High-speed Bi-directional Function Approximation Using Plausible Neural Networks

Kuo-chen Li, Dar-jen Chang, and Yuan Yan Chen

Abstract— **This paper applies a recently developed neural network called plausible neural network (PNN) to function approximation. Instead of using error correction, PNN estimates the mutual information of neurons between input layer and hidden layer. The simple theory and training algorithm of PNN lead to a faster converging rate over that of feedforward neural networks. Experiment results confirm PNN has much better training performance. In addition, the bi-directional network structure of PNN provides the flexibility of approximating any attribute of the data within a single framework. As a result, PNN can compute a function and its inverse in the same network even the inverse function generally is a one-to-many mapping.**

I. Introduction

FUNCTION approximation has attracted a great deal of research from different disciplines such as statistic, data mining, and neural networks [1][2]. Among those function approximation tools, neural networks provide a framework which can learn or approximate any function from given data samples through a training process. The black-box function representation generated by neural networks is easily used to estimate the relationship between inputs and outputs. Various neural network architectures have been proposed to be general function approximators using different training methods and activation functions. For such applications, the multilayer feedforward neural networks and radial basis function neural networks are the most popular approaches [3][4]. Although the detailed implementations vary, all feedforward neural network function approximators are based on similar theorems that have been proven to be able to approximate any continuous function to any degree of accuracy with sufficient amount of hidden neurons [5][6].

However, some drawbacks have been raised and discussed for function approximation using neural networks. First, most commonly mentioned, neural networks converge very slowly while training the networks. A great deal of research has been done to improve the convergence performance [7] [8]. Second, neural networks have the so-called "curse of dimensionality" problem, which means if there exist many local minima in the approximated function, the training may be trapped in a local minimum. In addition, the sufficient number of hidden neurons to approximate an arbitrary function might be impractically large in some cases.

In this paper, we apply a recently developed neural network called Plausible Neural Network (PNN) to function approximation. PNN is introduced by Chen in 2003 [9][10]. It is a hybrid model of estimating probabilistic and possibilistic inferences [9]. PNN uses the mutual information as the basis for approximate functions instead of training with error gradient descend. Based on this characteristic, PNN performs a rapid training with good function approximation results. Moreover, the fuzzy set theory is integrated in PNN for the continuous variable coding. Along with bi-directional feature and missing-data tolerance structure, PNN can approximate any single-valued variable as well as multi-valued variable in the same network. In the training phrase, all the variables in a PNN are considered as inputs. After the PNN is trained, users can freely decide which variables are inputs and which are outputs. In order to compare the function approximation results of PNN with those of other neural networks, we apply PNN and feedforward error back propagation neural networks to approximating the same functions.

This paper is organized as follows. Section II describes the PNN architecture as proposed by Chen. Section III illustrates how we implement function approximation using PNN. Section IV demonstrates the experiment results comparing to multilayer feedforward neural networks. Section V closes the paper with conclusions and open issues of using PNN for function approximation.

II. Plausible neural network

A. Network Architecture

A basic PNN architecture consists of two layers (input layer and hidden layer) of cooperative and competitive neurons with complete, bidirectional, and symmetric connections. Fig.1 shows the basic architecture for a PNN model. Each input attribute is encoded into a group of competitive neurons which uses winner-take-all (WTA) algorithm to interpret the value of the attribute(e.g. in Fig. 1 the input WTA ensemble (X_1, X_2, X_3) encodes the values of attribute A1). WTA not only works with mutual information content weights but also has the computational power of nonlinear activation functions [11]. Afterward, the input WTA ensembles cooperate to determine the values of the WTA ensemble in the hidden layer. Each hidden neuron, generally speaking, represents a pattern or cluster for the

Kuo-chen Li is with the University of Louisville, Louisville, KY 40241, USA. (e-mail: k0li0005@louisville.edu)

Dar-jen Chang is with the University of Louisville, Louisville, KY 40241, USA (corresponding author to provide phone: 502-852-0472; fax: 502-852-4713; e-mail: djchan01@uofl.edu)

Yuan Yan Chen is with PNN technologies Inc, Woodbridge, VA 22195, USA (e-mail: yan_chen@pnntech.com)

given training dataset. In other words, while inputting a data sample to the trained PNN, PNN is able to determine which patterns or clusters the data sample might contain or belong to.

