

# Iterative Inversion of Fuzzified Neural Networks

Sungwoo Park and Taisook Han

*Abstract*— The inversion of a neural network is a process of computing inputs which produce a given target when fed into the neural network. The inversion algorithm of crisp neural networks is based on the gradient descent search in which a candidate inverse is iteratively refined to decrease the error between its output and the target. In this paper, we derive an inversion algorithm of fuzzified neural networks from that of crisp neural networks. First we present a framework of learning algorithms of fuzzified neural networks and introduce the idea of adjusting schemes for fuzzy variables. Next we derive the inversion algorithm of fuzzified neural networks by applying the adjusting scheme for fuzzy variables to total inputs in the input layer. Finally we make three experiments on the parity-3 problem; we examine the effect of the size of training sets on the inversion and investigate how the fuzziness of inputs and targets of training sets affects the inversion.

*Keywords*— Fuzzified Neural Network, Learning Algorithm, Inversion Algorithm

## I. INTRODUCTION

Fuzzy neural networks are network architectures designed to process fuzzy data. The largest class of fuzzy neural networks is trainable fuzzy rule-based systems with network architectures (Horikawa *et al.* [1], Jang [2], Kwan and Cai [3], and Shann and Fu [4]). Another class of fuzzy neural networks is fuzzified neural networks, which are derived from regular crisp neural networks by substituting fuzzy neurons for crisp neurons (Ishibuchi *et al.* [5] and Hayashi *et al.* [6]). Buckley and Hayashi [7] classifies fuzzified neural networks into three types:  $FNN_1$  with real inputs and fuzzy weight factors;  $FNN_2$  with fuzzy inputs and real weight factors;  $FNN_3$  with fuzzy inputs and fuzzy weight factors.

There have been several researches on the learning algorithm of fuzzified neural networks. Hayashi *et al.* [6] and Buckley and Hayashi [8] present the fuzzified delta rule by replacing real variables in the generalized delta rule of Rumelhart *et al.* [9] by fuzzy variables, but it cannot be used practically because of the lack of theoretical support. Feuring [10] proposes a learning algorithm designed for triangular weight factors. Ishibuchi *et al.* [11], [12] propose an architecture of fuzzified neural networks in which weight factors are real numbers. Ishibuchi *et al.* [13], [14] use symmetric triangular fuzzy numbers for weight factors. Ishibuchi and Nii [15] use non-symmetric trapezoidal fuzzy numbers for weight factors. Duniak and Wunsch [16] propose a transformation which does not simplify the representation of weight factors.

Sungwoo Park is with School of Computer Science, Carnegie Mellon University. [gla+@cs.cmu.edu](mailto:gla+@cs.cmu.edu)

Taisook Han is with Department of Computer Science, Korea Advanced Institute of Science and Technology (KAIST), Taejeon, Korea. [han@cs.kaist.ac.kr](mailto:han@cs.kaist.ac.kr)

This work was partially supported by the KOSEF through the AITrc.

A neural network can be regarded as representing a function  $F$  determined by its weight factors and network architecture. Usually we train a neural network with a training set, present inputs to the neural network, and interpret the outputs according to the logical rules in the training set. In contrast, the inversion is a process of computing inputs which produce a given target when fed into the neural network; it evaluates the inverse function  $F^{-1}$  for the target. In other words, the inversion answers the question: “Which input must be fed into the neural network to produce the given target?”

If the neural network simulates a many-to-one function, a target may have more than one inverse. Furthermore it requires a large computational cost to find an exact inverse of the target even on neural networks of moderate size. For these reasons, the inversion yields estimate inverses which approximate the inverse of the target to a certain degree.

Linden and Kindermann [17] introduce the inversion algorithm of crisp neural networks. It is based on the gradient descent search in which a candidate inverse is iteratively refined to decrease the error between its output and the target. The inversion of crisp neural networks has been used in several applications (Hoskins *et al.* [18], Davis *et al.* [19], Hwang and Chan [20], and Hwang *et al.* [21]).

To our knowledge, the inversion of fuzzified neural networks has not been investigated in the literature. In this paper, we derive an inversion algorithm of fuzzified neural networks from that of crisp neural networks. It is also based on the gradient descent search. Since we do not have the calculus for fuzzy numbers, the inversion algorithm of crisp neural networks cannot be directly fuzzified; hence, we introduce the idea of adjusting scheme for fuzzy variables. An adjusting scheme for fuzzy variables tells how to adjust fuzzy variables in the gradient descent search.

We first present a general framework of learning algorithms of fuzzified neural networks which is based on the generalized delta rule. This framework uses adjusting schemes for fuzzy variables to adjust weight factors. Then we derive adjusting schemes for fuzzy variables from existing learning algorithms which fit into this framework. Finally we apply adjusting schemes for fuzzy variables to total inputs in the input layer and obtain the inversion algorithm of fuzzified neural networks.

With three adjusting schemes for fuzzy variables, we make three experiments on the parity-3 problem. The first experiment demonstrates the learning and inversion of fuzzified neural networks. In the second experiment, we examine the effect of the size of training sets on the inversion. In the third experiment, we investigate how the fuzziness of inputs and targets of training sets affects the inversion.

The inversion of fuzzified neural networks has several

applications. For example, a fuzzy controller system implemented with fuzzified neural networks can benefit from the inversion in finding control signals for desired operations. The reliability of fuzzy systems based on fuzzified neural networks can also be examined by the inversion.

This paper is organized as follows. We explain the architecture and operation of fuzzified neural networks in Section II. We derive adjusting schemes for fuzzy variables in Section III. We derive the inversion algorithm of fuzzified neural networks in Section IV. We analyze experimental results in Section V. We summarize this paper in Section VI.

## II. FUZZIFIED NEURAL NETWORKS

In this section, we first give an overview of fuzzy arithmetic and interval arithmetic.<sup>1</sup> Next we describe the architecture and operation of fuzzified neural networks.

### A. Fuzzy Numbers

Fuzzy numbers are continuous fuzzy sets defined on  $\mathcal{R}$ , the set of real numbers. A membership function  $\mu_A : \mathcal{R} \rightarrow [0, 1]$  gives a unique fuzzy number  $A$  in  $\hat{\mathcal{R}}$ , the set of fuzzy numbers defined on  $\mathcal{R}$ . A normal triangular-shaped fuzzy number  $A$  is a fuzzy number such that

$$\alpha_1 > \alpha_2 \iff A^{\alpha_1} \subset A^{\alpha_2} \quad , \quad \alpha_1, \alpha_2 \in [0, 1] \quad (1)$$

where  $A^{\alpha_1}$  and  $A^{\alpha_2}$  are the  $\alpha_1$ -level set and the  $\alpha_2$ -level set of  $A$ , respectively.<sup>2</sup> Kaufmann and Gupta [22] define a fuzzy number as a convex and normal fuzzy set defined on a real line, which is a normal triangular-shaped fuzzy number in our terminology. We consider only normal triangular-shaped fuzzy numbers throughout this paper.

Basic fuzzy number operations are obtained through applying the extension principle of Zadeh [22] to numerical operations. For fuzzified neural networks, we need three operations: addition, multiplication, and monotonic function mapping. These operations are defined by the extension principle as

$$\mu_{A+B}(z) = \sup\{\min(\mu_A(x), \mu_B(y)) \mid z = x + y\} \quad (2)$$

$$\mu_{A \cdot B}(z) = \sup\{\min(\mu_A(x), \mu_B(y)) \mid z = xy\} \quad (3)$$

$$\mu_{f(A)}(z) = \sup\{\mu_A(x) \mid z = f(x)\} \quad (4)$$

where  $A, B$  are fuzzy numbers,  $f : \mathcal{R} \rightarrow \mathcal{R}$  is a monotonically increasing function, and  $\hat{f} : \hat{\mathcal{R}} \rightarrow \hat{\mathcal{R}}$  is a fuzzy function extended from  $f$ .

An interval  $P$  is defined as a set of real numbers contained in a range:

$$P = [P_L, P_R] = \{x \in \mathcal{R} \mid P_L \leq x \leq P_R\} \quad (5)$$

<sup>1</sup>In this paper, we denote fuzzy numbers and fuzzy functions by sans serif letters (for example,  $A, X, f$ ), real numbers and real-valued functions by italic letters (for example,  $a, x, f$ ), and intervals and interval functions by typewriter letters (for example,  $A, X, \mathbf{f}$ ).

<sup>2</sup>This definition of normal triangular-shaped fuzzy numbers includes trapezoidal fuzzy numbers.

A real number  $a$  can be regarded as a particular interval  $[a, a]$ . Basic interval operations are addition, multiplication, and monotonic function mapping:

$$P + Q = [P_L + Q_L, P_R + Q_R] \quad (6)$$

$$P \cdot Q = [\min\{P_L Q_L, P_L Q_R, P_R Q_L, P_R Q_R\}, \max\{P_L Q_L, P_L Q_R, P_R Q_L, P_R Q_R\}] \quad (7)$$

$$\mathbf{f}(P) = [f(P_L), f(P_R)] \quad (8)$$

where  $f : \mathcal{R} \rightarrow \mathcal{R}$  is a monotonically increasing function and  $\mathbf{f}$  is an interval function extended from  $f$ . If  $Q_L > 0$ , that is, the interval  $Q$  consists only of positive numbers, (7) can be rewritten as

$$P \cdot Q = [\min\{P_L Q_L, P_L Q_R\}, \max\{P_R Q_L, P_R Q_R\}] \quad (9)$$

A fuzzy number has an infinite number of level sets, which makes it difficult to devise a precise representation scheme of fuzzy numbers. For this reason, we approximate fuzzy numbers with their  $h$ -level sets on predefined levels  $h = h_1, \dots, h_v$ ,  $0 \leq h_1 < \dots < h_v \leq 1$ . The set of predefined levels is determined according to the desired representation precision. In this simplified representation scheme, we can implement the three fuzzy number operations only with interval arithmetic:

$$(A + B)^h = A^h + B^h \quad (10)$$

$$(A \cdot B)^h = A^h \cdot B^h \quad (11)$$

$$\mathbf{f}(A)^h = \mathbf{f}(A^h) \quad (12)$$

For the implementation of fuzzy arithmetic, see Anile *et al.* [23].

### B. Architecture of Fuzzified Neural Networks

A fuzzy neuron is a computation unit which consists of a set of fuzzy input variables ( $X_1, X_2, \dots, X_n$ ), a set of fuzzy weight factors ( $W_1, W_2, \dots, W_n$ ), and a fuzzy activation function  $f : \hat{\mathcal{R}} \rightarrow \hat{\mathcal{R}}$  extended from a monotonically increasing real-valued function. It receives fuzzy signals as inputs and computes the total input  $\mathbf{Net}$  and the output  $\mathbf{O}$  as follows:

$$\mathbf{Net} = \sum_{i=1}^n X_i \cdot W_i \quad (13)$$

$$\mathbf{O} = \mathbf{f}(\mathbf{Net}) \quad (14)$$

That is, it first computes the weighted sum of inputs and then produces the function value of the weighted sum on the activation function  $f$  as its final result. We adopt the standard sigmoidal function:

$$\mathbf{f}(\mathbf{Net}) = \frac{1}{1 + \exp(-\mathbf{Net})} \quad (15)$$

A fuzzified neural network is derived from a regular crisp neural network by substituting fuzzy neurons for crisp neurons (Ishibuchi *et al.* [5] and Hayashi *et al.* [6]). It is composed of one input layer, one output layer, and one or more

hidden layers, as shown in Figure 1. Each layer has one or more neurons, and a special bias neuron is attached to every layer except the output layer. Every neuron except bias neurons and those in the input layer maintains a weight factor for each neuron in the preceding layer.

