A Simple Fuzzy Neural Network

Carl G. Looney and Sergiu Dascalu
Computer Science & Engineering/171
University of Nevada, Reno
Reno, NV 89557
<looney,dascalus>@cse.unr.edu

Abstract

Our simple fuzzy neural network first thins the set of exemplar input feature vectors and then centers a Gaussian function on each remaining one and saves its associated output label (target). Next, any unknown feature vector to be classified is put through each Gaussian to get the fuzzy truth that it belongs to that center. The fuzzy truths for all Gaussian centers are then maximized and the label of the winner is the class of the input feature vector. We use the knowledge in the exemplar-label pairs directly with no training, no weights, no local minima, no epochs, no defuzzification, no overtraining, and no experience needed to use it. It sets up automatically and then classifies all input feature vectors from the same population as the exemplar feature vectors. We compare our results on well known data with those of several other fuzzy neural networks, which themselves compared favorably to other neural networks.

1. INTRODUCTION

In this paper we discuss: 1) *what* the simple fuzzy neural network is; 2) *why* a simple but accurate neural network is needed; 3) *how* this one works; 4) *when* to use this type of neural network; and 5) its performance on well known data. NNs are regression machines that associate inputs with outputs, which is why they are so useful. They may represent particular transformations for which no models are known.

A neural network (NN) is a black box to which we input N values $x_1,...,x_N$ that form a *feature vector* $\boldsymbol{x}$ to obtain an output vector $\boldsymbol{z}$ that designates the class, identification, group, pattern, or associated output codeword of the input vector $\boldsymbol{x}$. A trained NN represents a system that maps a set of exemplar input feature vectors $\{\boldsymbol{x}^{(q)}: q = 1,...,Q\}$ to a set of output *target vectors* $\{\boldsymbol{t}^{(q)}: q = 1,...,Q\}$, also called *labels*, so that each $\boldsymbol{x}^{(q)}$ maps more closely to $\boldsymbol{t}^{(q)}$ than to another target. This allows the network to make interpolations and extrapolations that map any input $\boldsymbol{x}$ to $\boldsymbol{z}$ that best matches label $\boldsymbol{t}^{(q)}$ for the correct index q.

When trained, a NN is a computational machine that implements an algorithm that is specified by the input nodes, output nodes, layers of hidden nodes between them, connecting lines, functions at the transforming nodes, and weight multipliers $w_1,...,w_R$ on these lines. We denote the weights here as a vector $\mathbf{w}$. The NN is a composition of many functions designated by the overall NN function as

$$z = f(\boldsymbol{x};\boldsymbol{w}) \tag{1}$$

Here, $\boldsymbol{x}$ is any input feature vector and $\boldsymbol{z}$ is the resulting output vector that should match best a label $\boldsymbol{t}^{(q)}$ for some q.

Figure 1 displays a traditional *backpropagation* type of NN (e.g., see [6] for details). The vertical bar to the right of the input nodes binds the N lines together into a bus to each of M nodes in the hidden (middle) layer. There is a weight on each line to the M hidden nodes, and also on each line to the J output nodes. These weights are adjusted to train the NN to match exemplar inputs $\{\boldsymbol{x}^{(q)}\}$ with exemplar targets $\{\boldsymbol{t}^{(q)}\}$. At the hidden nodes the weighted vector components are summed and put through a sigmoid (logistic, or "S" shaped) soft threshold function to go to the output nodes to form the outputs $\{\boldsymbol{z}^{(q)}\}$ [6]. Each output node computes a linear combination of (weighted) line values for $\mathbf{z}$.