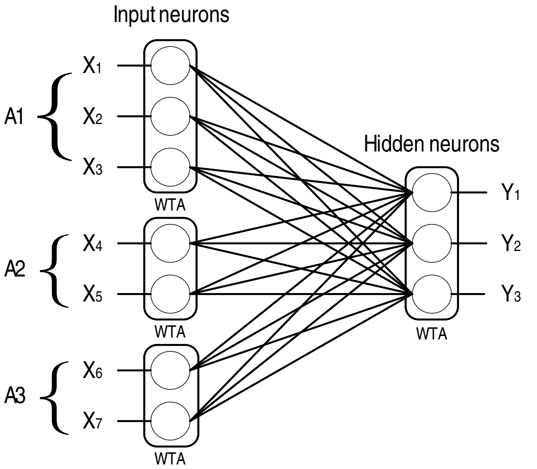

Fig. 1. A general PNN architecture for a dataset contains three attributes (A1, A2, and A3).

B. *Attribute Value Coding*

To encode the attribute value into a WTA ensemble, first, each WTA ensemble has to be under one condition: for a WTA ensemble ($X_1, X_2, \ldots ,X_k$) where

$$\sum_{i=1}^{k} X_i = 1 \text{ and } 0 \leq X_i \leq 1 \text{ for all } X_i$$

In this manner, for a categorical attribute, we can use one neuron to represent one categorical value intuitively. For example, an attribute, color, with three possible values (*red*, *green*, and *blue*) can be expressed by a WTA ensemble with three neurons (X1, X2, X3). An input (1,0,0) represents *red*, such as, (0,1,0) is *green* while (0,0,1) as *blue*.

As to the continuous attribute, a fuzzy set coding fits in perfectly for the WTA ensemble. Each neuron in the continuous-attribute WTA represents one fuzzy membership function and the value is the degree of membership for the specific membership function. For example, a continuous attribute ranged from 0 to 6 can be expressed by a four-neuron WTA ensemble (X1, X2, X3, X4). Assuming the triangular fuzzy membership function is chosen, the WTA represents four fuzzy membership functions where the centers locate at 0, 2, 4, and 6. The encode process is called fuzzification. A continuous value, say 3.2, can be encoded as (0, 0.4, 0.6, 0) representing the degrees of membership for each corresponding membership function. To recover the value from the encoded fuzzy set, defuzzification is applied. From the last example, 3.2 can be recovered from 0.4*2+0.6*4. In PNN terms, each fuzzy set in the ensemble is called a bin and a triangular bin is a triangular membership function. The number of bins and the type of fuzzy membership functions chosen to encode a continuous attribute are up to the user.

For this coding scheme, PNN can handle different types of attributes within the frame work. In addition, uncertain value and missing data can be resolved easily. For example, an input (0.5, 0.5, 0) for the color attribute clearly expresses uncertain values for red and green. Missing attribute values are represented by a null vector, i.e. all X_i's in the attribute ensemble are zero. With this coding, since the input from every neuron in the attribute ensemble is zero, activation potential (i.e. input times the weight) contributed by the attribute is zero. As a consequence, the attribute with missing value will not participate in the WTA activation of the competing hidden neurons.

C. *Connection Weights*

The weight definition in PNN is based on the mutual information content which can determine the strength of the relationship between an input neuron and a hidden neuron. Consider two neurons, *x* and *y*, where the input for neurons is continuous variable in [0,1], which represent the state of neurons, and the weight between two neurons is given by the mutual information content. (Note we call (1) mutual information content since (1) is a factor in the mutual information formula.)

$$\omega_{ij} = \ln\left(\frac{P(X_i, Y_j)}{P(X_i)P(Y_j)}\right) \tag{1}$$

Based on the given weight definition, if X_i, Y_j are independent, $P(X_i,Y_j)= P(X_i)*P(Y_j)$ and we can compute W_{ij}=0. It shows there is no relationship between X_i and Y_j. On the other hand, if the calculated weight is positive, it is called positively associated. It indicates that neuron Y_j is more likely fire if neuron X_i fires. If the weight is negative, it is called negatively associated. That means neuron Y_j is more likely rest if neuron X_i fires. Using mutual information contents as connection weights makes the explanation of knowledge transparent to the evaluated weights. Moreover, its statistical inference fits the possibility measure inference for PNN. Combining WTA activation function, which gets the max values of possibility measurements, completes the PNN inference.