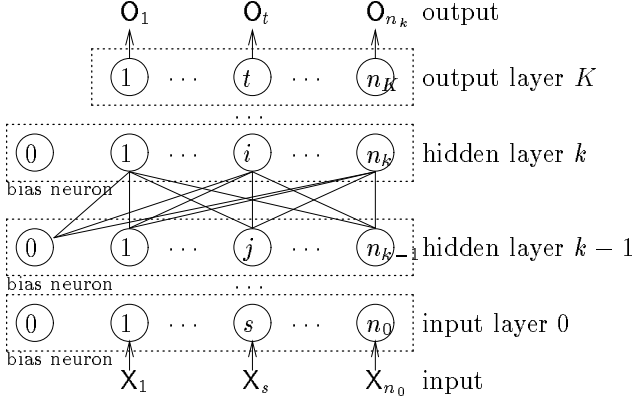


Fig. 1. The architecture of fuzzified neural networks with  $K+1$  layers

When an  $n_0$ -dimensional fuzzy vector  $\vec{X} = (X_1, X_2, \dots, X_{n_0})$  is presented to a fuzzified neural network in Figure 1, the input-output relation of each neuron is obtained from (13)–(14) as follows:

$$O_{0s} = X_s, \quad s = 1, 2, \dots, n_0 \quad (16)$$

$$O_{ki} = f(\text{Net}_{ki}) \quad , \quad 1 \leq k \leq K, \quad i = 1, 2, \dots, n_k$$

$$= f\left(\sum_{j=0}^{n_{k-1}} O_{(k-1)j} \cdot W_{kij}\right) \quad (17)$$

$$O_{k0} = 1 \quad , \quad 0 \leq k \leq K-1 \quad (18)$$

In (16)–(18),  $\text{Net}_{ki}$  and  $O_{ki}$  are the total input and the final output of the neuron  $i$  in the layer  $k$ , respectively.  $W_{kij}$  is the weight factor of the neuron  $i$  in the layer  $k$  for the neuron  $j$  in the layer  $k-1$ . (16) says that the input to a neuron in the input layer becomes the output of the neuron, and (18) says that the output of every bias neuron is the real number 1.

Since we approximate fuzzy numbers with their level sets on predefined levels, we use the input-output relation defined in terms of interval arithmetic to implement fuzzified neural networks:

$$O_{0s}^h = X_s^h, \quad s = 1, 2, \dots, n_0 \quad (19)$$

$$O_{ki}^h = \mathbf{f}(\text{Net}_{ki}^h) \quad , \quad 1 \leq k \leq K, \quad i = 1, 2, \dots, n_k$$

$$= \mathbf{f}\left(\sum_{j=0}^{n_{k-1}} O_{(k-1)j}^h \cdot W_{kij}^h\right) \quad (20)$$

$$O_{k0}^h = 1 \quad , \quad 0 \leq k \leq K-1 \quad (21)$$

where  $h \in [0, 1]$  is a level value.

### III. LEARNING OF FUZZIFIED NEURAL NETWORKS

A learning algorithm specifies how to adjust weight factors from the logical rules in a training set. In this section,

we first present a general framework of learning algorithms of fuzzified neural networks which is based on the generalized delta rule of Rumelhart *et al.* [9]. Next we introduce adjusting schemes for fuzzy variables and show that specific learning algorithms are obtained by combining adjusting schemes for fuzzy variables with the framework of learning algorithms.

#### A. Framework of Learning Algorithms

Suppose that an input fuzzy vector  $\vec{X} = (X_1, X_2, \dots, X_{n_0})$  and an associated target fuzzy vector  $\vec{T} = (T_1, T_2, \dots, T_{n_K})$  are presented to a learning algorithm of the fuzzified neural network in Figure 1. We assume that  $X_s, s = 1, \dots, n_0$ , is defined on  $[0, 1]$ , that is,  $X_s^0 \subseteq [0, 1]$ . We do not lose the generality of input fuzzy vectors by this assumption, for any fuzzy number  $A \in \hat{\mathcal{R}}$  can be converted to another unique fuzzy number  $A_+$  defined on  $[0, 1]$  by taking  $A_+ = f(A)$  where  $f$  is the sigmoidal function.  $T_t, t = 1, \dots, n_K$ , must also be defined on  $[0, 1]$  to be a valid component of  $\vec{T}$  because the output of fuzzy neurons is fuzzy numbers defined on  $[0, 1]$ . The output  $\vec{O}$  for the input  $\vec{X}$  can be written as

$$\vec{O} = (O_{K1}, O_{K2}, \dots, O_{Kn_K})$$

$$= F(X_1, X_2, \dots, X_{n_0}) \quad , \quad F: \hat{\mathcal{R}}^{n_0} \rightarrow \hat{\mathcal{R}}^{n_K} \quad (22)$$

To introduce the gradient descent search to the learning algorithm, we should have a cost function  $e: \hat{\mathcal{R}} \times \hat{\mathcal{R}} \rightarrow \mathcal{R}$  computing the difference between two fuzzy numbers. To this end, we define a cost function  $e_h: \hat{\mathcal{R}} \times \hat{\mathcal{R}} \rightarrow \mathcal{R}$  for each level  $h = h_1, \dots, h_v$  which computes the difference between the  $h$ -level sets of two fuzzy numbers. Examples of  $e_h$  (Krishnamraju *et al.* [24] and Ishibuchi *et al.* [5]) are

$$e_h(A, B) = \frac{(A^h_L - B^h_L)^2 + (A^h_R - B^h_R)^2}{2} \quad (23)$$

$$e_h(A, B) = \frac{h}{2}((A^h_L - B^h_L)^2 + (A^h_R - B^h_R)^2) \quad (24)$$

Then we can obtain the total error  $E$  between the output  $\vec{O}$  and the target  $\vec{T}$  by

$$E = \sum_{t=1}^{n_K} e(O_{Kt}, T_t)$$

$$= \sum_{t=1}^{n_K} \sum_{h=h_1, \dots, h_v} e_h(O_{Kt}, T_t) \quad (25)$$

The learning algorithm adjusts weight factors in order to minimize  $E$ . Consider a weight factor  $W$ .  $W$  is represented by a set of  $2v$  real variables

$$\{W^{h_1}_L, W^{h_1}_R, \dots, W^{h_v}_L, W^{h_v}_R\} \quad (26)$$

for which the following constraint holds:

$$W^{h_1}_L \leq W^{h_2}_L \leq \dots \leq W^{h_v}_L \leq W^{h_v}_R \leq \dots \leq W^{h_2}_R \leq W^{h_1}_R \quad (27)$$

We can obtain a new value of  $W$  by taking incremental changes  $\Delta W^h_L$  and  $\Delta W^h_R$  proportional to  $-\frac{\partial E}{\partial W^h_L}$  and

$-\frac{\partial E}{\partial W^h_R}$ , respectively, for the level  $h$ :

$$\Delta W^h_L = -\eta \frac{\partial E}{\partial W^h_L} \quad (28)$$

$$\Delta W^h_R = -\eta \frac{\partial E}{\partial W^h_R} \quad (29)$$

where  $\eta$  is a positive constant. Ishibuchi *et al.* [5] give a detailed calculation of the partial derivatives. In Appendix, we reformulate the calculation for fuzzified neural networks with more than one hidden layer.

Although incremental changes  $\Delta W^h_L$  and  $\Delta W^h_R$  in (28)–(29) decrease the total error  $E$ , the resultant weight factor may not be a valid fuzzy number as illustrated in Figure 2; it does not satisfy the constraint in (27). We must therefore take  $\Delta W^h_L$  and  $\Delta W^h_R$  such that the resultant weight factor does not violate the constraint.

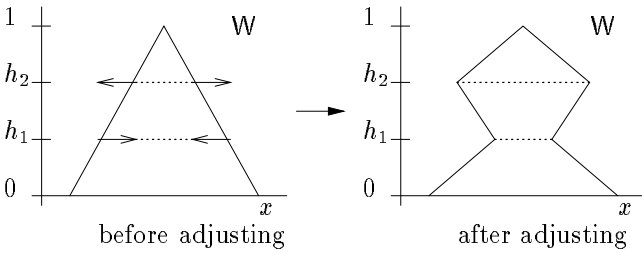


Fig. 2. An abnormal case in adjusting weight factors

There have been several researches on the learning algorithm of fuzzified neural networks which fits into this framework. Ishibuchi *et al.* [11], [12] propose an architecture of fuzzified neural networks in which weight factors are real numbers. Ishibuchi *et al.* [13], [14] use symmetric triangular fuzzy numbers for weight factors. Ishibuchi and Nii [15] use non-symmetric trapezoidal fuzzy numbers for weight factors. Their algorithms all simplify the representation of weight factors, that is, weight factors are represented with less parameters than in the original representation scheme. Duniyakh and Wunsch [16] propose a transformation which does not simplify the representation of weight factors; their algorithm allows general fuzzy numbers to be used for weight factors. In the next subsection, we interpret these algorithms in a consistent way by defining corresponding adjusting schemes for fuzzy variables.

### B. Adjusting Schemes for Fuzzy Variables

An adjusting scheme for fuzzy variables tells how to adjust a fuzzy variable or weight factor  $W$  and decrease  $E$  when given partial derivatives  $\frac{\partial E}{\partial W^h_L}$  and  $\frac{\partial E}{\partial W^h_R}$ ,  $h = h_1, \dots, h_v$ . It is defined by a one-to-one correspondence function  $g : \mathcal{R}^z \rightarrow \mathcal{R}^{2v}$  which specifies a unique value of  $W$  for a set of  $z$  real parameters  $P = \{p_1, p_2, \dots, p_z\}$  by

$$g(p_1, p_2, \dots, p_z) = (W^{h_1_L}, W^{h_1_R}, \dots, W^{h_v_L}, W^{h_v_R}) \quad (30)$$

The function  $g$  should be defined in such a way that we can calculate  $\frac{\partial E}{\partial p_i}$ ,  $i = 1, \dots, z$ , from  $\frac{\partial E}{\partial W^h_L}$  and  $\frac{\partial E}{\partial W^h_R}$ . For  $W$  to have a valid fuzzy number, the set  $P$  may have to satisfy some constraint. Depending on  $g$ , the adjusting scheme

may assume fuzzy variables to be of a particular type of fuzzy numbers.

With an adjusting scheme, we can obtain a new value of  $W$  by adjusting the parameters in  $P$  instead of  $W^h_L$  and  $W^h_R$ . First we calculate  $\frac{\partial E}{\partial p_i}$ . Next we obtain new values for  $P$  by

$$\Delta p_i = -\eta \frac{\partial E}{\partial p_i} \quad (31)$$

where  $\eta$  is a positive constant. If the set  $P$  has a constraint, (31) may cause the new values for  $P$  to violate the constraint. In such cases, we must provide an algorithm or a heuristic to keep the constraint. Finally we obtain a new value of  $W$  by (30).