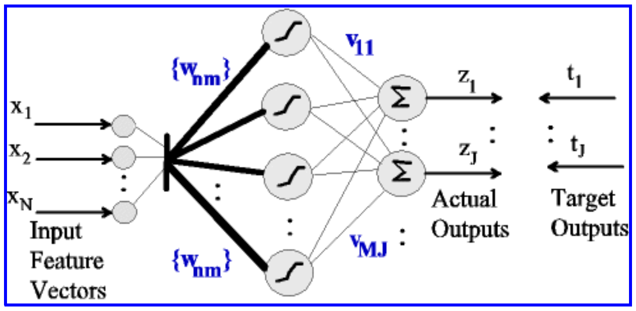

Figure 1. A backpropagation neural network.

As the weights $\boldsymbol{w}$ vary, the NN function varies, and so a particular form of neural network is an infinite family of neural networks. For a particular application we must train $\boldsymbol{w}$ so that the NN outputs a close approximation $\boldsymbol{z}$ to the correct target $\boldsymbol{t}^{(q)}$ for any input $\boldsymbol{x}^{(q)}$. This is done by training on a set $\{(\boldsymbol{x}^{(q)}, \boldsymbol{t}^{(q)}) : q = 1,...,Q\}$ of exemplar vector pairs of inputs and targets where the input vectors represent sufficiently well the portion of the input space from which the input vectors to be classified will be drawn. The trained NN then can interpolate on an input vector **x** and match its output **z** to the nearest target $\boldsymbol{t}^{(q)}$ to provide the class of **x**.

The original *backpropagation* NNs (BPNNs), as well as *radial basis function* NNs (RBFNNs) (e.g., see [6, 7]) are trained by steepest descent on the weights that minimize the output sum-squared error (SSE) E, where

$$E = \sum_{q=1,Q} ||\boldsymbol{z}^{(q)} - \boldsymbol{t}^{(q)}||^2 \tag{1}$$

Here $\boldsymbol{z}^{(q)}$ is the computed output for the input vector $\boldsymbol{x}^{(q)}$, and $\boldsymbol{t}^{(q)}$ is the target output (label) to which $\boldsymbol{x}^{(q)}$ is supposed to map. Each $\boldsymbol{z}^{(q)}$ is a differentiable function of the weights, so training is done on each single weight w_{nm} by taking steps along the direction of steepest descent of the SSE E via

$$w_{nm}^{(i+1)} = w_{nm}^{(i)} + \alpha(\partial E/\partial w_{nm}) \tag{2}$$

where α is the step size parameter, also called the *learning rate*, and *i* is the iteration number. The starting values of the w_{nm} are drawn randomly, usually between -0.5 and 0.5 for a cautious start.

Training usually requires thousands of *epochs*, of which each is a set of steps to adjust each weight in $\{w_{nm}\}$ once (or sometimes more than once). However, the learning of one weight tends to unlearn the other weights, so epochs are continued until the SSE is sufficiently small. Another problem of BPNNs is that the learned set of weights yields a local minimum, of which it has been shown that there are many [6] so that the learning is very likely to not be optimal. RBFNNs have only a single global minimum and are thus preferable. But for most trained NNs there is also the problem of overtraining, by which reducing the SSE to a very small value causes the noise on the input exemplars to be learned. This reduces the accuracy when other feature vectors are put through the NN that have different noise values.

A *fuzzy NN* (FNN) may have *4* layers [2, 3, 4] as follows: 1) the first is the input layer as in a BPNN or RBFNN that simply fans out the inputs to the next layer; 2) a hidden layer that fuzzifies the inputs, e.g., into LOW, MEDIUM, and HIGH linguistic variables as rule *antecedents* with a fuzzy truth for each, obtained by passing each input value through a *fuzzy set membership function* (FSMF) for a linguistic variable (see [5]); 3) a rule layer where arrows from certain fuzzifying nodes imply a *consequent* fuzzy variable in this layer; and 4) the defuzzification layer (also see [4]).