D. *Forward and Reverse Firing*

One of the properties of PNN is bidirectional weight. PNN can be carried out in both directions between input neurons and hidden neurons. In this sense, forward firing is referred to triggering input neurons to activate the hidden neurons and reverse firing is referred to triggering hidden neurons to activate input neurons.

For the forward firing, to determine which hidden neurons to activate, a competition method is applied for each hidden neuron in the WTA ensemble. Consider a *m**n* PNN network, there are *n* competitive hidden neurons, $y_1,y_2\ldots y_n$, in the same WTA, the input vector is $[x_1,x_2,\ldots x_m]$. Each hidden neuron takes input values from the input vector and multiplies with the corresponding weights. Through the WTA algorithm and the competition method, the output of hidden neuron can be represented by the following:

$$y_j = S(\sum_i w_{ij}x_i), \forall j, \frac{e^{\sum_i w_{ij}X_i}}{\sup_j e^{\sum_i w_{ij}x_i}} > \alpha \qquad (2)$$

$$y_j = 0, otherwise$$

where S(t_j) is the normalization function

$$S(t_j) = \frac{e^{\kappa t_j}}{\sum_j e^{\kappa t_j}} \qquad (3)$$

In (2), α is a threshold value to cut the weak signal. Each competitive neuron in WTA has to obtain a value greater than α to become a winner. Since the above firing method could produce multiple winners depending on the α value, sometimes, it is referred to the soft-max competition method. On the other hand, if we set up PNN to determine only one winner at a time, it is referred to the hard-max competition method. In (3), κ is the temperature argument that can amplify the signals. The default setting for κ is usually 1.

Unlike the traditional neural networks, bi-directional PNN structure allows to reverse firing direction. In that case, the input vector is taken from the hidden layer, and the outputs can be computed at the input layer. The reverse firing works the same way as forward firing only in the different direction. However, if we reverse fire to a continuous attribute, a defuzzification needs to be applied after the winners are determined.

E. Training Algorithm

In PNN, a training method is required to estimate the weights which contain the max information knowledge between input neurons and hidden neurons for the given training dataset. To evaluate weights, we have to evaluate $P(X_i,Y_j)$, $P(X_i)$, and $P(Y_j)$ based on the definition given in (1). We can use the training dataset to evaluate $P(X_i,Y_j)$, $P(X_i)$, and $P(Y_j)$. Given the past n co-firing history of two neurons X_i and Y_j, ($X_{i\,k}$, Y_{jk}), k = 1,2,…,n, based on the binary coding and fuzzy set coding, the maximum estimate likelihood function in weight connection can be denoted as:

$$\omega_{ij} = \ln\left(\frac{n\sum_{k=1}^{n} x_{ik}\,y_{jk}}{\sum_{k=1}^{n} x_{ik}\sum_{k=1}^{n} y_{jk}}\right) \qquad (4)$$

The learning method is based on the computation of belief. Each training procedure measures the action potentials for hidden neurons. The training algorithm is shown in Fig. 2. First, PNN fires hidden neurons randomly for each training sample in order to produce an initial fire table. Based on the initial fire table and input training samples, PNN calculates the weight tables for each weight connection between input neurons and hidden neurons. After a new weight table estimated, PNN fires each training sample using firing method to get a new fire table. The next step is to compare two fire tables. If two fire tables are identical, it means weight table hasn't changed and PNN is stable. Otherwise, based on the new fire table, estimate the new weight table and repeat previous steps until PNN is stable.

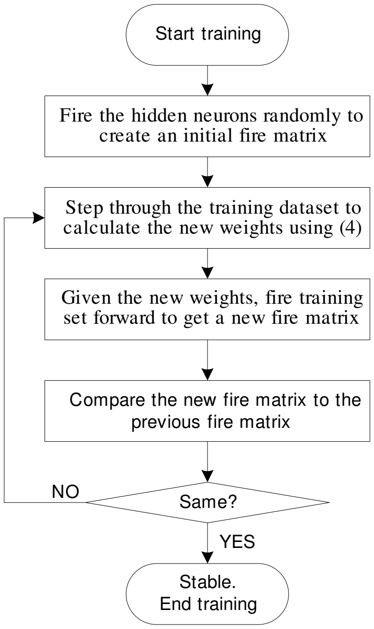
Start training
Fire the hidden neurons randomly to create an initial fire matrix
Step through the training dataset to calculate the new weights using (4)
Given the new weights, fire training set forward to get a new fire matrix
Compare the new fire matrix to the previous fire matrix
NO
Same?
YES
Stable.
End training