An adjusting scheme combined with the framework of learning algorithms gives a specific learning algorithm; conversely, we can derive an adjusting scheme from any learning algorithm which fits into the framework. For example, an adjusting scheme defined by  $g : \mathcal{R} \rightarrow \mathcal{R}^{2v}$  with  $P = \{p\}$  such that

$$W^h_L = W^h_R = p \quad (32)$$

gives the learning algorithm for real weight factors in Ishibuchi *et al.* [11], [12]. The partial derivative  $\frac{\partial E}{\partial p}$  is obtained by

$$\frac{\partial E}{\partial p} = \sum_{h=h_1, \dots, h_v} \left( \frac{\partial E}{\partial W^h_L} + \frac{\partial E}{\partial W^h_R} \right) \quad (33)$$

Since no constraint is imposed on  $P$ , we may adjust  $p$  by

$$\Delta p = -\eta \frac{\partial E}{\partial p} \quad (34)$$

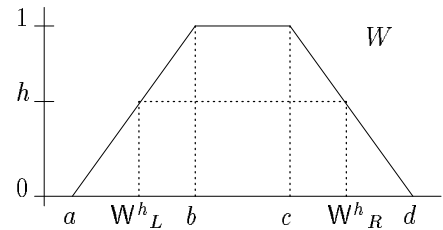


Fig. 3. A trapezoidal fuzzy number

As another example, consider the learning algorithm for non-symmetric trapezoidal weight factors in Ishibuchi and Nii [15]. The corresponding adjusting scheme is defined by  $g : \mathcal{R}^4 \rightarrow \mathcal{R}^{2v}$  with  $P = \{a, b, c, d\}$  such that

$$W^h_L = (1-h)a + hb \quad (35)$$

$$W^h_R = hc + (1-h)d \quad (36)$$

$$a \leq b \leq c \leq d \quad (37)$$

$P$  specifies a unique trapezoidal fuzzy number for  $W$  as shown in Figure 3. From the chain rule, we have

$$\frac{\partial E}{\partial a} = \sum_{h=h_1, \dots, h_v} (1-h) \frac{\partial E}{\partial W^h_L} \quad (38)$$

$$\frac{\partial E}{\partial b} = \sum_{h=h_1, \dots, h_v} h \frac{\partial E}{\partial W^h_L} \quad (39)$$

$$\frac{\partial E}{\partial c} = \sum_{h=h_1, \dots, h_v} h \frac{\partial E}{\partial W^h_R} \quad (40)$$

$$\frac{\partial E}{\partial d} = \sum_{h=h_1, \dots, h_v} (1-h) \frac{\partial E}{\partial W^h_R} \quad (41)$$

from which we calculate  $\Delta a$ ,  $\Delta b$ ,  $\Delta c$ , and  $\Delta d$ . The resulting set  $P' = \{a + \Delta a, b + \Delta b, c + \Delta c, d + \Delta d\}$  may not, however, satisfy the constraint in (37). For this reason, we provide a heuristic for adjusting the parameters to keep the constraint. To reorder the parameters by sorting the four new values in  $P'$  proves to be a good candidate for the heuristic, which is demonstrated in Section V-A.

The transformation of Duniak and Wunsch [16] gives an adjusting scheme defined by  $g : \mathcal{R}^{2v} \rightarrow \mathcal{R}^{2v}$  with  $P = \{s, p_1, p_2, \dots, p_{v-1}, q_v, \dots, q_1\}$  such that

$$W^{h_1}_L = s \quad (42)$$

$$W^{h_{l+1}}_L = W^{h_l}_L + p_l^2, \quad 1 \leq l \leq v-1 \quad (43)$$

$$W^{h_v}_R = W^{h_v}_L + q_v^2 \quad (44)$$

$$W^{h_m}_R = W^{h_{m+1}}_R + q_m^2, \quad 1 \leq m \leq v-1 \quad (45)$$

The partial derivatives for the parameters are calculated from the chain rule as follows:

$$\frac{\partial E}{\partial s} = \frac{\partial E}{\partial W^{h_1}_L} + \sum_{u=2}^v \frac{\partial E}{\partial W^{h_u}_L} + \sum_{w=1}^v \frac{\partial E}{\partial W^{h_w}_L} \quad (46)$$

$$\frac{\partial E}{\partial p_l} = 2p_l \left( \sum_{u=l+1}^v \frac{\partial E}{\partial W^{h_u}_L} + \sum_{w=1}^v \frac{\partial E}{\partial W^{h_w}_L} \right), \quad 1 \leq l \leq v-1 \quad (47)$$

$$\frac{\partial E}{\partial q_m} = 2q_m \sum_{w=1}^m \frac{\partial E}{\partial W^{h_w}_L}, \quad 1 \leq m \leq v \quad (48)$$

$P$  has no constraint and always gives a valid fuzzy number for  $W$ . Note that the set  $P$  does not change if all the parameters  $p_l$  and  $q_m$  are set to 0.

One reason for introducing the idea of adjusting scheme for fuzzy variables is to formulate the existing learning algorithms in a single framework. Another reason is that it is essential to the inversion algorithm of fuzzified neural networks, which is shown in the next section.

#### IV. INVERSION OF FUZZIFIED NEURAL NETWORKS

In this section, we first explain the problem of inversion. Next we derive the inversion algorithm of fuzzified neural networks from that of crisp neural networks.

##### A. Introduction

Consider a fuzzified neural network in Figure 1. After it learns the rules in a training set a sufficient number of times so as to reach a stable state, it approximates a fuzzy function  $F : \hat{\mathcal{R}}^{n_0} \rightarrow \hat{\mathcal{R}}^{n_K}$  which complies with the rules. The inversion is a process of computing the inverse of a target  $\vec{T} = (T_1, \dots, T_{n_K})$  on the function  $F$ . In other words, it computes an input  $\vec{X} = (X_1, \dots, X_{n_0})$  which produces the desired target  $\vec{T}$  when fed into the fuzzified neural network:

$$F(X_1, \dots, X_{n_0}) = (T_1, \dots, T_{n_K}) \quad (49)$$

If the fuzzified neural network simulates a many-to-one function, the target  $\vec{T}$  may have more than one inverse. Furthermore it requires a large computational cost to find the exact inverse of  $\vec{T}$  even on fuzzified neural networks of moderate size. For these reasons, we instead find an estimate inverse  $\vec{X}$  which approximates the inverse of  $\vec{T}$  to a certain degree as follows:

$$F(X_1, \dots, X_{n_0}) \approx (T_1, \dots, T_{n_K}) \quad (50)$$

Linden and Kindermann [17] introduce the inversion algorithm of crisp neural networks. It is based on the gradient descent search in which a candidate inverse is iteratively refined to decrease the error between its output and the target; it adjusts repeatedly the candidate inverse using error signals backpropagated from the output layer. The error signals are backpropagated to the input layer in the same way as in the learning algorithm. The inversion of crisp neural networks has been used in several applications (Hoskins *et al.* [18], Davis *et al.* [19], Hwang and Chan [20], and Hwang *et al.* [21]).

As with crisp neural networks, the inversion algorithm of fuzzified neural networks is based on the gradient descent search. Since we do not have the calculus for fuzzy numbers, the inversion algorithm of crisp neural networks cannot be directly fuzzified. Therefore the inversion algorithm of fuzzified neural networks uses adjusting schemes for fuzzy variables to adjust estimate inverses, which is explained in the next subsection.

##### B. Computing the Inverse

Suppose that we want to find the inverse of a target  $\vec{T} = (T_1, \dots, T_{n_K})$ . Let  $\vec{X} = (X_1, \dots, X_{n_0})$  be an estimate inverse. We assume that  $T_t$ ,  $t = 1, \dots, n_K$ , and  $X_s$ ,  $s = 1, \dots, n_0$ , are defined on  $[0, 1]$ . The output  $\vec{O} = (O_{K1}, \dots, O_{Kn_K})$  is given by

$$\vec{O} = F(X_1, \dots, X_{n_0}) \quad (51)$$

The total error  $E$  between  $\vec{T}$  and  $\vec{O}$  is computed by (25).

We note from (16) that  $\vec{X}$  becomes the output of the input layer without modification. In another perspective, the input layer can be considered, like other layers, to maintain the total input  $\text{Net}_{0s}$  from which we compute  $X_s$  by

$$X_s = f(\text{Net}_{0s}) \quad (52)$$

where  $f$  is the standard sigmoidal function. We can also obtain a unique value of  $\text{Net}_{0s}$  from  $X_s$  by

$$\begin{aligned} \text{Net}_{0s} &= f^{-1}(X_s) \\ &= -\ln \left( \frac{1}{X_s} - 1 \right) \end{aligned} \quad (53)$$

Therefore, to obtain a new estimate inverse of  $\vec{T}$ , we may adjust the total inputs in the input layer instead of  $\vec{X}$ .

In (65)–(66) (see Appendix), we define error signals  $\delta_{ki}^h$  and  $\delta_{ki}^h$ ,  $k = 1, \dots, K$ ,  $i = 1, \dots, n_k$ ,  $h = h_1, \dots, h_v$ . In

the inversion, we can compute error signals for the hidden layers and the output layer in the same way as in the backpropagation of the learning algorithm. Since  $X_s$  is defined on  $[0, 1]$ , the input layer is in the same condition as hidden layers. Therefore we can compute the error signals  $\delta_{0_s}^{h_L}$  and  $\delta_{0_s}^{h_R}$  for the input layer using (78)–(79) with  $k = 0$ . From  $\delta_{0_s}^{h_L}$  and  $\delta_{0_s}^{h_R}$ , we obtain partial derivatives  $\frac{\partial E}{\partial \text{Net}_{0_s}^{h_L}}$  and  $\frac{\partial E}{\partial \text{Net}_{0_s}^{h_R}}$ .

With  $\frac{\partial E}{\partial \text{Net}_{0_s}^{h_L}}$  and  $\frac{\partial E}{\partial \text{Net}_{0_s}^{h_R}}$  available, we employ an adjusting scheme for fuzzy variables to obtain a new value of  $\text{Net}_{0_s}$  which decreases  $E$ . The new value of  $\text{Net}_{0_s}$  gives in turn a new value of  $X_s$  by (52). Thus we can obtain a more accurate estimate inverse of  $\vec{T}$ .

Figure 4 describes the inversion algorithm of fuzzified neural networks. It assumes an adjusting scheme for fuzzy variables, but need not use the same adjusting scheme as in the learning algorithm because it does not modify weight factors. After initializing  $\text{Net}_{0_s}$  to a fuzzy number which can be specified by a valid set  $P$  in the adjusting scheme, it refines iteratively the total inputs in the input layer to decrease the total error  $E$  between the target and the output. Since we adjust only the total inputs, the inversion algorithm does not use (53). To compute error signals and partial derivatives, it uses the formulae shown in Appendix.

The inversion is expected to produce by (52) a sequence of inputs converging to a minimum in the input space, which is taken as the final estimate inverse of  $\vec{T}$ . Note that the quality of estimate inverses depends on the adjusting scheme because it specifies the type of fuzzy numbers of which the total inputs can be. It is completed when the output  $\vec{O}$  suffices the termination condition, that is,  $E$  goes below a positive constant  $\varepsilon$ . Experimental results of the inversion are given in the next section.

## V. EXPERIMENTS

We make three experiments about the learning and inversion of fuzzified neural networks. The first experiment demonstrates the learning and inversion of fuzzified neural networks. In the second experiment, we examine the effect of the size of training sets on the inversion. In the third experiment, we investigate how the fuzziness of inputs and targets of training sets affects the inversion.

In these experiments, every fuzzy number is represented with 11 level sets for  $h = 0.0, 0.1, \dots, 1.0$  ( $v = 11$ ). The learning algorithm uses the cost function  $e_h$  in (24). It proceeds in off-line mode: weight factors are modified only once for the whole training set at any iteration. (31) is also modified to include a momentum term:

$$\Delta p_i(r+1) = -\eta \frac{\partial E}{\partial p_i} + \alpha \Delta p_i(r) \quad (54)$$

where  $\Delta p_i(r)$  is the incremental change for the parameter  $p_i$  at the  $r$ -th iteration of the learning or inversion process.  $f$  and  $\mathbf{f}$  denote the standard sigmoidal functions for real numbers and fuzzy numbers, respectively.<sup>3</sup>

<sup>3</sup>All the experiments are performed on Silicon Graphics Octane, 195 MHZ R10000 CPU and R10010 FPU, 128 Mbytes main memory.

