Growing and Learning Algorithms of Radial Basis Function Networks

by

Tiantian Xie

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 4th, 2013

Keywords: Radial basis function network, Incremental design,
Error Back Propagation, Levenberg Marquardt, neural networks

Abstract

Radial Basis Function (RBF) network is a type of artificial neural network, which uses the Gaussian kernel activation function. It has a fixed three-layer architecture. The RBF network is easier to be designed and trained than traditional neural networks, and they can also act as an universal approximator. They have good generalization properties and can respond well for patterns which are not used for training. RBF networks have strong tolerance to input noise, which enhances the stability of the designed systems. Therefore, RBF network can be considered as a valid alternative for nonlinear system design.

This work presents an improved second order algorithm for training RBF networks. The output weights, the centers, widths, and input weights are adjusted during the training process. More accurate results will be obtained by increasing variable dimensions. Taking the advantages of fast convergence and powerful search ability of second order algorithms, the proposed algorithms can reach smaller training and testing errors with a much less number of RBF units.

A new error correction algorithm is proposed for the incremental design of radial basis function networks. In this algorithm the number of RBF units is increased one by one until the training evaluation reaches desired accuracy. The initial center of the newly added RBF unit is properly selected based on the location of highest peak/lowest valley in the error surface; while for the other RBF units, the initial conditions are copied from the training results of the last step. Parameter adjustments, including weights, centers, and widths are performed by the Levenberg Marquardt algorithm. This algorithm is very efficient to design a compact network by comparing with other sequential algorithms in constructing radial basis function networks. The duplicate patterns test and the noise patterns test are applied to show the robustness of the proposed algorithm.

Acknowledgments

First of all, I would like to sincerely thank my supervisor, Prof. Bogdan M. Wilamowski, for his great patience and knowledgeable guidance during the past three years of Ph.D study. His professional research experience taught me how to be creative, how to find problems, and solve them. His active attitude of life encourages me to work hard towards my destination. His kindness and great sense of humor made me feel warm and happy. All the things I have learned from him are marked deeply in my memory and will benefit the rest of my life. Without his help, I could not have finished my dissertation and Ph.D study successfully. Also, I would like to express my special appreciation to both Prof. Bogdan Wilamowski and his wife, Mrs. Barbova Wilamowski, for their kindness, caring about me, and letting me feel like studying at home.

Special thanks are also given to my committee members, Prof. Hulya Kirkici, Prof. Vitaly Vodyanoy, and the outside reader Prof. Pradeep Lall. From their critical and valuable comments, I noticed the weakness in my dissertation and made the necessary improvements according to their suggestions.

I would like to express my appreciation to my good friends who have helped me with my studying and living in Auburn. They are Yuehai Jin, Haitao Zhao, Hua Mu, Jinho Hyun, Qing Dai, Yu Zhang, Chao Han, Xin Jin, Pengcheng Li, Fang Li and Jiao Yu. I am very lucky to be their friend.

I also would like to thank Prof. John Hung, Prof. Fa Foster Dai, Prof. Hulya Kirkici, Prof. Vishwani Agrawal and Prof. Bogdan Wilamowski for their excellent teaching skills and professional knowledge in their courses.

Last but not least, I am greatly indebted to my husband, Dr. Hao Yu, my daughter, Amy, and my parents. They are the backbone and origin of my happiness. Without their support and encouragement, I could never finish my Ph.D study successfully. I owe my every achievement to my family.

Thanks to everyone.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ANN: artificial Neural Network

NN: Neural Network

LM: Lavernberg Marquardt

MLP: Multilayer Perceptron

FCC: Fully Connected Cascade

RBF: Radial Basis Function

ISO: Improved Second Order

ErrCor: Error Correction

RAN: Resource-allocating network

EKF: Extended Kalman Filter

LMS: Least Mean Square

MRAN: Minimal Resource Allocating Network

GAP: Growing and Pruning RBF networks

GMM: Gaussian Mixture Model

Chapter 1
Introduction

An Artificial Neural network (ANN) is a mathematical model which can infer a function from training data. Neural networks can be used for data classification [1], pattern recognition [2] and function approximation [3]. Neural networks are also applied for solving various problems in industrial applications, such as nonlinear control [4], image/audio signal processing [5], system diagnosis, and faults detection [6]. In this dissertation we will focus on one kind of ANN which is called Radial basis function (RBF) networks. We will discuss the differences between RBF network and ANN, the advantages and challenges of a RBF network over a traditional ANN, and the improvements we have made in the design of a RBF network.