Fig. 2. The PNN Training Algorithm

III. FUNCTION APPROXIMATION USING PNN

PNN provides a unified architecture for multiple tasks related to pattern recognition since the hidden neurons represent the hidden patterns or clusters for the dataset. Based on the discovered patterns, several data analysis tasks can be applied such as clustering, classification, rule discovery and prediction. PNN also can apply to function approximation by interpreting the found patterns. The architecture of PNN for function approximation is same as that shown in Fig. 1. Instead of pre-defining input/output attributes as done in the feedforward neural networks, all the attributes are considered as inputs in PNN during the training phase. The main PNN design factor at this step is to determine how many hidden neurons are sufficient to capture all the patterns in the given dataset. After PNN is trained, users can freely use a set of known attributes to approximate any unknown attribute in the network.

The idea behind the function approximation using PNN is described next. In a trained PNN, given the known attributes, (X_1,X_2…,X_n), we want to compute the unknown attribute Y representing the functional relationship $Y= f$ (X_1,X_2…,X_n). First, all the attributes (X_1,X_2…,X_n, and Y) are encoded to an input vector based on the PNN coding scheme. Since Y is unknown, it is treated as a missing value which is coded as a null vector. Next, we fire the input vector forward to trigger the hidden neurons. The activated hidden neurons, in general, indicate the combined patterns this input vector contains. We then reverse fire the output of the hidden neurons back to the input layer. In the reverse firing, we only need to fire to the unknown attribute, *Y*. In other words, we use the reverse firing to find the possible value or values of *Y* that are associated with the fired pattern. After the forward and reverse firing computation, the approximated value of attribute *Y* can be obtained from the output of the WTA ensemble using the coding scheme.

Due to the PNN coding scheme, the output may contain multiple values in a single WTA ensemble (attribute). To extract the multiple values from a categorical attribute, α cut is applied. Specifically every competitive neuron in the WTA with value greater than α cut value is considered to be a potential output. As to the continuous attribute, multiple values may overlap in the same WTA. For example, the continuous attribute shown in section II contains the multiple output values *2.2* (*0, 0.9, 0.1, 0*) and *5.6* (*0, 0, 0.2. 0.8*). If PNN can approximate the values correctly, the output for the attribute would be (*0, 0.45, 0.15, 0.4*) since two fuzzy sets have been overlapped and normalized. A simple algorithm is devised to separate multiple values that may be contained in a given fuzzy set. Suppose the continuous attribute is encoded as a fuzzy set (X_1,X_2…,X_n). The algorithm is proceeded as follows:

1) Identify all the local minima in the fuzzy set, where a local minimum, X_i, satisfying $X_i < X_{i-1}$ and $X_i <= X_{i+1}$. The local minima are numbered from left to right.
2) Based on the local minima, separate the fuzzy set. For i-th local minimum, create a new fuzzy set (Y_1,Y_2…,Y_n), where $Y_j = X_j$ if j is the index of the fuzzy member between the i-th local minimum fuzzy member and (i-1)-th local minimum fuzzy member; otherwise, Y_j = 0.
3) If there exists positive values from the last local minimum to the last fuzzy member. Create a fuzzy set consisting of all the fuzzy members from the current local minimum to the last fuzzy member.
4) Normalize the fuzzy sets produced in step 2 and 3.
5) Perform defuzzification of each fuzzy set from step 4.

To give an example of the above algorithm applications, consider the combined fuzzy set (*0, 0.45, 0.15, 0.4*) (using triangular membership functions centered at 0, 2, 4, and 6, respectively). There is only one local minimum in the set, namely, *0.15*. Using step 2) of the algorithm, we create two fuzzy sets: (*0, 0.45, 0.15, 0*) and (*0, 0, 0.15, 0.4*). Using step 3), we obtain the normalized fuzzy sets (*0, 0.75, 0.25, 0*) and (0, 0, 0.27, 0.73). And finally using step 4), we obtain the values *2.5* and *5.46* from the normalized fuzzy sets, respectively. The example shows that if positive memberships of the multiple values are overlapped in the close proximity of the combined fuzzy set (due to the small number of bins used in the example coding), the multiple values can only be approximated but not be exactly predicted. One approach to avoiding the close overlapping is to use a larger number of bins for the continuous attribute coding.