Many FNNs currently use the more general *5* layers for fuzzy rules where the third is the AND (min.) layer and the fourth is the OR (max.) layer [9]. Figure 2 shows the general min-max FNN that allows the most general rule-based representation. Consider the rule: *(A_1 is LOW) AND (A_2 is HIGH) => (C is LOW)*. The antecedents A_1 and A_2 propagate the minimum of their fuzzy truths to the consequent *C*. If *(A_3 is MEDIUM)AND (A_4 is HIGH) => (C is LOW)* also, then *C* takes the maximum of the two fuzzy implications as its fuzzy truth, which is fuzzy ORing of the two rules that imply *(C is LOW)*.

Thus such a FNN is a min-max (AND-OR) fuzzy rule-based system conceptuatlized in network format. There are many variations and applications of FNNs (e.g., see [2, 3, 4, 5]). Rules can be built into the architecture by experts or they can be learned by training to produce known correct outputs.

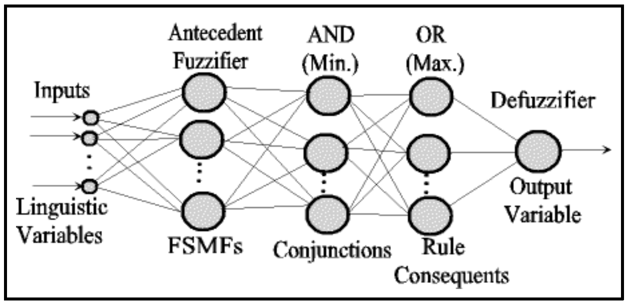

Figure 2. The general min-max FNN.

NNs are needed for repetitive tasks and for input-output associations for which there are no mathematical models available for the system that is to be represented. The NNs interpolate and extrapolate from known input-output pairs, and provide fast on-line computation of outputs that either could not be directly computed or would be slow to compute. They can obviously be used only in situations where there is good representative labeled data available for training the NN. All training should be tested on known (labeled) feature vectors that were not used in the training (e.g., see [6]) so as to validate the training.

2. A SIMPLE FUZZY NEURAL NETWORK

From the above discussion we see the need for a NN that avoids training. The general architecture for our *simple fuzzy neural network* (SFNN) that satisfies this need is shown in Figure 3. However, for the purpose of explanation of how it works, we use the simplified case of only two classes as shown in Figure 4. Each class grouping of the SFNN works the same way. More classes yield fuzzy memberships in more classes from which to select a maximum value winner at the final output node.

In Figure 4 we have N features in the input exemplar feature vectors. Here there are two classes in the training exemplar data *$\{(\boldsymbol{x}^{(q)}, \boldsymbol{t}^{(q)}): q = 1,...,Q\}$*, i.e., the $\boldsymbol{t}^{(q)}$ have two unique labels, so we use $K = 2$ class groups of hidden nodes where each such node represents a Gaussian function centered on an exemplar feature vector that has an associated label. Each Gaussian in a class group has a different center but the same label. Consider the first group of hidden nodes for Class 1 in Figure 4.

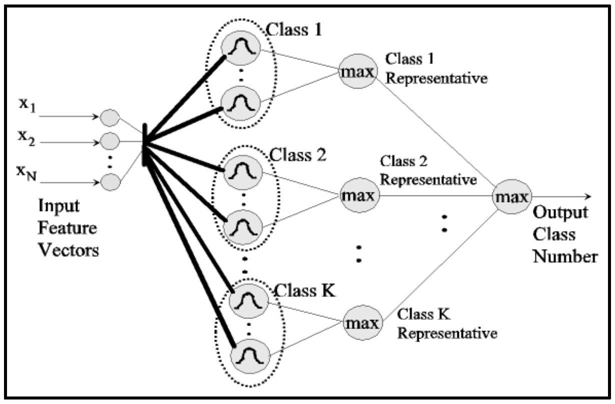

Figure 3. The simple fuzzy neural network.