## Algorithm

### Iterative Inversion of Fuzzified Neural Networks

Input:

a target  $\vec{T} = (T_1, \dots, T_{n_K})$ , a fuzzified neural network

Output: an estimate inverse of  $\vec{T}$

```

initialize (Net01, ..., Net0n0) /* initialize the total inputs */
for 1 ≤ s ≤ n0 do
  Xs ← f(Net0s) /* compute an estimate inverse */
while true do
   $\vec{O} = (O_{K1}, \dots, O_{Kn_K}) \leftarrow F(X_1, \dots, X_{n_0})$  /* feedforward the input */
   $E \leftarrow \sum_{t=1}^{n_K} e(O_{Kt}, T_t)$  /* compute the total error */
  if E < ε then /* check the termination condition */
    break
  for k = K, ..., 0, 1 ≤ i ≤ nk, h = h1, ..., hv do
    compute  $\delta_{ki}^{h_L}$  and  $\delta_{ki}^{h_R}$  /* backpropagate error signals */
  for 1 ≤ s ≤ n0, h = h1, ..., hv do /* compute partial derivatives */
    compute  $\frac{\partial E}{\partial \text{Net}_{0s}^{h_L}}$  and  $\frac{\partial E}{\partial \text{Net}_{0s}^{h_R}}$  /* by (78)–(79) with k = 0 */
  for 1 ≤ s ≤ n0 do
    adjust Net0s /* adjust total inputs */
    Xs ← f(Net0s) /* obtain a new estimate inverse */
  endfor
endwhile
output  $\vec{X} = (X_1, \dots, X_{n_0})$  /* output the final estimate inverse */

```

End

Fig. 4. The inversion algorithm of fuzzified neural networks

### A. Experiment 1: Learning and Inversion of Fuzzified Neural Networks

In this experiment, we demonstrate the learning and inversion of fuzzified neural networks. We experiment on three adjusting schemes for both weight factors in the learning algorithm and total inputs in the input layer in the inversion algorithm. First we construct a fuzzified neural network for each adjusting scheme. It learns a training set for the parity-3 problem. Then we compute estimate inverses of a target on these fuzzified neural networks. We also explain how to obtain several estimate inverses of a given target.

#### A.1 Learning of Fuzzified Neural Networks

The training set for the parity-3 problem has eight rules, each of which consists of an input  $\vec{X} = (X_1, X_2, X_3)$  and an associated target  $\vec{T} = (T_1)$  as shown in Table I. Figure 5 shows the membership functions of fuzzy numbers **On** and **Off**.

We use three adjusting schemes for weight factors:

- (a)  $g : \mathcal{R} \rightarrow \mathcal{R}^{2v}$  defined by (32),  $P = \{p\}$
- (b)  $g : \mathcal{R}^4 \rightarrow \mathcal{R}^{2v}$  defined by (35)–(36),  $P = \{a, b, c, d\}$  with the reordering heuristic to keep the constraint in (37)
- (c)  $g : \mathcal{R}^{2v} \rightarrow \mathcal{R}^{2v}$  defined by (42)–(45),  $P = \{s, p_1, p_2, \dots, p_{v-1}, q_v, \dots, q_1\}$

For each adjusting scheme, we construct a fuzzified neural network in Figure 1 such that  $K = 2$ ,  $n_0 = 3$ ,  $n_1 = 3$ ,  $n_2 = 1$ . In adjusting schemes (a) and (b), weight factors are initialized to random real numbers in  $[-1, 1]$ . In the

Rule	X <sub>1</sub>	X <sub>2</sub>	X <sub>3</sub>	T <sub>1</sub>
1	Off	Off	Off	On
2	Off	Off	On	Off
3	Off	On	Off	Off
4	Off	On	On	On
5	On	Off	Off	Off
6	On	Off	On	On
7	On	On	Off	On
8	On	On	On	Off

TABLE I  
THE TRAINING SET FOR THE PARITY-3 PROBLEM

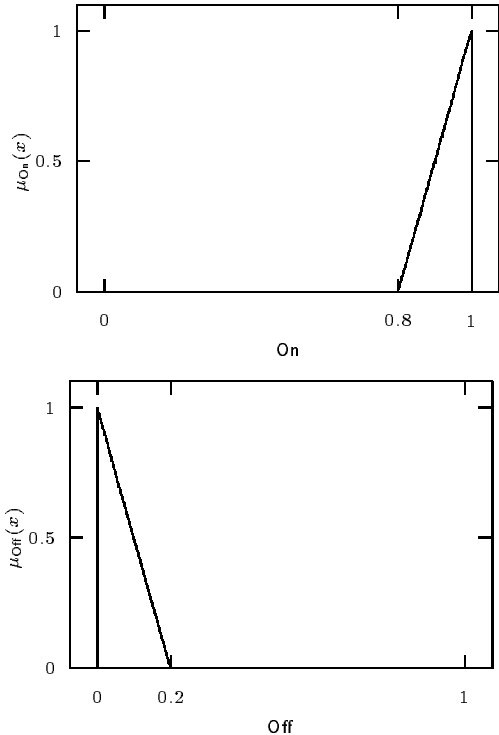


Fig. 5. The membership functions of On and Off

adjusting scheme (c), weight factors are initialized to triangular fuzzy numbers specified by  $(a-0.5, a, a+0.5)$  where  $a$  is a random real number in  $[-1, 1]$ .<sup>4</sup> We use  $\eta = 0.2$  and  $\alpha = 0.5$  in (54) for both learning and inversion.

Figure 6 shows the progress of learning for 10000 iterations in each adjusting scheme. The vertical axis denotes the sum of total errors for the whole training set on a logarithm scale. The learning times are: (a) 26.40 seconds; (b) 26.91 seconds; (c) 27.52 seconds. Although the adjusting scheme (c) maintains a set  $P$  with 22 parameters for each weight factor, its learning time is only 4.24% and 2.27% longer than those of adjusting schemes (a) and (b), respec-

<sup>4</sup>A triangular fuzzy number  $A$  specified by  $(p, q, r)$ ,  $p < q < r$ , is defined by

$$\mu_A(x) = \begin{cases} \frac{x-p}{q-p} & \text{if } x \in [p, q] \\ \frac{q-r}{x-r} & \text{if } x \in [q, r] \\ 0 & \text{otherwise} \end{cases} \quad (55)$$

tively. The reason is that  $\Delta W^h_L$  and  $\Delta W^h_R$  in (28)–(29) require a larger computational cost than  $\Delta p_i(r+1)$  in (54) and that  $\Delta W^h_L$  and  $\Delta W^h_R$  are computed in the same way in all the adjusting schemes: for  $\Delta W^h_L$  and  $\Delta W^h_R$ , (a) 25.91 seconds, (b) 25.89 seconds, and (c) 26.23 seconds; for  $\Delta p_i(r+1)$ , (a) 0.48 seconds, (b) 1.02 seconds, and (c) 1.27 seconds. The sums of total errors at the 10000-th iteration are: (a)  $\Sigma E = 0.0327687$ ; (b)  $\Sigma E = 0.0455238$ ; (c)  $\Sigma E = 0.0190321$ . The adjusting scheme (a), in which the set  $P$  has only one parameter and weight factors are real numbers, produces a smaller sum of total errors than the adjusting scheme (b). This is attributed in part to the characteristic of the parity-3 problem: the fuzziness of the input is equal to that of the target and thus real weight factors are adequate.

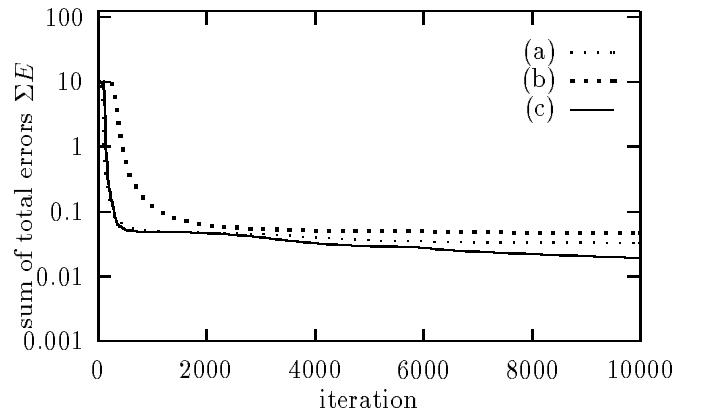


Fig. 6. The progress of learning in each adjusting scheme

After 10000 iterations of learning, we compute the output  $\hat{O} = (\mathbf{O}_{K1})$  for an input  $\vec{X} = (\text{On}, \text{Off}, \text{Off})$  on each trained fuzzified neural network. Figure 7 shows the membership function  $\mu_{\mathbf{O}_{K1}}(x)$ . The total errors are: (a)  $E = 0.00466547$ ; (b)  $E = 0.00429045$ ; (c)  $E = 0.00235177$ . All the outputs have high membership values in  $[0, 0.2]$  and thus can be interpreted as **Off**, which is the target associated with the input  $\vec{X}$  in the training set. We see that the three adjusting schemes produce similar outputs for the same input after 10000 iterations of learning, which means that they are all adequate for learning the training set for the parity-3 problem. Their behavior in inversion is not, however, so similar as in learning.

## A.2 Inversion of Fuzzified Neural Networks

To demonstrate the inversion, we use the three fuzzified neural networks achieved through 10000 iterations of learning. On each of these fuzzified neural networks, we compute an estimate inverse of the target  $\vec{T} = (\text{On})$  through 10000 iterations of inversion. The total inputs in the input layer are represented by the same adjusting scheme as in the learning process.

The total inputs ( $\text{Net}_{01}, \text{Net}_{02}, \text{Net}_{03}$ ) in the input layer are initialized so as to satisfy

$$f(\text{Net}_{01}), f(\text{Net}_{02}), f(\text{Net}_{03}) \approx (\text{On}, \text{Off}, \text{Off}) \quad (56)$$

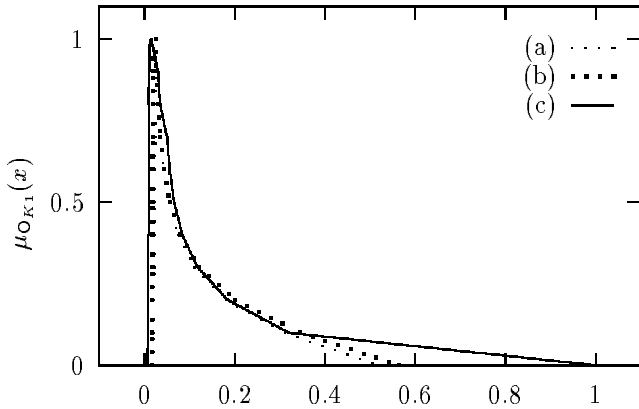


Fig. 7. The membership function  $\mu_{O_{K1}}(x)$  for the input  $\vec{X} = (\text{On}, \text{Off}, \text{Off})$  after 10000 iterations of learning

In the adjusting scheme (a),  $(\text{Net}_{01}, \text{Net}_{02}, \text{Net}_{03})$  are initialized to real numbers  $(f^{-1}(0.9), f^{-1}(0.1), f^{-1}(0.1))$ . In adjusting schemes (b) and (c),  $(\text{Net}_{01}, \text{Net}_{02}, \text{Net}_{03})$  are initialized to  $(\text{On}_{\text{inv}}, \text{Off}_{\text{inv}}, \text{Off}_{\text{inv}})$  where  $\text{On}_{\text{inv}}$  and  $\text{Off}_{\text{inv}}$  are triangular fuzzy numbers specified by  $(f^{-1}(0.8), f^{-1}(0.9), f^{-1}(0.999999))$  and  $(f^{-1}(0.000001), f^{-1}(0.1), f^{-1}(0.2))$ , respectively. Since the input  $(\text{On}, \text{Off}, \text{Off})$  is associated with the target **Off** in the training set, the inversion algorithm cannot find by chance an acceptable estimate inverse during first few iterations.