To achieve better results of function approximation, low α cut is suggested for the hidden layer. Although higher α cut leads to faster convergence, it also produces a discrete function for the approximation due to the lost of information. Experiment also shows that the temperature argument κ can help to avoid trapping in the local minima. In addition, higher temperature also leads to faster convergence. However, higher temperature also amplifies the separation of signals and thus may introduce unwanted noise.

IV. Experiments

In this section, we present several experimental results for function approximation using PNN. To compare the results, a feedforward neural network (FFNN) is implemented to approximate the same functions as used for the PNN experiments. First, a Gaussian function $y = e^{-(x-\mu)^2/2\sigma^2}$ is used to test the performance of PNN and FFNN. We set up the Gaussian function with μ=50 and σ=15. The independent variable of the Gaussian function , X, ranges from 0 to 100. We generate 150 points with outliers for the training dataset and 350 points for testing.

For the PNN configuration, 30 bins are specified for each continuous attribute, and the number of hidden neurons is 20. To train the PNN, we assign α cut = 0.001 and κ = 1. The number of maximum iterations is set to 300. In this experiment, PNN stabilized in 164 iterations with mean square error (MSE) 0.0019. The MSE for the testing dataset tested with the trained PNN is 0.0023. Fig. 3 shows the result of this approximation using PNN. The result also shows the capability of PNN to reduce the effect of the outliers.

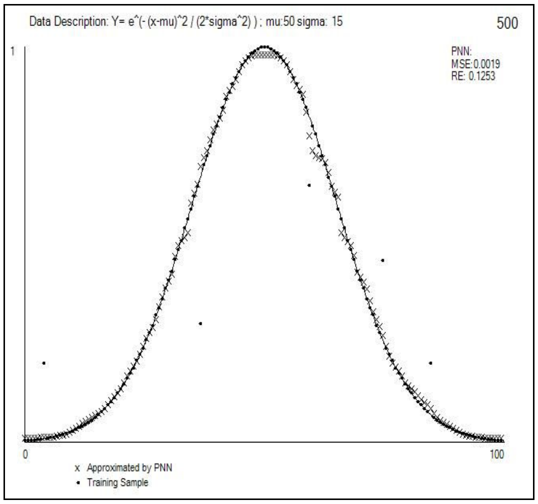

Fig. 3. Approximate Gaussian function using PNN

Fig. 4 shows the result of the approximation of the same function using FFNN. The neural network contains one hidden layer with 20 hidden neurons. The error criterion is set to 0.001 and the maximum iteration is 10,000. The result in Fig. 4 indicates the error criterion is not satisfied. The training stops at 10,000 iterations with mean square error 0.0018. Comparing the results, to meet the similar error criterion, PNN converges much faster than the FFNN.

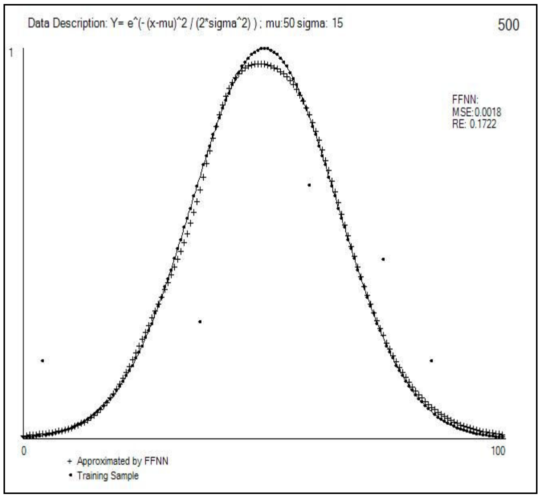

Fig. 4. Approximate Gaussian function using FFNN

Table 1 and Fig. 5 show the mean square errors of the first ten iterations for both PNN and FFNN in the last experiments. As shown, PNN reduces the error much more rapidly than FFNN does in the first 10 iterations. However, the error fluctuates slightly (increases sometimes) when PNN is near stabilized because the PNN training is based on maximizing the total mutual information rather than on minimizing the error function.