In the usual case there may be a large number K_p of feature vectors in Class p ($p = 1,2$ here), so we eliminate those feature vectors that are close to another feature vector with the same label. This reduces the number of centers, and thus Gaussians (nodes), that represent each Class p. The fuzzy truth that input vector $\boldsymbol{x}$ is in the same class as $\boldsymbol{x}^{(q)}$ is given by the Gaussian FSMF centered on $\boldsymbol{x}^{(q)}$. The q^{th} Gaussian FSMF is the function

$$\boldsymbol{x} \rightarrow g(\boldsymbol{x};\boldsymbol{x}^{(q)}) = exp\{-||\boldsymbol{x} - \boldsymbol{x}^{(q)}||^2/(2\sigma^2)\} \quad (3)$$

where σ can be taken to be one-half of the average distance between all exemplar pairs.

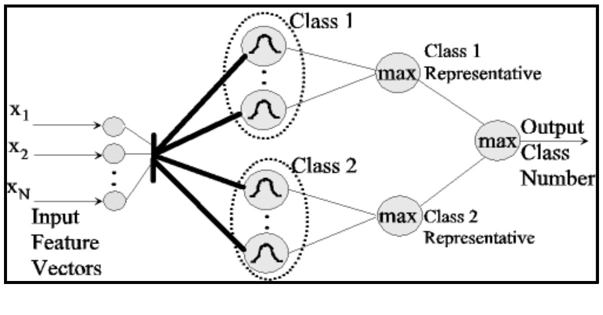

Figure 4. A SFNN for only two classes.

All of the fuzzy truths for the centers of Gaussians in Class *1* (see Fig. 4) are now fed from their Gaussian nodes to the maximizer node of the Class *1* fuzzy truths, which acts as a fuzzy OR node in selecting the representative center and fuzzy truth that $\boldsymbol{x}$ belongs to some $\boldsymbol{x}^{(k)}$ for Class 1. This maximum fuzzy truth for $\boldsymbol{x}$ to be in Class *1* is now sent to the final output maximizer node as the Class *1* representative. The final output maximizer node also receives the Class *2* representative (maximum fuzzy truth) that **x** belongs to Class *2* and determines the maximum of these fuzzy truths, so the class that sent it is the winner. Thus the input $\boldsymbol{x}$ belongs to the winning class determined by the label of the winning Gaussian center vector.

3. WHY THE SFNN IS NEEDED

For this fuzzy neural network there is no adjustment of parameters (no steepest descent), no SSEs, no epochs, no explicit rules, no overtraining, and no local or other minima to find. There are also no fuzzy rules to learn by training. Thus the SFNN can be used by researchers in other fields who have no experience or intuition about training NNs. The algorithm specifies the computations with no parameters or thresholds to be estimated.

We start with a set of labeled data $\{(\boldsymbol{x}^{(q)}, \boldsymbol{t}^{(q)}): q = 1,...,Q\}$ and eliminate all exemplar input vectors that are too close to another exemplar vector with the same label. Of the remaining thinned vector set, we center a Gaussian on each vector and keep a list of indices of the labels for the vectors that remain as Gaussian centers. The knowledge is in the labels, so we use it directly instead of creating extraneous mappings to try to reach the equivalent knowledge.

If there are not too many labeled exemplars (say, 100 or less for each class), and with reasonable input dimension N, then we do not need to thin them due to the speed of today's computers. We can just use each exemplar as a Gaussian center that is a FSMF. Our computer program allows us to thin or not thin the labeled exemplar feature vectors to become centers. Obviously the algorithm is faster for fewer Gaussians, which is the motivator for thinning, but it can be (but is not necessarily) more accurate for more Gaussians, just as more rules provides more accuracy in other FNNs [4].

4. THE SFNN ALGORITHM

The high level algorithm is straight forward as given in the steps listed below, and is easy to program. The training and learning reside in the exemplar feature vector and their labels so we don't need to expend computation time to train. We sometimes use some computation to thin the exemplars that are very similar and belong to the same class. The algorithm is given here in a form for human understanding, but can be readily adapted to program code.