Figure 8 shows the results of inversion after 10000 iterations. The inversion times are: (a) 3.07 seconds; (b) 3.19 seconds; (c) 3.27 seconds. In adjusting schemes (a) and (b), the estimate inverses are close to  $(\text{Off}, \text{Off}, \text{Off})$  and the outputs are obviously interpreted as **On**. In contrast, the adjusting scheme (c) presents an estimate inverse which can be interpreted as  $(\text{On}, \text{On}, \text{Off})$ . The estimate inverse in the adjusting scheme (a) consists only of real numbers while those in adjusting schemes (b) and (c) consist of general fuzzy numbers; the adjusting scheme determines the type of fuzzy numbers for total inputs in the input layer as explained in Section IV-B. The total errors are: (a)  $E = 0.0115500$ ; (b)  $E = 0.0140861$ ; (c)  $E = 0.0338730$ . Note that we cannot compare the estimate inverses by these total errors because they are computed on different fuzzified neural networks.

Figure 9 shows the change of estimate inverses in the adjusting scheme (b). The total errors are: (1)  $E = 3.88880$ ; (2)  $E = 2.37550$ ; (3)  $E = 1.87615$ ; (4)  $E = 0.0142497$ . We observe that the inversion produces an output close to **On** by shifting  $X_1$  gradually toward **Off** while keeping  $X_2$  and  $X_3$  near **Off**. After 5000 iterations, we obtain an estimate inverse which can be interpreted as  $(\text{Off}, \text{Off}, \text{Off})$ .

The inversion is a sort of gradient descent search. Therefore an estimate inverse produced by inversion on a fuzzified neural network depends on the initial value of the total inputs in the input layer as well as the weight factors. Since the inversion does not affect the weight factors, we can obtain several estimate inverses by varying the initial values of the total inputs.

Figure 10 shows that four estimate inverses are obtained for the common target  $\vec{T} = (\text{On})$  through 10000 iterations of inversion in the adjusting scheme (c). The initial values of the total inputs in the input layer are: (1)  $(A, A, A)$ ; (2)  $(A, B, D)$ ; (3)  $(D, D, D)$ ; (4)  $(D, C, D)$ , where  $A, B, C,$  and  $D$  are triangular fuzzy numbers specified by  $(f^{-1}(0.1), f^{-1}(0.2), f^{-1}(0.3))$ ,  $(f^{-1}(0.3), f^{-1}(0.4), f^{-1}(0.5))$ ,  $(f^{-1}(0.5), f^{-1}(0.6), f^{-1}(0.7))$ , and  $(f^{-1}(0.7), f^{-1}(0.8), f^{-1}(0.9))$ , respectively. The total errors are: (1)  $E = 0.000473488$ ; (2)  $E = 0.000536986$ ; (3)  $E = 0.000654636$ ; (4)  $E = 0.00109205$ . The estimate inverses can be interpreted as: (1)  $(\text{Off}, \text{Off}, \text{Off})$ ; (2)  $(\text{Off}, \text{On}, \text{On})$ ; (3)  $(\text{On}, \text{On}, \text{Off})$ ; (4)  $(\text{On}, \text{Off}, \text{On})$ . These estimate inverses give all the inputs, in an approximate form, associated with the target  $\vec{T} = (\text{On})$  in the training set.

### B. Experiment 2: Effect of the Size of Training Sets on the Inversion

In this experiment, we examine the effect of the size of training sets on the inversion. We construct four fuzzified neural networks each of which has a different training set. The training sets are all based on the parity-3 problem; one is the standard training set in Table I while the others extend the standard training set. We train the four fuzzified neural networks until they produce the same average total error. Then we investigate how the size of training sets affects the inversion by comparing the total errors of estimate inverses obtained from each fuzzified neural network.

#### B.1 Training Fuzzified Neural Networks

To extend the standard training set for the parity-3 problem, we introduce seven new rules in Table II, where **Mid** is a triangular fuzzy number specified by  $(0.4, 0.5, 0.6)$ . The parity-3 problem accepts the new rules if we consider **Mid** as a fuzzy variable whose value can be either of **On** and **Off**. We test four training sets:

- (1) the standard training set
- (2) the standard training set with the rule 15
- (3) the standard training set with rules 9–14
- (4) the standard training set with rules 9–15

Note that the training set (4) includes any other training set.

Rule	$X_1$	$X_2$	$X_3$	$T_1$
9	Mid	Mid	On	Mid
10	Mid	Mid	Off	Mid
11	Mid	On	Mid	Mid
12	Mid	Off	Mid	Mid
13	On	Mid	Mid	Mid
14	Off	Mid	Mid	Mid
15	Mid	Mid	Mid	Mid

TABLE II  
NEW RULES FOR THE PARITY-3 PROBLEM

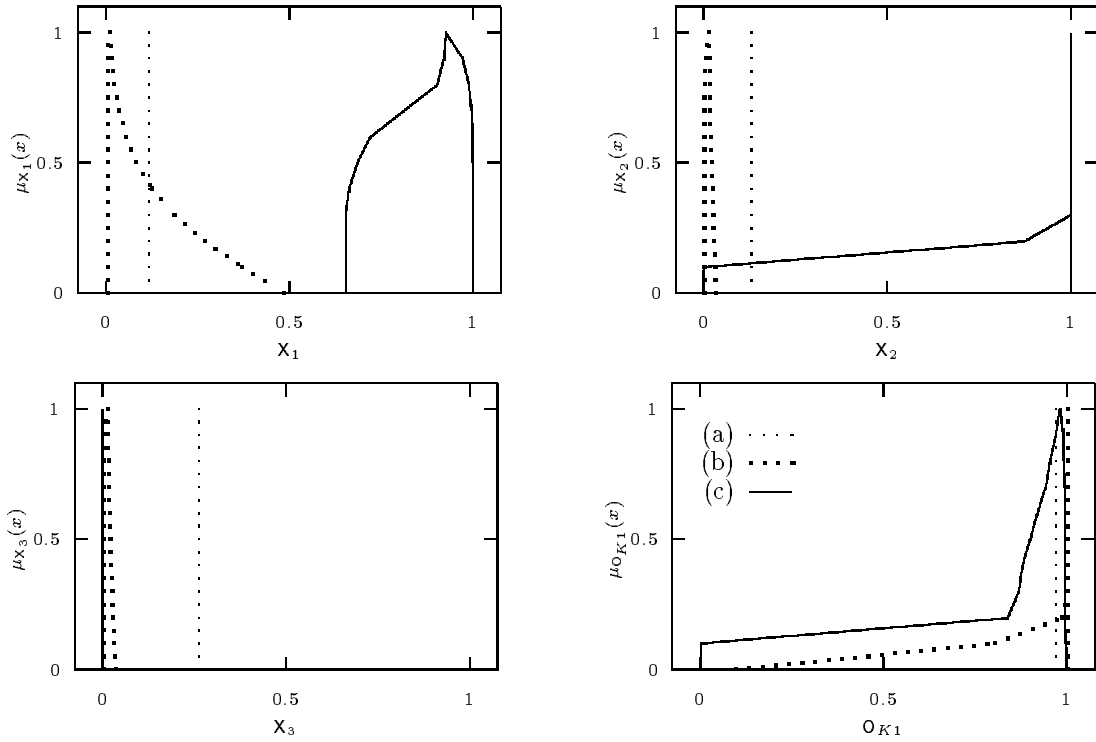


Fig. 8. The results of inversion after 10000 iterations

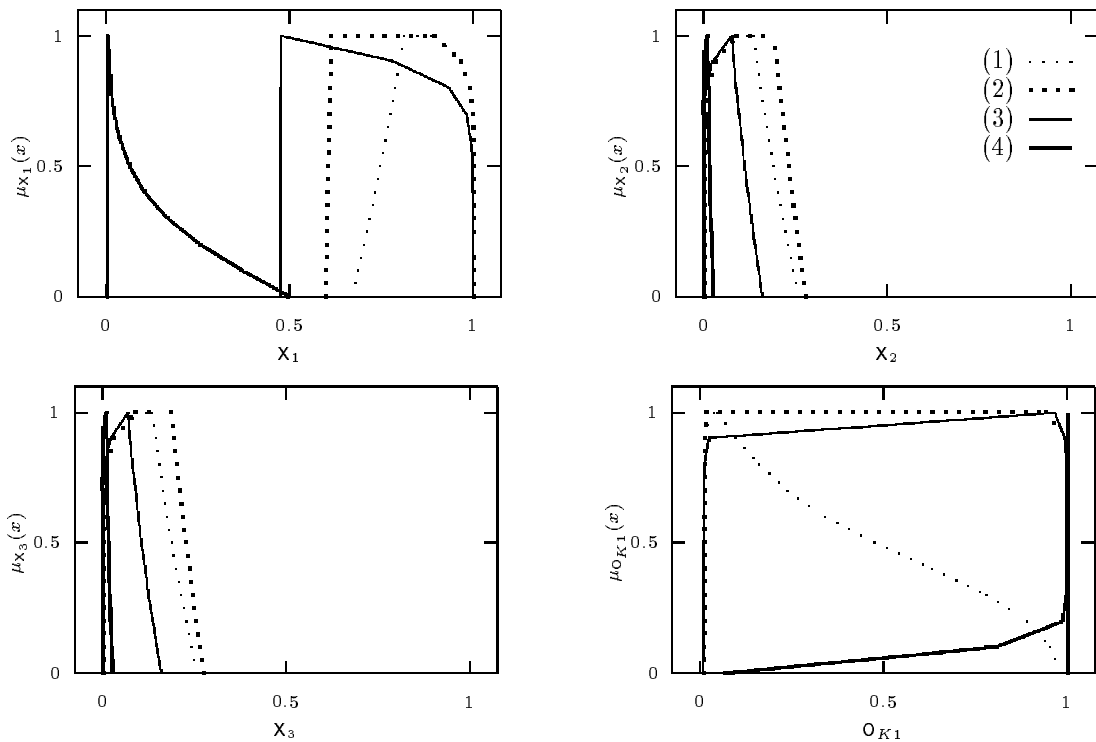


Fig. 9. Estimate inverses in the adjusting scheme (b) after: (1) 5 iterations; (2) 50 iterations; (3) 500 iterations; (4) 5000 iterations

For each training set, we construct a fuzzified neural network in Figure 1 such that  $K = 2$ ,  $n_0 = 3$ ,  $n_1 = 9$ ,  $n_2 = 1$ . The four fuzzified neural networks use the adjusting scheme (c) in Section V-A.1 and initialize weight factors to triangular fuzzy numbers specified by  $(a - 0.5, a, a + 0.5)$  where

$a$  is a random real number in  $[-1, 1]$ . We use  $\eta = 0.075$  and  $\alpha = 0.5$  in (54) for both learning and inversion.

Figure 11 shows the progress of learning for 100000 iterations on each fuzzified neural network. The vertical axis denotes the average total error of the whole training set.