Table 1. The MSE of the first 10 iterations for PNN and FFNN

	1	2	3	4	5	6	7	8	9	10
PNN	0.136	0.135	0.13	0.114	0.073	0.018	0.0025	0.0023	0.0021	0.002
FFNN	0.205	0.202	0.198	0.195	0.192	0.19	0.188	0.186	0.185	0.184

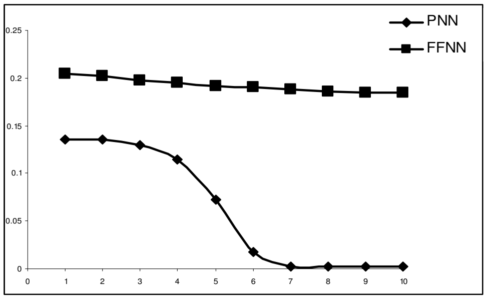

Fig. 5. The MSE of the first 10 iterations for PNN and FFNN

Another experiment is conducted to test the ability of PNN to approximate multi-valued function. We use the trained PNN in the previous experiment to approximate the inverse of the Gaussian function, which is a multi-valued function. The input testing set consists of 100 Y values ranging from 0 to 1. We want to compute the associated X values of the input Y values. In Fig. 6, we show 197 (X,Y) points that are computed from the given Y values using the trained PNN. Note that each Y value produces two X's except the top three Y's whose associated X values are too close to separate by the algorithm proposed in section III. However, the result shows the flexibility and capability of PNN to approximate single-valued as well as multi-valued functions.

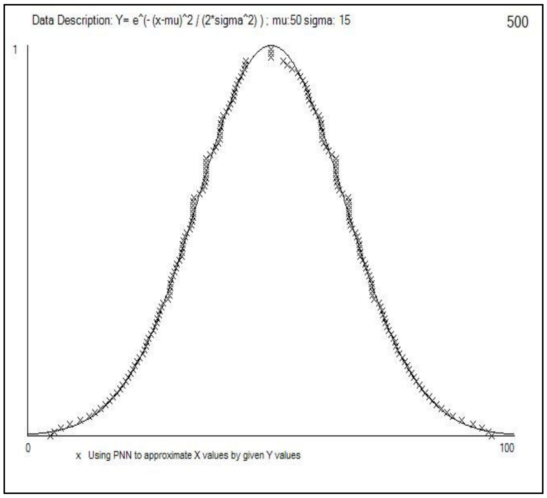

Fig. 6. Approximate multi-value function using PNN

Same configurations for both PNN and FFNN are applied to approximate another function *Y=sinc(X)+1*. The training set consists of 150 (X,Y) points, where X ranges from 0 to 30. And 350 (X,Y) points are created for testing. PNN is stable after 283 training iterations. On the other hand, FFNN fails again to converge to the specified error criterion after 10,000 training iterations. In addition, after the training, the PNN has the mean square error of 0.0013 which is better than FFNN's 0.0035. Validating the testing data on the trained PNN and FFNN clearly shows that PNN (with MSE=0.0031) outperforms FFNN (with MSE=0.0081).

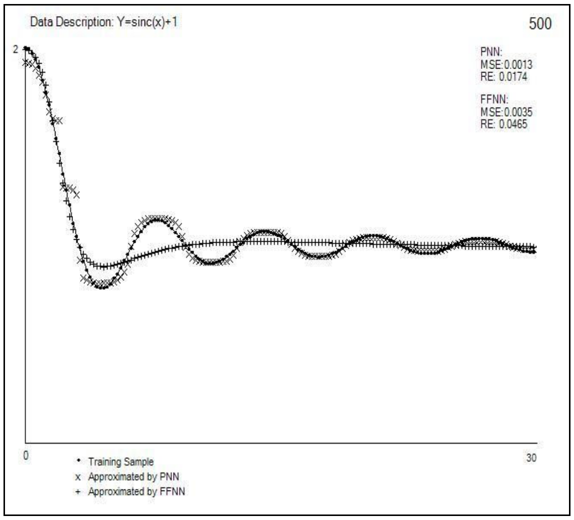

Fig. 7. Function approximation using PNN and FFNN

Fig. 7 illustrates the combined results of PNN and FFNN. Although in theory FFNN can approximate any real continuous function to any degree of accuracy with sufficient amount of hidden neurons, finding sufficient hidden neurons to satisfy the desired accuracy with reasonable training time is not very practical.

Fig. 8 shows the outcome of approximating a free-drawing function using a PNN. In this PNN, 40 bins are assigned for each attribute and 200 (X,Y) points of the function are used to train the PNN. The training process is stable in 236 iterations. To validate the performance, 100 Y values ranging from 0 to 1 are input to the PNN to approximate X values. For each of the testing Y value, the PNN can approximate the associated X values, which may vary from zero to four distinct values. However, noise caused by the PNN computation produces inappropriate approximation in some testing values as shown in Fig. 8.