1.1 Basic concepts of neural networks

An neural network is a typical supervised learning method, which is inspired by biological neural networks. It consists of an interconnected group of neurons. Its structure can be changed during a learning phase. A single neuron includes the linear/nonlinear activation function f(x) and weighted connections as shown in Fig. 1.1.

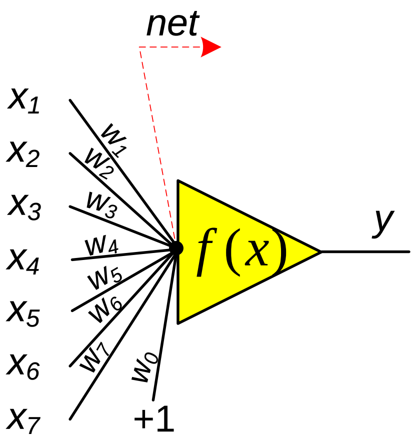


Fig. 1.1 Single neuron structure

There are two steps for a single neuron calculation:

Step 1: Calculate the *net* value as sum of weighted input signals:

$$net = \sum_{i=1}^{7} x_i w_i + w_0 \tag{1-1}$$

Step 2: Calculate the output y:

$$y = f(net) \tag{1-2}$$

There are a number of common activation functions in use with neural networks, such as step function, linear function, sigmoidal shape function, and radial basis function.

For more neurons interconnected together, the two basic computations (1-1) and (1-2) for each neuron remain the same; the only difference is that the inputs of a neuron could be provided by either the outputs of neurons from previous layers or network inputs.

Technically, the interconnections among neurons can be arbitrary. The most popular architecture is multilayer perceptron network (MLP). The other common neural network architecture is full connected cascade (FCC) network. In FCC networks all possible routines are weighted and each neuron contributes to a layer; it can solve problems using the smallest possible number of neurons. Fig. 1.2 shows the architecture of MLP with one hidden layer and FCC network.

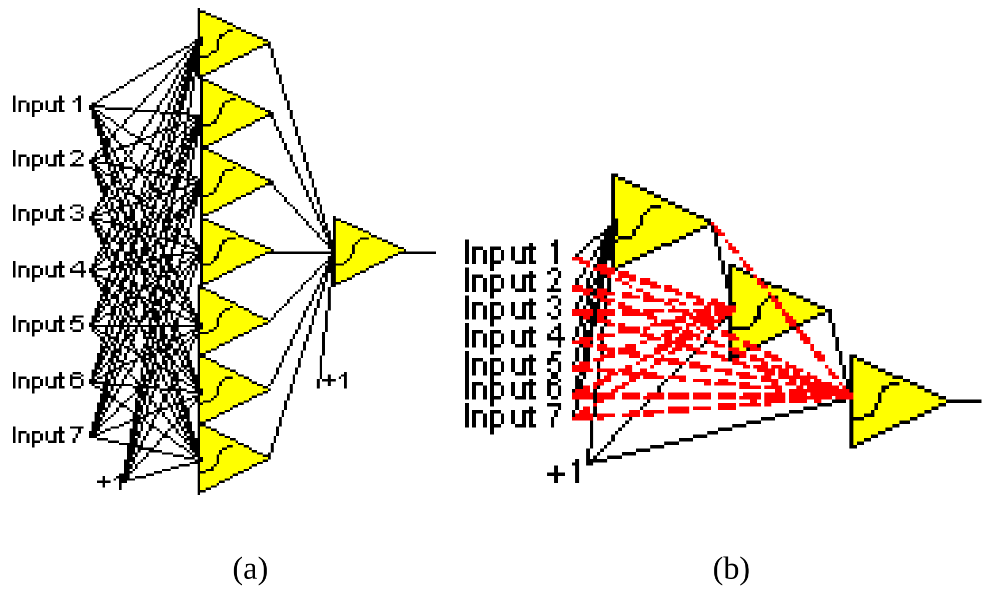


Fig.1.2 (a) Standard MLP network architecture with one hidden layer; (b)FCC networks

The radial basis function network is a type of artificial neural network that uses radial basis functions as activation functions and MLP architecture. It can be used for application to problems of supervised learning, such as regression, classification, and time series prediction.