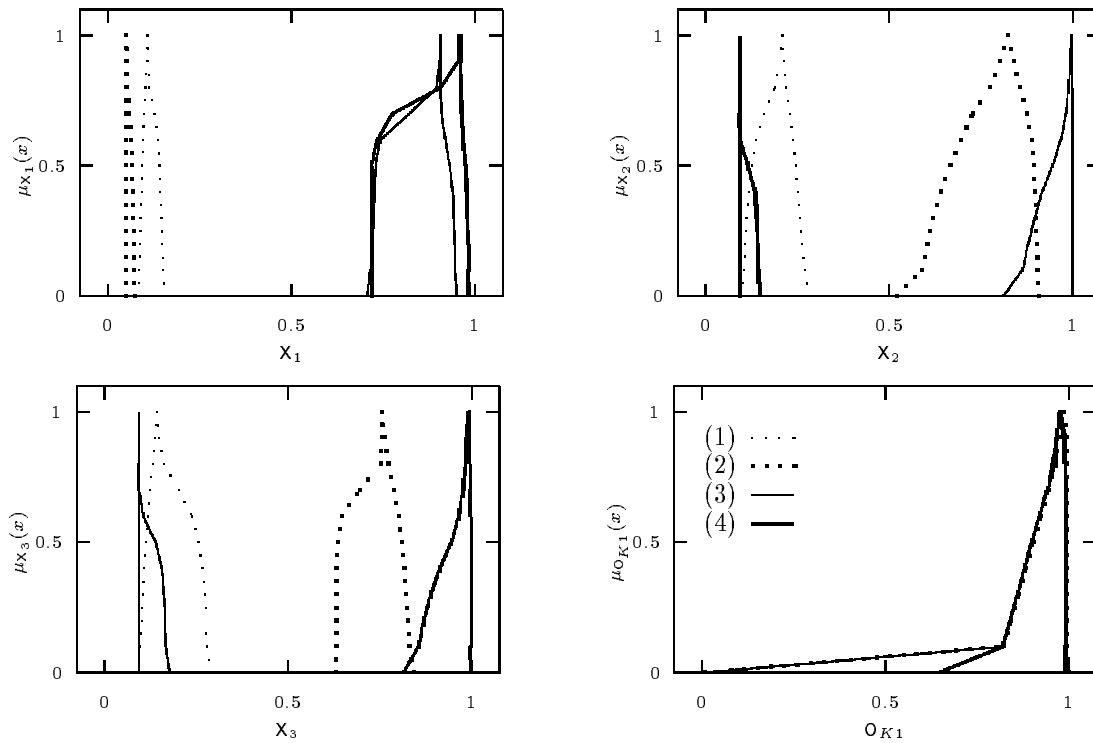


Fig. 10. Different estimate inverses for the target  $\bar{T} = (\text{On})$  in the adjusting scheme (c)

We use the average total error instead of the sum of total errors because the sizes of the training sets are all different. The learning times are proportional to the sizes of the training sets: (1) 745.43 seconds; (2) 826.1 seconds; (3) 1269.3 seconds; (4) 1352.61 seconds.

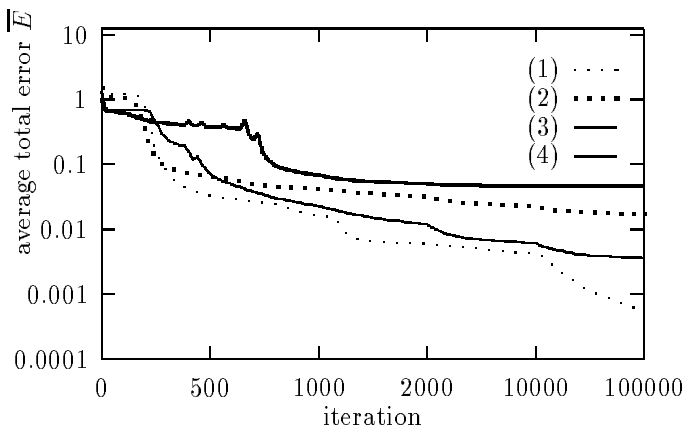


Fig. 11. The progress of learning on each fuzzified neural network

We expect that if we train a fuzzified neural network with a new training set extended with additional rules, it results in greater average total errors than the original training set after a sufficient number of iterations; the more rules a training set has, the harder to learn it is. This is exemplified by the average total errors at the 100000-th iteration in Figure 11: (1)  $\bar{E} = 0.000558843$ ; (2)  $\bar{E} = 0.0167640$ ; (3)  $\bar{E} = 0.00358692$ ; (4)  $\bar{E} = 0.0469673$ . For example,

the training set (4), which includes any other training set, produces the greatest average total error at the 100000-th iteration. The training set (2) produces a greater average total error than the training set (3), which has more rules, but we cannot compare training sets (2) and (3) by their average total errors because neither of them includes the other. Instead it implies that the rule 15 is harder than rules 9–14 for a fuzzified neural network to learn in addition to those in the standard training set.

To investigate the effect of training sets on the inversion, we compare estimate inverses in terms of accuracy by their total errors. Before comparing estimate inverses from fuzzified neural networks with different training sets, the fuzzified neural networks must learn their training sets equally accurately, not for the same number of iterations. The average total error is a measurement about how accurately a fuzzified neural network learns its training set; hence, the fuzzified neural networks for the experiment must have the same average total error.

The fuzzified neural network with the training set (4) has the greatest average total error  $\bar{E}_4 = 0.0469673$  at the 100000-th iteration. We train the other three fuzzified neural networks until their average total errors become less than or equal to  $\bar{E}_4$ : (1)  $\bar{E}_1 = 0.0469331$  at the 369-th iteration; (2)  $\bar{E}_2 = 0.0469271$  at the 747-th iteration; (3)  $\bar{E}_3 = 0.0468554$  at the 618-th iteration. Now the four fuzzified neural networks have approximately the same average total error and can be used in comparing estimate inverses.

## B.2 Comparing Estimate Inverses

A test case of inversion is specified by a target and initial values of the total inputs in the input layer. We use 432 ( $= 6 \times 6 \times 6 \times 2$ ) test cases of inversion; the target is either **On** or **Off**, and  $\text{Net}_{01}$ ,  $\text{Net}_{02}$ , and  $\text{Net}_{03}$  are initialized to A, B, C, D, E, or F. The six fuzzy numbers are all triangular fuzzy numbers specified by: A (0.000001, 0.1, 0.2); B (0.1, 0.2, 0.3); C (0.3, 0.4, 0.5); D (0.5, 0.6, 0.7); E (0.7, 0.8, 0.9); F (0.8, 0.9, 0.999999).

In each test case, we compute four estimate inverses through 10000 iterations of inversion on the above fuzzified neural networks. Table III shows the distribution of the total errors of the 432 estimate inverses from each fuzzified neural network. Each column gives the numbers of estimate inverses whose total errors are in the corresponding range. Table IV summarizes the total errors. Trimmed means are computed from those total errors less than 0.1 for each fuzzified neural network.

The mean from the training set (1) is greater than those from training sets (2), (3), and (4), all of which extend the training set (1). The means from training sets (2) and (3) are, however, less than that from the training set (4) although it extends training sets (2) and (3) with rules 9–14 and the rule 15, respectively; if we extend a training set with new rules which refine on the existing rules, the new training set can result in more accurate estimate inverses, but if the new rules are hard to learn in addition to the existing rules, the new training set may result in less accurate estimate inverses. Note that the training set (2) extends the training set (1) only with the rule 15 and yet results in the least mean of total errors. Thus, even if a training set has no contradictory rules, its size does not determine the quality of estimate inverses measured by their total errors.

The above analysis uses only the means of total errors and thus deals with the representative estimate inverses of the four fuzzified neural networks. If we obtain a poor estimate inverse with a relatively large total error, we can discard it and compute another estimate inverse using new initial values of the total inputs in the input layer. Therefore the distribution of total errors and such measurements as minimum and trimmed mean, which use the total errors of estimate inverses computed in favorable test case, are more appropriate than the mean alone to investigate the effect of training sets on the inversion in a practical perspective.

The trimmed means indicate that the training set (4) produces estimate inverses with smaller total errors on average than the training set (3), which is included by the training set (4). Note that the training set (4) gives the least minimum of total errors and that only the training set (4) produces estimate inverses whose total errors are less than 0.001; we can obtain the most accurate estimate inverses from the training set (4), not from the training set (2), if we are allowed to use a sufficient number of test cases. Therefore, although extending a training set with new rules which refine on the existing rules does not always result in more accurate estimate inverses on average, we may obtain more accurate estimate inverses from the

extended training set in best cases as long as the new rules do not contradict the existing rules.

## C. Experiment 3: Effect of the Fuzziness of Inputs and Targets on the Inversion

In this experiment, we test three new training sets for the parity-3 problem which are obtained by changing the fuzziness of the targets in Table I while keeping the inputs unmodified. We train four fuzzified neural networks for each training set. Then we compare the total errors of estimate inverses from the three training sets to investigate how the fuzziness of inputs and targets of training sets affects the inversion.

### C.1 Training Fuzzified Neural Networks

The inputs and the targets of the training set in Table I have the same fuzziness because both the inputs and the targets use fuzzy numbers **On** and **Off**. To obtain training sets (1), (2), and (3) for the parity-3 problem whose targets have different fuzziness from their inputs, we substitute fuzzy numbers **NewOn** and **NewOff** in Figure 12 for the targets **On** and **Off** in Table I, respectively, while still using **On** and **Off** for the input  $\vec{X} = (X_1, X_2, X_3)$ . The targets of the training set (1) have the greatest fuzziness, and those of the training set (3) have the least fuzziness.

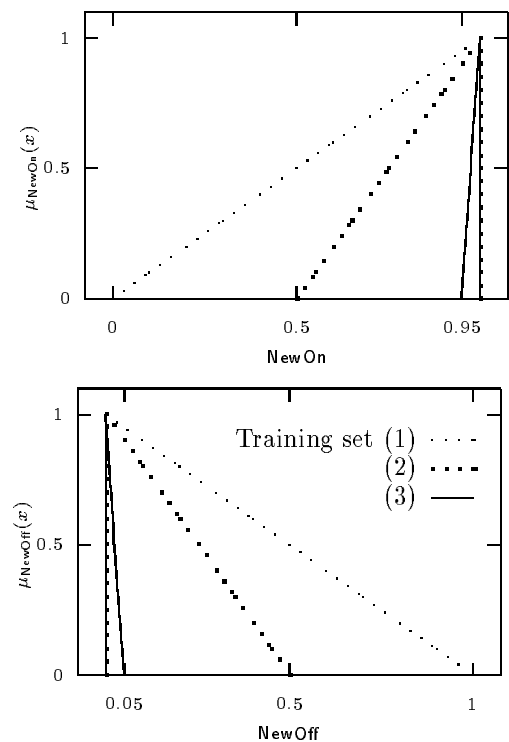


Fig. 12. The membership functions of **NewOn** and **NewOff** which are used for the targets of training sets (1), (2), and (3). The fuzziness of **NewOn** and **NewOff**: (1) > (2) > (3)

For each of training sets (1), (2), and (3), we construct four different fuzzified neural networks in Figure 1 such that  $K = 2$ ,  $n_0 = 3$ ,  $n_1 = 9$ ,  $n_2 = 1$ . All the fuzzified neural networks use the adjusting scheme (c) in Section V-A.1

Training set	$\leq 0.001$	$\leq 0.005$	$\leq 0.01$	$\leq 0.05$	$\leq 0.1$	$\leq 0.5$	$> 0.5$
(1)	0	29	188	212	0	2	1
(2)	0	22	345	65	0	0	0
(3)	0	0	168	263	0	1	0
(4)	46	53	0	329	0	4	0

TABLE III  
DISTRIBUTION OF THE TOTAL ERRORS OF ESTIMATE INVERSES

Training set	Mean	Standard deviation	Minimum	Trimmed Mean
(1)	0.0178516	0.0294641	0.00319207	0.0161523
(2)	0.0108387	0.0102893	0.00289624	0.0108387
(3)	0.0138223	0.00747597	0.00825655	0.0135630
(4)	0.0149146	0.0218801	0.000391168	0.0129182

TABLE IV  
SUMMARY OF THE TOTAL ERRORS OF ESTIMATE INVERSES

and initialize weight factors to triangular fuzzy numbers specified by  $(a - 0.5, a, a + 0.5)$  where  $a$  is a random real number in  $[-1, 1]$ . We use  $\eta = 0.075$  and  $\alpha = 0.5$  in (54) for both learning and inversion.

