731, 4.389, 1.332)	(6.848, 3.078, 5.632, 2.058)
$\epsilon = 0.02$	(5.007, 3.425, 1.473, 0.250)	(5.882, 2.752, 4.353, 1.421)	(6.832, 3.061, 5.611, 2.052)

robust. However, there are many challenging theoretical problems remaining open to further research. For instance, rule generalization in SSNFS, convergence property, rule-patch validation in the context of soft computing, etc.

REFERENCES

[1] H. R. Berenji and P. Khedkar, "Learning and tuning fuzzy logic controllers through reinforcements," *IEEE Trans. Neural Networks,* vol. 3, no. 5, pp. 724-740, 1992.

[2] V. Cherkassky and P. Mulier, *Learning From Data.* John Wiley & Sons, Inc. New York, 1998.

[3] W. Duch, R. Adamczak and K. Grąbczewski, "A new methodology of extraction, optimization and application of crisp and fuzzy rules," *IEEE Trans. Neural Networks,* vol.12, no.2, pp.277-306, March 2001.

[4] W. Duch, R. Setiono and J.M. Żurada, "Computational intelligence methods for rule-based data understanding," *Proc. IEEE,* vol.92, no.5, pp.771-805, May 2004.

[5] H. Frigui and R. Krishnapuram, "A robust competitive clustering algorithm with applications in computer vision," *IEEE Trans. Patt. Anal. Machine Intell.,* vol. 21, no. 5, pp. 450-465, May 1999.

[6] N. Tschichold-Gürman, "Generation and improvement of fuzzy classifiers with incremental learning using fuzzy rulenet," in K. M. Geroge, J. H. Carrol, E. Deaton, D. Oppenheim, and J. Hightower, eds, Applied Computing 1995. *Proc. 1995 ACM Symposium on Applied Computing,* pp. 466-470, Feb. 26-28, ACM Press, New York, 1995.

[7] J. R. Jang, "ANFIS: Adaptive-Network-Based Fuzzy Inference System," *IEEE Trans. Syst., Man, Cybern.,* vol. 23, no. 3, pp. 665-685, 1993.

[8] J. M. Keller and H. Tahani, "Implementation of conjunctive and disconjunctive fuzzy logic rules with neural networks," *Int. J. Approximate Reasoning,* vol. 6, pp. 221-240, Feb., 1992.

[9] J. M. Keller, R. R. Yager, and H. Tahani, "Neural network implementation of fuzzy logic," *Fuzzy Sets and Systems,* vol. 45, pp. 1-12, 1992.

[10] J. M. Keller and H. Tahani, "Backpropagation neural networks for fuzzy logic," *Information Science,* vol 62, pp205-221, 1992.

[11] R. Kruse and J. Gebhardt, *Foundations of Fuzzy Systems,* Wiley, Chichester, 1994

[12] J. MacQueen, "Some methods for classification and analysis of multivariate observations," *Proc. 5th Berkeley Symposium on Mathematical Statistics and Probability,* pp. 281-297, Berkeley, 1967, University of California Press.

[13] M. C. Mackey and L. Glass, "Oscillation and chaos in physiological control systems," *Science,* vol. 197, pp. 287-289, July, 1977.

[14] S. Mitra and L. Kuncheva, "Improving classification performance using fuzzy map and two level selective partitioning of the feature space," *Fuzzy Sets and Systems,* vol 70., pp. 1-13, 1995.

[15] D. Nauck and R. Kruse, "NEFCLASS-A neuro-fuzzy approach for the classification of data," *Applied Computing,* K. M. George, J. H. Carrol, E. Deaton, D. Oppenheim, and J. Hightower, eds., 1995. In Proc. Of the 1995 ACM Symposium On Applied Computing, Nashville, Feb. 26-28, pp. 461-465, ACM Press: New York, 1995.

[16] D. Nauck, "Neuro-Fuzzy systems: review and prospects," *Proc. the 5th European Congress on Intelligent Techniques and Soft Computing (EUFIT'97),* pp. 1044-1053, Aachen, Sep. 8-11, 1997.

[17] S. K. Pal and S. Mitra, "Multi-layer perceptron, fuzzy sets and classification," *IEEE Trans. Neural Networks,*

[18] M. Setnes, "Supervised fuzzy clustering for rule extraction," *IEEE Trans. Fuzzy Syst.,* vol. 8, no. 4, pp. 416-424, 2000.

[19] H. G. Zimmermann, R. Neuneier, H. Dichtl, and S. Siekmann, "Modeling the german stock index dax with neuo-fuzzy," *Proc. Fourth European Congress on Intelligent Techniques and Soft Computing (EUFIT96),* Aachen, Sep., 1996.