1.2 Basic concepts of Radial Basis Function Networks

RBF networks have a fixed three-layer architecture which consists of input layer, hidden layer, and output layer. It has the similar layer-by-layer topology as multilayer perceptron networks. The input layer provides network inputs; the hidden layer remaps the input data in order to make them linearly separable; the output layer does linear separation. Fig.1.3 shows the general form of RBF networks with I inputs, H hidden units and M outputs.

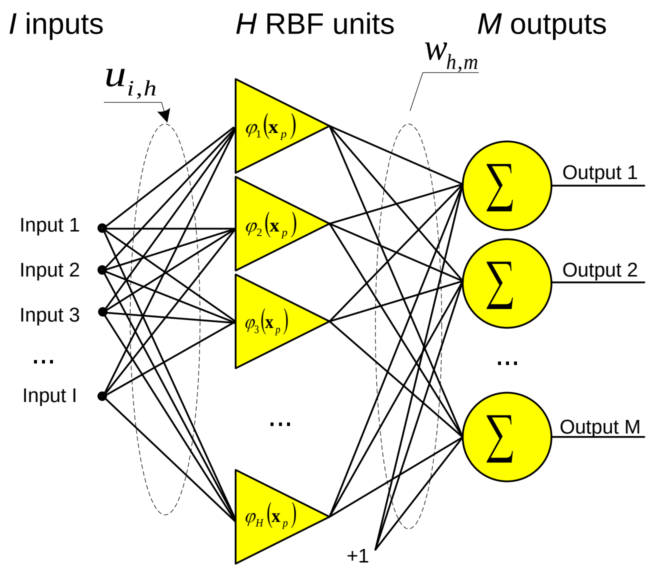


Fig. 1.3 RBF network with I inputs, H hidden units and M outputs.

The basic computations in the RBF network above include:

i) Input layer computation

At the input of the hidden unit, the input vector x is weighted by input weights $u_{i,h}$ which represents the weight connection between the *i*-th input and RBF unit *h*:

$$y_{p,h,i} = x_{p,i} u_{i,h} \tag{1-3}$$

ii) Hidden layer computation

The output of the hidden unit is calculated by:

$$\varphi_h(\mathbf{x}_p) = \exp\left(-\frac{\left\| \mathbf{y}_{p,h} - \mathbf{c}_h \right\|^2}{\sigma_h} \right) \tag{1-4}$$

Where: the activation function $\varphi_l(\bullet)$ for hidden unit h is normally chosen as Gaussian function; c_h is the center of hidden unit; σ_h is the width of hidden unit.

iii) Output layer computation

The network output m is calculated by:

$$o_{p,m} = \sum_{h=1}^{H} w_{h,m} \varphi_h(\mathbf{x}_p) + w_{0,m} \tag{1-5}$$

Where: m is the index of output; $w_{h,m}$ is the output weight between hidden unit l; output unit m; $w_{0,m}$ is the bias weight of output unit m.

From the special architecture of the RBF networks, we can see the design of RBF network involves three fundamental steps: (1) find proper network size; (2) find proper initial parameters (centers and widths); (3) train the networks.

1.3 Development of Radial Basis Function Networks

The concept of RBF network originated from Cover's Theorem on separability of patterns in 1965 which state that a complex pattern-classification problem cast in a high-dimensional space nonlinearly is more likely to be linearly separable than in a low-dimensional space provided that the space is not densely populated[7]. In 1988, J.Moody and C.J. Darken first proposed the network architecture which uses a single internal layer of locally-tuned processing units to learn both classification tasks and function approximations [8]. In 1991, S. Chen, C.F.N. Cowan and P.M. Grant proposed the orthogonal least squares learning algorithm for RBF network [9]. This algorithm quantified the impact of the input to output. It chose the initial parameter based on this impact other than choosing it randomly. In 1992, D. Wettschereck and T.Dietterich employed a supervised learning of the center locations as well as the output weights to train RBF networks [10]. This supervised training process is much more efficient than unsupervised RBF. In 1997, Karayiannis, N. B proposed a framework for constructing and training RBF networks which merging supervised and unsupervised learning with network growth techniques [11]. In 2004, Saratchandran,P. and Sundararajan, N. proposed an algorithm referred to as growing and pruning(GAP)-RBF uses the concept of “Significance” of a neuron and links it to the learning accuracy [12]. It used a piecewise-linear approximation for Gaussian function to derive a way of computing the significance. This algorithm can provide comparable generalization performance with a considerably reduced network size and training time.