We train the twelve fuzzified neural networks until their average total errors become less than or equal to  $\overline{E}_4 = 0.0469673$  in Experiment 2. We use  $\overline{E}_4$  in Experiment 2 in the termination condition of learning so that we can compare the result of inversion for the training set (1) in Experiment 2 with those for the training sets in this experiment. Table V shows the number of iterations and the average total error after learning for each fuzzified neural network. From Table V, we see that as the fuzziness of the targets of the training set decreases, it tends to take more iterations of learning for a fuzzified neural network to reach a specific average total error.

	Iteration	Average total error
Training set (1) - 1	206	0.0467067
- 2	231	0.0466678
- 3	251	0.0464900
- 4	164	0.0462627
Training set (2) - 1	266	0.0469239
- 2	316	0.0467653
- 3	302	0.0468446
- 4	218	0.0467959
Training set (3) - 1	525	0.0469600
- 2	593	0.0468564
- 3	681	0.0467465
- 4	504	0.0468948

TABLE V  
THE RESULTS OF LEARNING ON THE TWELVE FUZZIFIED NEURAL NETWORKS

## C.2 Comparing Estimate Inverses

For each training set, we use 432 ( $= 6 \times 6 \times 6 \times 2$ ) test cases of inversion; the target is either of **NewOn** and **NewOff** which are the targets of the training set, and **Net<sub>01</sub>**, **Net<sub>02</sub>**, and **Net<sub>03</sub>** are initialized as in Experiment 2. We present these test cases to the four fuzzified neural networks and obtain 1728 ( $= 4 \times 432$ ) estimate inverses through 10000 iterations of inversion. Table VI shows the distribution of the total errors of the 1728 estimate inverses from each training set. Table VII summarizes the total errors.

We assume that the fuzziness of the initial values of **Net<sub>01</sub>**, **Net<sub>02</sub>**, and **Net<sub>03</sub>** does not affect the inversion because estimate inverses are obtained through a sufficient number of iterations of inversion.

Training set	Mean	Standard deviation
(1)	0.0292707	0.0148515
(2)	0.0247102	0.0161116
(3)	0.0137492	0.0313872

TABLE VII  
SUMMARY OF THE TOTAL ERRORS OF ESTIMATE INVERSES

From Table VI and Table VII, we see that the mean of total errors is approximately proportional to the fuzziness of the targets of the training set. For example, the training set (1), whose targets have the greatest fuzziness, yields the greatest mean of total errors. In contrast, the standard deviation of total errors is approximately inversely proportional to the fuzziness of the targets of the training set. For example, the training set (3), whose targets have the least fuzziness, has its total errors distributed in a relatively wide range. These observations imply that if the fuzziness of the targets of the training set decreases, we can get more accurate estimate inverses on average, but we may have to make more attempts in order to obtain an estimate inverse satisfying a given criterion because the quality of the computed estimate inverses is less stable.

Training set	$\leq 0.001$	$\leq 0.005$	$\leq 0.01$	$\leq 0.05$	$\leq 0.1$	$\leq 0.5$	$> 0.5$
(1)	0	0	0	1614	114	0	0
(2)	0	334	43	1132	219	0	0
(3)	823	419	69	291	71	55	0

TABLE VI  
DISTRIBUTION OF THE TOTAL ERRORS OF ESTIMATE INVERSES

Note that the result of inversion for the training set (1) in Experiment 2, whose inputs and targets have the same fuzziness, conforms to the above observation though it uses only 432 estimate inverses; the fuzziness of the targets, the mean of total errors (0.0178516), or the standard deviation of total errors (0.0294641) from the training set (1) in Experiment 2 are all between those from training sets (2) and (3) in this experiment. This observation implies that the mean and the standard deviation of total errors are affected more strongly by the fuzziness of the targets than by the difference of fuzziness between the inputs and the targets of the training set.

## VI. CONCLUSIONS

We described the architecture of fuzzified neural networks with fuzzy input signals and fuzzy weight factors. We presented a general framework of learning algorithms of fuzzified neural networks and derived adjusting schemes for fuzzy variables from existing learning algorithms which fit into this framework. We formulated the inversion algorithm of fuzzified neural networks which applies the adjusting scheme for fuzzy variables to the total inputs in the input layer. We compared three adjusting schemes for fuzzy variables with respect to their effect on the learning and inversion of fuzzified neural networks. Experimental results showed that the quality of estimate inverses depends on adjusting schemes. We showed that the size of training sets does not determine the quality of estimate inverses on average even if they have no contradictory rules, but that we may obtain more accurate estimate inverses in best cases if we extend a training set with new rules which refine on the existing rules. We showed that the quality of estimate inverses is more strongly affected by the fuzziness of the targets than by the difference of fuzziness between the inputs and the targets of the training set.

A crisp or fuzzified neural network can be viewed as a mathematical model for brain-like systems. The learning process increases the sum of knowledge of the neural network by improving the configuration of weight factors. In this point of view, the inversion may be compared to retrieving the knowledge accumulated in an artificial brain.

The inversion of fuzzified neural networks has several applications. For example, a fuzzy controller system implemented with fuzzified neural networks can benefit from the inversion in finding control signals for desired operations. The reliability of fuzzy systems based on fuzzified neural networks can also be examined by the inversion; we can measure indirectly how much knowledge they have or how much amount of learning they have performed. In future

works, we investigate the applicability of the inversion of fuzzified neural networks to practical problems.

## ACKNOWLEDGMENTS

The authors are grateful to anonymous referees for their helpful suggestions.

## REFERENCES

- [1] S. Horikawa, T. Furuhashi, and Y. Uchikawa, "On fuzzy modeling using fuzzy neural networks with the back-propagation algorithm," *IEEE Transactions on Neural Networks*, vol. 3, no. 5, pp. 801–806, 1992.
- [2] J.-S. R. Jang, "Anfis: adaptive-network-based fuzzy inference system," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 23, no. 3, pp. 665–685, 1993.
- [3] H. K. Kwan and Y. Cai, "A fuzzy neural network and its application to pattern recognition," *IEEE Transactions on Fuzzy Systems*, vol. 2, no. 3, pp. 185–193, 1994.
- [4] J. J. Shann and H. C. Fu, "A fuzzy neural network for rule acquiring on fuzzy control systems," *Fuzzy Sets and Systems*, vol. 71, pp. 345–357, 1995.
- [5] H. Ishibuchi, K. Morioka, and I.B. Turksen, "Learning by fuzzified neural networks," *International Journal of Approximate Reasoning*, vol. 13, no. 4, pp. 327–358, 1995.
- [6] Y. Hayashi, J.J. Buckley, and E. Czogala, "Fuzzy neural network with fuzzy signals and weights," in *Proc. of International Joint Conf. on Neural Networks*, Baltimore, 1992, vol. 2, pp. 696–701.
- [7] J.J. Buckley and Y. Hayashi, "Fuzzy neural networks: a survey," *Fuzzy Sets and Systems*, vol. 66, pp. 1–13, 1994.
- [8] J.J. Buckley and Y. Hayashi, "Direct fuzzification of neural networks," in *Proc. of 1st Asian Fuzzy Systems Symposium*, Singapore, 1993, pp. 560–567.
- [9] D.E. Rumelhart, G.E. Hinton, and R.J. Williams, *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, vol. 1: Foundations, MIT Press, Cambridge, MA, 1986.
- [10] T. Feuring, "Learning in fuzzy neural networks," in *Proc. of IEEE International Conf. on Neural Networks*, Washington, 1996, pp. 1061–1066.
- [11] H. Ishibuchi, R. Fujioka, and H. Tanaka, "An architecture of neural networks for input vectors of fuzzy numbers," in *Proc. of IEEE International Conf. on Fuzzy Systems*, San Diego, 1992, pp. 1293–1300.
- [12] H. Ishibuchi, R. Fujioka, and H. Tanaka, "Neural networks that learn from fuzzy if-then rules," *IEEE Transactions on Fuzzy Systems*, vol. 1, no. 2, pp. 85–97, 1993.
- [13] H. Ishibuchi, K. Kwon, and H. Tanaka, "Learning of fuzzy neural networks from fuzzy inputs and fuzzy targets," in *Proc. of 5th IFSA World Congress*, Seoul, Korea, 1993, pp. 147–150.
- [14] H. Ishibuchi, K. Kwon, and H. Tanaka, "A learning algorithm of fuzzy neural networks with triangular fuzzy weights," *Fuzzy Sets and Systems*, vol. 71, pp. 277–293, 1995.
- [15] H. Ishibuchi and M. Nii, "Learning of fuzzy connection weights in fuzzified neural networks," in *Proc. of IEEE International Conf. on Fuzzy Systems*, Washington, 1996, pp. 373–379.
- [16] J. Duniak and D. Wunsch, "A training technique for fuzzy number neural networks," in *Proc. of IEEE International Conf. on Neural Networks*, 1997, pp. 533–536.
- [17] A. Linden and J. Kindermann, "Inversion of multilayer nets," in *Proc. of International Joint Conf. on Neural Networks*, 1989, vol. 2, pp. 425–430.
- [18] D.A. Hoskins, J.N. Hwang, and J. Vagners, "Iterative inversion of neural networks and its application to adaptive control,"

*IEEE Transactions on Neural Networks*, vol. 3, pp. 292–301, 1992.

- [19] D.T. Davis, Z. Chen, L. Tsang, J.N. Hwang, and A. Chang, “Retrieval of snow parameters by iterative inversion of a neural network,” *IEEE Transactions on Geoscience and Remote Sensing*, vol. 31, pp. 842–852, 1993.
- [20] J.N. Hwang and C.H. Chan, “Iterative constrained inversion and its applications,” in *Proc. of 24th Conf. on Information Systems and Sciences*, Princeton, 1990, pp. 754–759.
- [21] J.N. Hwang, J.J. Choi, S. Oh, and R. J. Marks II, “Query-based learning applied to partially trained multilayer perceptrons,” *IEEE Transactions on Neural Networks*, vol. 2, no. 1, pp. 131–136, 1992.
- [22] A. Kaufmann and M.M. Gupta, *Introduction to Fuzzy Arithmetic: Theory and Applications*, Van Nostrand Reinhold, New York, 1991.
- [23] A.M. Anile, S. Deodato, and G. Privitera, “Implementing fuzzy arithmetic,” Dipartimento Di Matematica, Università Degli Studi Di Catania, Italy, 1994.
- [24] P.V. Krishnamraju, J.J. Buckley, K.D. Reilly, and Y. Hayashi, “Genetic learning algorithms for fuzzy neural nets,” in *Proc. of FUZZ-IEEE '94*, Orlando, 1994, pp. 1969–1974.