Step 0: Read in the data file (the number of features N, the number of feature vectors Q, the dimension J of the labels, the number K of classes, all Q feature vectors and all Q labels).

```
Step 1: Find minimal distance Dmin over all feature vector
        pairs
    Put $\sigma = Dmin/2$        //Use σ in Eqn. (3)
    Put $G = Q$                  //Starting no. Gaussian centers

Step 2: Find two exemplar vectors of min. distance $d$ with
        indices $k1$ and $k2$
    If $d < (½)Dmin$             //If vectors are close and
      If $label[k1] = label[k2]$ // have same label
        Eliminate Gaussian center $k2$
        $G = G - 1$              //Reduce no. Gaussians
      Goto Step 2

Step 3: Input next unknown $\mathbf{x}$ to SFNN to be classified
    For $k = 1$ to $G$ do        //For each Gaussian center
      Compute $g[k] = exp\{-||\mathbf{x} - \mathbf{x}^{(k)}||^2/(2\sigma^2)\}$
    Find maximum $g[k^*]$, over $k = 1,...,G$
    Output $\mathbf{x}$, $label[k^*]$ //$label[k^*]$ is class of $\mathbf{x}$

Step 4: If all inputs for classifying are done, stop
    Else, goto Step 3
```

5. TESTING ON THE *IRIS* DATASET

The *iris* dataset of Anderson [1], 1935, remains a rigorous test of any classification system. There are $Q = 150$ feature vectors consisting of $N = 4$ features for $K = 3$ classes of *iris* flowers. The classes are *sestosa*, *virginicus*, and *versicolor*. The features are *sepal length* and *width*, and *petal length* and *width*. The second feature tends to be in a narrow range with overlapping values between the classes, which prevents clustering from yielding the 3 classes of 50 vectors each. Clustering algorithms, including the fuzzy c-means, do not agree with Anderson's labels completely.

For neural networks, the training is *supervised learning* instead of *self-organizing* as done by clustering methods, and the data is learned accurately, so the data needs to be quite good. Overlapping feature values between the classes (as in the *iris* data) prevent learning that accurately discriminates all feature vectors. But in general, while neural network algorithms learn what they are trained to know, they learn any errors in the data as facts, as do other supervised learning machines, including support vector machines [4].

We obtained the *iris* data set from the repository at University of California, Irvine [8] whose Internet URL is: www.ics.uci.edu/~mlearn/MLRepository.html . The data set from the repository was arranged as follows: 50 feature vectors in Class *1*, followed by 50 feature vectors in Class *2,* and 50 feature vectors in Class *3* (each feature vector also contained the class number as the last field).

We rearranged our feature vectors so that: the first was from Class *1*, the second was from Class *2* and the third was from Class *3*. The next three features alternated this way, as did the remainder of the data file. There is no advantage to either ordering of the data (just personal preference).

We ran our SFNN on the *iris* data and obtained the results shown in Table 1. For comparison, we also include the runs made in [5] with their new *general fuzzy NN* (GFNN), which compared favorably to two other types of FNNs, the *general fuzzy min-max NN* (GFMMNN) and the *general fuzzy hyperspheroidal NN* (GFHSNN). We also include those results for comparison with our method, as well as those of the *support vector fuzzy neural network* (SVFNN) of C.-ST. Lin et al. [4].

Table 1. Comparisons of SFNN Results, *Iris* Data

Algorithm	Ave. No. Errors	No. Runs
SFNN	1.0	8
GFNN	2.6	8
GFMMNN	3.1	8
GFHSNN	4.0	10
SFNN*	3.5	2
SVFNN**	3.5	2

* SFNN used 75 training, 75 test, vectors, 2 runs with disjoint test sets (3 and 4 errors, respectively)

** SVFNN used 75 training, 75 test, vectors, 2 single runs with 14, and 7 rules, respectively (with 3 and 4 errors)

For all of the runs shown in Table 1, except the two in the bottom section (where SFNN and SVFNN used *75* of the exemplars for training and the remaining *75* for testing), *25* *iris* feature vectors were kept out for testing and the training was done on the *125* remaining *iris* vectors. All errors were from the test vectors, where the number of incorrect classifications were averaged over all runs. In our first group of SFNN runs, all of the 125 feature vectors used for setting up the NN were correctly classified. Our first 6 sets of 25 feature vectors for testing were disjoint, but because these exhausted the 150 feature vectors, the remaining two sets of 25 feature vectors were not disjoint, but overlapped two other test sets of 25 vectors about half way.