Since the RBF network architecture was invented, it has been used in many different areas. Sue Inn Ch’ng et al used an adaptive momentum Levenberg-Marquardt RBF for face recognition [13]. RBF networks can also be applied in fault diagnosis [14-15], industrial control [16, 17], image processing [18,19]; and system identification [20].

1.4 The sequential algorithms of constructing RBF networks

1.4.1 Resource-allocating network (RAN) and RAN-EKF algorithm

RAN architectures were found to be suitable for online modeling of non-stationary processes. In this sequential learning method, the network starts with a blank slate: no patterns are yet stored. As patterns are presented to it, the network chooses to store some of them. At any given point the network has a current state, which reflects the patterns that have been stored previously [21]. The training process is as follows:

1. The allocator identifies a pattern that is not currently well represented by the network and

allocates a new unit that memorizes the pattern. After the new unit is allocated, the desired output is marked as T.

2. The center of the kernel function of RBF network is set to the novel input I:

$$c_i = I \tag{1-6}$$

The linear synapses on the second layer are set to the difference between the output of the network and the novel output:

$$h_i = T - y \tag{1-7}$$

The width of the response of the new unit is proportional to the distance from the nearest stored vector to the novel input vector:

$$w_i = k\|I - c_{nearest}\| \tag{1-8}$$

Where k is an overlap factor; as k grows larger, the responses of the units overlap more and more.

3. The RAN uses a two-part novelty condition. An input-output pair (I, T) is considered novel if the input is far away from existing centers and if the difference between the desired output and the output of the network is large:

$$\|I - c_{nearest}\| > \delta(t) \tag{1-9}$$

$$\|T - y\| > \epsilon \tag{1-10}$$

where ϵ is a desired accuracy of output of the network. Errors larger than ϵ are immediately corrected by the allocation of a new unit, while errors smaller than ϵ are gradually repaired using gradient descent. The distance $\delta(t)$ is the scale of resolution that the network is fitting at the *tth* input presentation. The learning starts with

$$\delta(t) = \delta_{max} \tag{1-11}$$

where δ_{max} is the largest length scale of interest, typically the size of the entire input space of non-zero probability density. The system creates a coarse representation of the function at first, then refines the representation by allocating units with smaller and smaller widths. Finally, when the system has learned the entire function to the desired accuracy and length scale, it stops allocating new units altogether.

If the new pattern satisfies these two criteria, the RAN is grown; otherwise, the existing network parameters are adjusted using a least mean square (LMS) gradient descent whenever a new unit is not allocated.

The RAN-EKF algorithm is proposed to improve RAN-EKF by using an extended kalman filter algorithm (EKF) instead of the LMS to estimate the network parameters [22]. It is more compact and has better accuracy than RAN.

1.4.2 Minimal Resource Allocating Network (MRAN) Algorithm [23]

In the constructive stage of MRAN network, three criteria for adding hidden units are employed. Among them, the first two are identical to those in RAN and RAN-EKF for adding hidden units. The third criterion is called RMS criterion. It uses the Root Mean Square value of the output error over a sliding data window before adding a hidden neuron. The RMS value of the network output error at *n*th observation e_{rmsn} is given by:

$$e_{rmsn} = \sqrt{\sum_{i=n-(M-1)}^{n} \frac{e_i^* e_i}{M}} \quad (1\text{-}12)$$

The third growth criterion to be satisfied is

$$e_{rmsn} > e_{min} \quad (1\text{-}13)$$

Here, e_{min} is a threshold value to be selected. This equation checks whether the network has met the required sum of squared error specification for the past M outputs of network. This newly added pruning strategy removes the superfluous hidden neurons possibly generated during the network constructive phase and makes the network more compact.

1.4.3 Growing and Pruning (GAP) RBF networks [24]

The algorithm referred to as GAP-RBF uses the concept of "Significance" of a neuon and links it to the learning accuracy. "Significance" of a neuron is defined as its contribution to the network output averaged over all the input data received so far. This requires the knowledge of the input data distribution. Using a piecewise linear approximation for the Gaussian functions, the way of computing this significance was derived:

$$E_{sig}(k) = \left|\frac{1.8\sigma_k{}^l\alpha_k}{s(X)}\right| \tag{1-14}$$

Where σ is width of the hidden neuron, α is the connecting weight to output neuron; $s(X)$ is the range of the input pattern; l is the dimension of the input space.