## APPENDIX

### CALCULATION OF PARTIAL DERIVATIVES

Consider  $W_{kij}$ , the weight factor of the neuron  $i$  in the layer  $k$  for the neuron  $j$  in the layer  $k - 1$ ,  $k = 1, \dots, K$ ,  $i = 1, \dots, n_k$ ,  $j = 0, \dots, n_{k-1}$ . With  $j = 0$ ,  $W_{kij}$  becomes  $\Theta_{ki}$ , the bias for the neuron  $i$  in the layer  $k$ . For  $W_{kij}$ , we have the following relation for the level  $h = h_1, \dots, h_v$  from the chain rule:

$$\frac{\partial E}{\partial W_{kij}^h} = \frac{\partial E}{\partial \text{Net}_{ki}^h} \frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h} + \frac{\partial E}{\partial \text{Net}_{ki}^h} \frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h} \quad (57)$$

$$\frac{\partial E}{\partial W_{kij}^h} = \frac{\partial E}{\partial \text{Net}_{ki}^h} \frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h} + \frac{\partial E}{\partial \text{Net}_{ki}^h} \frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h} \quad (58)$$

Since  $\mathcal{O}_{(k-1)j}^h$  satisfies

$$0 \leq \mathcal{O}_{(k-1)j}^h \leq \mathcal{O}_{(k-1)j}^h \quad (59)$$

we obtain  $\text{Net}_{ki}^h$  as

$$\text{Net}_{ki}^h = \sum_{W_{kij}^h < 0} W_{kij}^h \mathcal{O}_{(k-1)j}^h + \sum_{W_{kij}^h \geq 0} W_{kij}^h \mathcal{O}_{(k-1)j}^h \quad (60)$$

$$\text{Net}_{ki}^h = \sum_{W_{kij}^h < 0} W_{kij}^h \mathcal{O}_{(k-1)j}^h + \sum_{W_{kij}^h \geq 0} W_{kij}^h \mathcal{O}_{(k-1)j}^h \quad (61)$$

where  $j$  ranges from 0 to  $n_{k-1}$ . From (60)–(61), we have

$$\frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h} = \frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h} = 0 \quad (62)$$

Then (57)–(58) are rewritten as

$$\frac{\partial E}{\partial W_{kij}^h} = \frac{\partial E}{\partial \text{Net}_{ki}^h} \frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h} \quad (63)$$

$$\frac{\partial E}{\partial W_{kij}^h} = \frac{\partial E}{\partial \text{Net}_{ki}^h} \frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h} \quad (64)$$

For the neuron  $i$  in the layer  $k$ , we define two new terms  $\delta_{ki}^h$  and  $\delta_{ki}^h$ , which are called error signal, as

$$\delta_{ki}^h = -\frac{\partial E}{\partial \text{Net}_{ki}^h} \quad (65)$$

$$\delta_{ki}^h = -\frac{\partial E}{\partial \text{Net}_{ki}^h} \quad (66)$$

Using (65)–(66), we obtain

$$\frac{\partial E}{\partial W_{kij}^h} = -\delta_{ki}^h \frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h} \quad (67)$$

$$\frac{\partial E}{\partial W_{kij}^h} = -\delta_{ki}^h \frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h} \quad (68)$$

The calculation of  $\frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h}$  and  $\frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h}$  depends on  $W_{kij}^h$ .

We have three situations of  $W_{kij}^h$ :

(1)  $0 \leq W_{kij}^h \leq W_{kij}^h$

$$\frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h} = \mathcal{O}_{(k-1)j}^h, \quad \frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h} = \mathcal{O}_{(k-1)j}^h \quad (69)$$

(2)  $W_{kij}^h \leq W_{kij}^h < 0$

$$\frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h} = \mathcal{O}_{(k-1)j}^h, \quad \frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h} = \mathcal{O}_{(k-1)j}^h \quad (70)$$

(3)  $W_{kij}^h < 0 \leq W_{kij}^h$

$$\frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h} = \mathcal{O}_{(k-1)j}^h, \quad \frac{\partial \text{Net}_{ki}^h}{\partial W_{kij}^h} = \mathcal{O}_{(k-1)j}^h \quad (71)$$

The calculation of partial derivatives  $\delta_{ki}^h$  and  $\delta_{ki}^h$  depends on  $k$ . We consider two cases of  $k$ , one for the output layer and the other for hidden layers.

A. Output Layer  $K$ ,  $k = K$

Each component of the target  $\vec{T} = (T_1, T_2, \dots, T_{n_K})$  is treated as a constant fuzzy number. To compute  $\delta_{Ki}^h$ ,  $i = 1, 2, \dots, n_K$ , we apply the chain rule to (65) as

$$\begin{aligned} \delta_{Ki}^h &= -\frac{\partial E}{\partial \text{Net}_{Ki}^h} \\ &= -\frac{\partial E}{\partial \mathcal{O}_{Ki}^h} \frac{\partial \mathcal{O}_{Ki}^h}{\partial \text{Net}_{Ki}^h} \\ &= -\frac{\partial \sum_{u=1}^{n_K} \sum_{g=h_1, \dots, h_v} e_g(\mathcal{O}_{Ku}, T_u) \partial f(\text{Net}_{Ki}^h)}{\partial \mathcal{O}_{Ki}^h} \frac{\partial f(\text{Net}_{Ki}^h)}{\partial \text{Net}_{Ki}^h} \\ &= -\frac{\partial e_h(\mathcal{O}_{Ki}, T_i)}{\partial \mathcal{O}_{Ki}^h} \mathcal{O}_{Ki}^h (1 - \mathcal{O}_{Ki}^h) \end{aligned} \quad (72)$$

where  $f$  is the standard sigmoidal function. The relation  $f'(x) = f(x)(1 - f(x))$  is used in computing  $\frac{\partial f(\text{Net}_{Ki^h_L})}{\partial \text{Net}_{Ki^h_L}}$ . In a similar way, we write

$$\delta_{Ki^h_R} = -\frac{\partial e_h(\mathbf{O}_{Ki}, \mathbf{T}_i)}{\partial \mathbf{O}_{Ki^h_R}} \mathbf{O}_{Ki^h_R} (1 - \mathbf{O}_{Ki^h_R}) \quad (73)$$

The calculation of  $\delta_{Ki^h_L}$  and  $\delta_{Ki^h_R}$ , thus, depends on the cost function  $e_h$ . For example, the cost function  $e_h$  in (23) gives

$$\delta_{Ki^h_L} = (\mathbf{T}_i^h - \mathbf{O}_{Ki^h_L}) \mathbf{O}_{Ki^h_L} (1 - \mathbf{O}_{Ki^h_L}) \quad (74)$$

$$\delta_{Ki^h_R} = (\mathbf{T}_i^h - \mathbf{O}_{Ki^h_R}) \mathbf{O}_{Ki^h_R} (1 - \mathbf{O}_{Ki^h_R}) \quad (75)$$

The cost function  $e_h$  in (24) gives,

$$\delta_{Ki^h_L} = h(\mathbf{T}_i^h - \mathbf{O}_{Ki^h_L}) \mathbf{O}_{Ki^h_L} (1 - \mathbf{O}_{Ki^h_L}) \quad (76)$$

$$\delta_{Ki^h_R} = h(\mathbf{T}_i^h - \mathbf{O}_{Ki^h_R}) \mathbf{O}_{Ki^h_R} (1 - \mathbf{O}_{Ki^h_R}) \quad (77)$$

### B. Hidden Layer $k$ , $k = 1, \dots, K-1$

$\delta_{ki^h_L}$  and  $\delta_{ki^h_R}$ ,  $i = 1, \dots, n_k$ , are obtained after the layer  $k+1$  is processed by calculating  $\delta_{(k+1)l^h_L}$  and  $\delta_{(k+1)l^h_R}$ ,  $l = 1, \dots, n_{k+1}$ . Specifically we apply the chain rule to (65) as

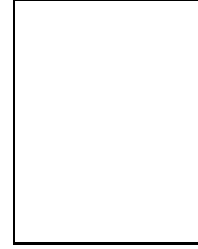
$$\begin{aligned} \delta_{ki^h_L} &= -\frac{\partial E}{\partial \text{Net}_{ki^h_L}} \\ &= -\frac{\partial E}{\partial \mathbf{O}_{ki^h_L}} \frac{\partial \mathbf{O}_{ki^h_L}}{\partial \text{Net}_{ki^h_L}} \\ &= -\sum_{l=1}^{n_{k+1}} \left( \frac{\partial E}{\partial \text{Net}_{(k+1)l^h_L}} \frac{\partial \text{Net}_{(k+1)l^h_L}}{\partial \mathbf{O}_{ki^h_L}} \right. \\ &\quad \left. + \frac{\partial E}{\partial \text{Net}_{(k+1)l^h_R}} \frac{\partial \text{Net}_{(k+1)l^h_R}}{\partial \mathbf{O}_{ki^h_L}} \right) \frac{\partial \mathbf{O}_{ki^h_L}}{\partial \text{Net}_{ki^h_L}} \\ &= -\left( \sum_{\substack{l=1 \\ \mathbf{W}_{(k+1)l^h_L} \geq 0}}^{n_{k+1}} \frac{\partial E}{\partial \text{Net}_{(k+1)l^h_L}} \frac{\partial \text{Net}_{(k+1)l^h_L}}{\partial \mathbf{O}_{ki^h_L}} \right. \\ &\quad \left. + \sum_{\substack{l=1 \\ \mathbf{W}_{(k+1)l^h_R} < 0}}^{n_{k+1}} \frac{\partial E}{\partial \text{Net}_{(k+1)l^h_R}} \frac{\partial \text{Net}_{(k+1)l^h_R}}{\partial \mathbf{O}_{ki^h_L}} \right) \frac{\partial \mathbf{O}_{ki^h_L}}{\partial \text{Net}_{ki^h_L}} \\ &= \left( \sum_{\substack{l=1 \\ \mathbf{W}_{(k+1)l^h_L} \geq 0}}^{n_{k+1}} \delta_{(k+1)l^h_L} \mathbf{W}_{(k+1)li^h_L} \right. \\ &\quad \left. + \sum_{\substack{l=1 \\ \mathbf{W}_{(k+1)l^h_R} < 0}}^{n_{k+1}} \delta_{(k+1)l^h_R} \mathbf{W}_{(k+1)li^h_R} \right) \mathbf{O}_{ki^h_L} (1 - \mathbf{O}_{ki^h_L}) \end{aligned} \quad (78)$$

$$\left( \sum_{\substack{l=1 \\ \mathbf{W}_{(k+1)l^h_L} \geq 0}}^{n_{k+1}} \delta_{(k+1)l^h_L} \mathbf{W}_{(k+1)li^h_L} \right. \\ \left. + \sum_{\substack{l=1 \\ \mathbf{W}_{(k+1)l^h_R} < 0}}^{n_{k+1}} \delta_{(k+1)l^h_R} \mathbf{W}_{(k+1)li^h_R} \right) \mathbf{O}_{ki^h_L} (1 - \mathbf{O}_{ki^h_L}) \quad (78)$$

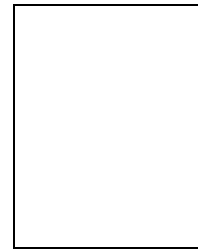
Similarly we obtain

$$\left( \sum_{\substack{l=1 \\ \mathbf{W}_{(k+1)l^h_L} < 0}}^{n_{k+1}} \delta_{(k+1)l^h_L} \mathbf{W}_{(k+1)li^h_L} \right. \\ \left. + \sum_{\substack{l=1 \\ \mathbf{W}_{(k+1)l^h_R} \geq 0}}^{n_{k+1}} \delta_{(k+1)l^h_R} \mathbf{W}_{(k+1)li^h_R} \right) \mathbf{O}_{ki^h_R} (1 - \mathbf{O}_{ki^h_R}) \quad (79)$$

In this manner, error terms  $\delta_{(k+1)l^h_L}$  and  $\delta_{(k+1)l^h_R}$  are backpropagated to the layer  $k$ .



**Sungwoo Park** received his B.S and M.S degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST), Korea, in 1996 and in 1998, respectively. From 1998 to 1999, he was with Electronics and Telecommunications Research Institute (ETRI), Korea. Currently he is a Ph.D. student at Computer Science Department, Carnegie Mellon University. He is interested in domain specific languages for robotics and machine learning.



**Taisook Han** received the B.S. degree in electronic engineering from Seoul National University, Korea in 1976, the M.S. degree in computer science from Korea Advanced Institute of Science and Technology (KAIST), Korea in 1978, and the Ph.D. degree in computer science from the University of North Carolina at Chapel Hill, in 1990. He is currently an Associate Professor at the Department of Computer Science, KAIST, Korea. His research interests are in the area of parallel processing, functional languages, and programming languages.