Our numbers of Gaussians for the various runs were automatically generated to be *37* and *74*. We did not make any runs using 150 Gaussians (one for each exemplar feature vector) because then we would have no remaining feature vectors for testing. It can be seen that our results were better than all FNNs except the SVFNN with which it tied. However, the SVFNN is much more complicated.

5. ANALYSIS AND CONCLUSIONS

We see that the SFNN is indeed simple to use with no parameters required to be selected by the user. There are no training (no steepest descent), no weights, no local minima, no epochs, no defuzzification, no overtraining, and no experience required to use the SFNN. It sets up automatically when given a file of labeled vectors by associating the knowledge in the labels with the Gaussian fuzzy set membership functions. Then a file of vectors to be classified is read and put through the SFNN and the results are written to a file.

On the tests made on the difficult *iris* data set, our results were better than all except the SVFNN, which were the samed. However, the SFNN is much simpler to use than the SVFNN and avoids the Cartesian product rectangular classes formed by rules, e.g., (A is LOW) AND (B is HIGH) => (C is MEDIUM) is satisfied with pairs (a,b) in the A×B Cartesian product space.

The SFNN is a powerful classifier/recognizer with labeled exemplar data being used to set up the fuzzy set membership functions. Like all supervised learning systems, its accuracy is only as good as the accuracy of the labeled exemplars and the absence of significant noise. Instead of the final crisp classification, the highest two or three fuzzy set membership values can be kept to designate the fuzzy classification of unknown feature vectors.

This method should be used to set up a fuzzy NN for accurately labeled feature vectors that have low noise values, especially where the number of exemplar feature vectors is fewer than 1,000. We used the dificult *iris* data and 25 test vectors on each run so we could compare our results with those of other current fuzzy neural network methods. Future work will apply the SFNN to other data sets.

6. REFERENCES

[1] E. Anderson, “The Irises of the Gaspe Peninsula,” *Bull. American Iris Soc*. **59**, 2-5, 1935.

[2] B. Gabrys and A. Burgiela, “General fuzzy min-max neural network for clustering and classification,” *IEEE Trans. Neural Networks* **11**(3), 769-783, 2000.

[3] H. M. Lee, C. M. Chen and Y. L. Jou, “An efficient fuzzy classifier with feature selection based on fuzzy entropy,” *IEEE Trans. SMC-B* **31**(3), 426-432, 2001.

[4] C.-T. Lin, C.-M. Yeh, S.-F. Liang, J.-F. Chung and N. Kumar, “Support vector based fuzzy neural network for pattern classification,” *IEEE Trans. Neural Networks* **14**(1), 31-41, 2006.

[5] P.M. Patil and T.R. Sontakke, “Rotation, scale and tanslation invariant handwritten Devanagari numeral character recognition using general fuzzy neural network,” *Pattern Recognition* **40**, 2110-2117, 2007.

[6] Carl G. Looney, *Pattern Recognition Using Neural Networks*, Oxford University Press, Oxford/NY, 1997.

[7] Carl G. Looney, “Radial basis functional link neural networks and fuzzy systems,” *Neurocomputing*, **48**(1-4), 489-509, 2002.

[8] *UCI Repository of Machine Learning Databases*, Info. Computer Sci., University of California, Irvine, on-line at: http://www.ics.uci.edu/~mlearn/MLRepository.html .

[9] X. Zhang, S. Tan, C.-C. Hang, and P.-Z. Wang, “An efficient computational algorithm for min-max operations,” *Fuzzy Sets and Systems* **14**, 297-304, 1999.