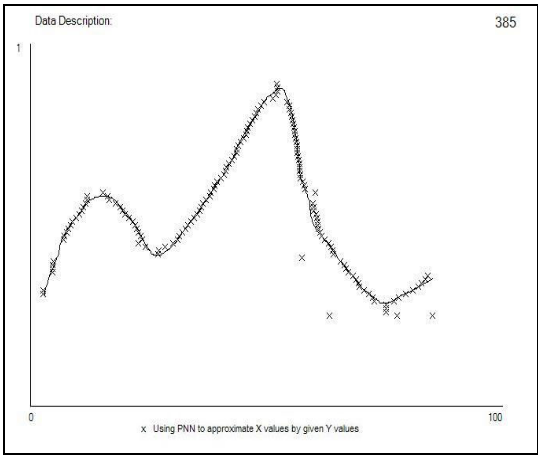

Fig. 8. PNN approximates the inverse of a free-drawing function (i.e. multi-valued approximation)

V. Conclusions and open issues

The proposed PNN for function approximation provides faster convergence and unified bi-directional function approximation. The traditional neural networks even with robust learning algorithms [8] need hundreds of iterations to get quality results, whereas PNN can reach acceptable results in dozens of iterations. Our experiments with many free-drawing functions support our belief that PNN using the training algorithm based on mutual information with sufficient training set and hidden neurons can approximate any continuous function to any degree of accuracy. Insufficient training set or hidden neurons, however, may compromise the approximation performance of PNN. The training set is considered sufficient if it contains important patterns or features of the approximated function.

The capability of PNN to approximate a function and its multi-valued inverse in the same network is a very unique feature not available in other neural network architectures. In terms of knowledge representation, a PNN can be used to represent all the relationships among attributes. In contrast, other neural networks can only be used to represent specific relationships between selected input and output attributes. The algorithm (Section III) used to recover the multiple values from the overlapped WTA fuzzy set in a PNN works well if the multiple values are relatively separable. It is still an issue open for further research if the multiple values are close to each other. Fig. 6 and Fig. 8 show the inadequate separations of the inversed multiple values near the local minima or maxima of the functions using the algorithm. For future research work, we like to extend the algorithm to enhance its separation power. Although applying PNN to the high-dimensional function approximation problem has not yet been tested, we believe that the simplicity and quick convergence of PNN can handle such a large-scale problem without much difficulty.

References

[1] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA, 1998.

[2] A. R. Barron and R. L. Barron, *Statistical learning networks: a unified view,* in "Symposium on the Interface: Statistics and Computing Science", Reston, Virginia, April, 1988.

[3] J. Zhang, G. G. Walter, Y. Miao, and W.N. W. Lee, "Wavelet Neural Networks for Function Learning," IEEE Trans. Signal Processing, Vol. 43, p1485-1497, June 1995.

[4] R. J. Shilling, J. J. Carroll and A. F. Al-Ajlouni, "Approximation of nonlinear systems with radial basis function neural networks", IEEE Transactions on neural networks, vol. 12, no. 1, 2001.

[5] K. Hornik, M. Stinchcombe and H. White, Multilayer feedforward networks are universal approximators, Neural Networks, 2 (1989), 359-366.

[6] M. J. D. Powell, The theory of radial basis function approximation, in " Advances in Numerical Analysis III, Wavelets, Subdivision Algorithms and Radial Basis Functions", (W. A. Light Ed.), Clarendon Press, Oxford, 1992, pp. 105-210.

[7] D. S. Chen and Ramesh C. Jain, "A Robust Back Propagation Learning Algorithm for Function Approximation", IEEE Trans. Neural Networks, Vol. 5, May 1994.

[8] Sheng-Tun Li, Shu-Ching Chen. "Function Approximation Using Robust Wavelet Neural Networks," *ictai*, p. 483, 14th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'02), 2002.

[9] Y. Y. Chen, "Plausible neural networks," *Advance in Neural Networks World,* A. Grmela and N. E. Mastorakis, Ed. WSEAS Press 2002, pp. 180-185.

[10] Y. Y. Chen, "Plausible neural network with supervised and unsupervised cluster analysis," U.S. Patent 20030140020, July 24, 2003.

[11] W. Maass, "On the computational power with winner-take-all," *Neural Computation*, 12 (11), 2000,pp. 2519-2536.