Given an approximation error e_{min}, for each observation (x_n, y_n), the process of GAP-RBF algorithm is described bellow:

1. Compute the overall network output.
2. Apply the three criterions in function 1-9, 1-10, 1-13. If the new pattern satisfies these three criteria, allocate a new hidden neuron.
3. If the new pattern does not satisfy, adjust the network parameters using the EKF method.
4. Check the criterion for pruning the hidden neuron: If $\left|\frac{1.8\sigma_k{}^l\alpha_k}{s(X)}\right| < e_{min}$, remove the *k*th hidden neuron.

The GAP- RBF algorithm was proven to have good generalization, small network size and fast training speed when the input data is uniformly distributed. When the input data is not uniformly distributed, GAP-RBF gives comparable generalization performance with a much smaller network size and much less training time.

The GAP was improved by M. Bortman and M. Aladjem in 2009 [25]. They used the Gaussian mixture model to calculate the significance of the neuron. This algorithm is called GAP-GMM algorithm. This makes it possible to employ the GAP algorithm for input data having complex and high dimension. The result showed that the GAP-GMM algorithm outperforms the original GAP achieving both lower prediction error and reduced the complexity of the trained network.

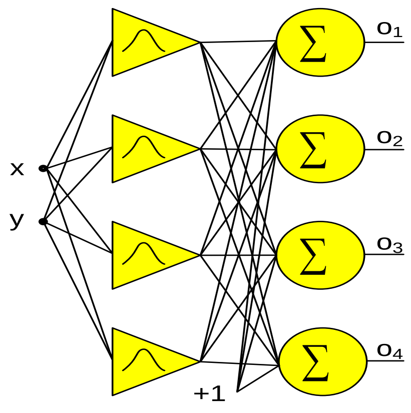
Chapter 2

Advantages and Challenges of RBF networks

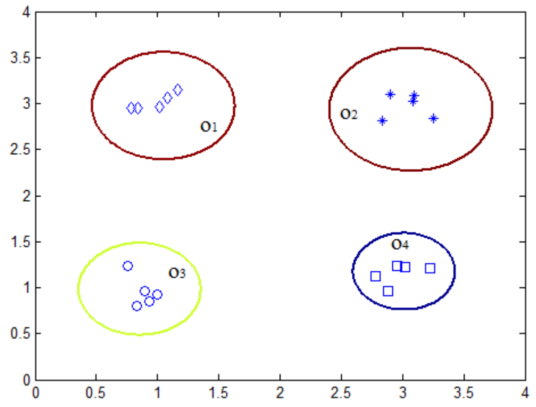
In this chapter, we will introduce the characteristics of RBF networks, the difference between RBF networks and traditional NN networks, and the advantages and disadvantages of RBF networks over traditional NNs.

2.1 Difference between traditional NN and RBF networks

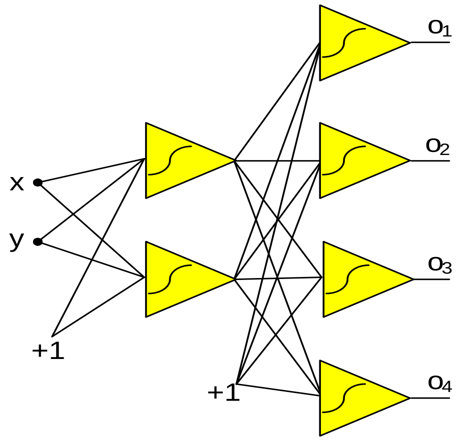
As introduced before, because of the similar layer-by-layer topology, it is often considered that RBF networks belong to MLP networks. It was proven that RBF networks can be implemented by MLP networks with increased input dimensions. Except the similarities of topologies, RBF networks and MLP networks behave very differently. First of all, RBF networks are simpler than MLP networks, which may have more than three layers architectures. Secondly, RBF networks act as local approximation networks because the network outputs are determined by specified hidden units in certain local receptive fields while MLP networks work globally, since the network outputs are decided by all the neurons. Thirdly, it is essential to set the correct initial states for RBF networks while MLP networks use randomly generated parameters initially. Last and most importantly, the mechanisms of classification for RBF networks and MLP networks are different: RBF clusters are separated by hyper spheres; while in neural networks, arbitrarily shaped hyper surfaces are used for separation. In the simple two-dimension case as shown in Fig. 2.1, the RBF network in Fig. 2.1a separates the four clusters by circles or ellipses (Fig. 2.1b) while the neural network in Fig. 2.1c does the separation by lines (Fig. 2.1d).



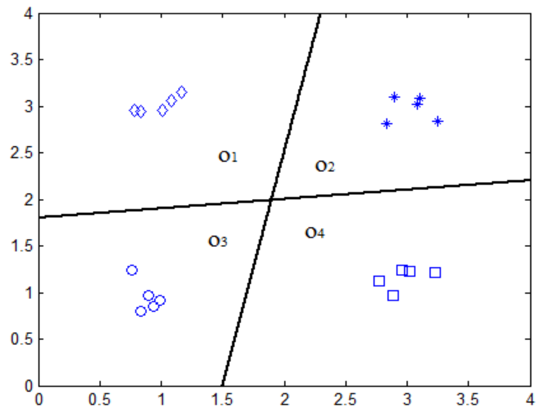
(a)



(b)



(c)



(d)

Fig. 2.1 Different classification mechanisms for pattern classification in two-dimension space: (a) RBF network; (b) Separation result of RBF network; (b) MLP network; (c) Separation result of MLP network.

2.2 Advantages and disadvantages of RBF networks over traditional NN

In this part four problems are applied respectively to RBF networks and traditional NN to test and compare the performance of traditional neural networks and RBF networks. The result is obtained from the points of architecture complexity, generalization ability which is defined to evaluate the ability of different kind of networks to successfully handle new patterns which are not used for training, and noise-tolerant ability. For traditional NN, the neuron-by-neuron (NBN) algorithm is applied for training [26]; while for RBF networks, the improved second order (ISO) method which will be described in the 3rd chapter is used for parameter updating.

The training/testing results are evaluated by the averaged sum square error calculated by:

$$E = \frac{1}{P}\frac{1}{M}\sum_{p=1}^{P}\sum_{m=1}^{M} e_{p,m}^{2} \tag{2-1}$$

where: p is the index of patterns, from 1 to P, where P is the number of patterns; m is the index of outputs, from 1 to M, where M is the number of outputs. $e_{p,m}$ is the error at output m when training pattern p, calculated as the difference between desired output and associated actual output.

The testing environment consists of: Windows 7 Professional 32-bit operating system; AMD Athlon (tm) ×2 Dual-Core QL-65 2.10GHz processor; 3.00GB (2.75GB usable) RAM; MATLAB 2007b platform.

The first experiment is called Forward kinematics which is one of practical examples well solved by neural networks. The purpose is to determine the position and orientation of the robot's end effectors when joint angles change. The figure of manipulator is shown in Fig 2.2.

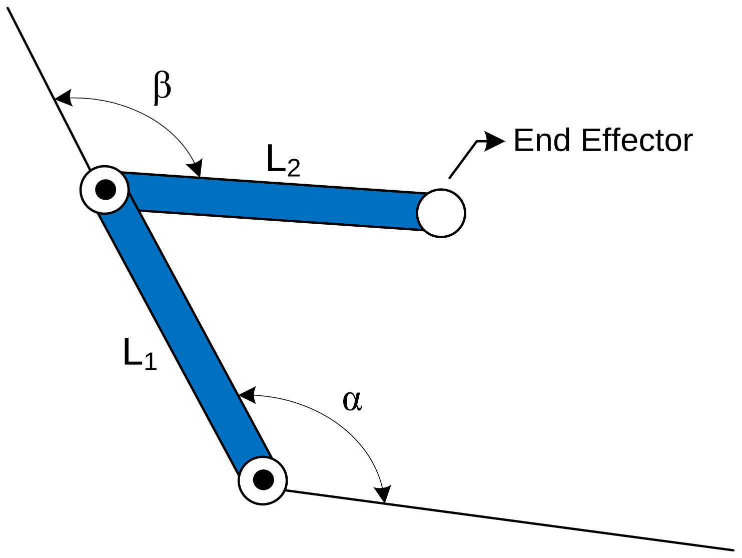


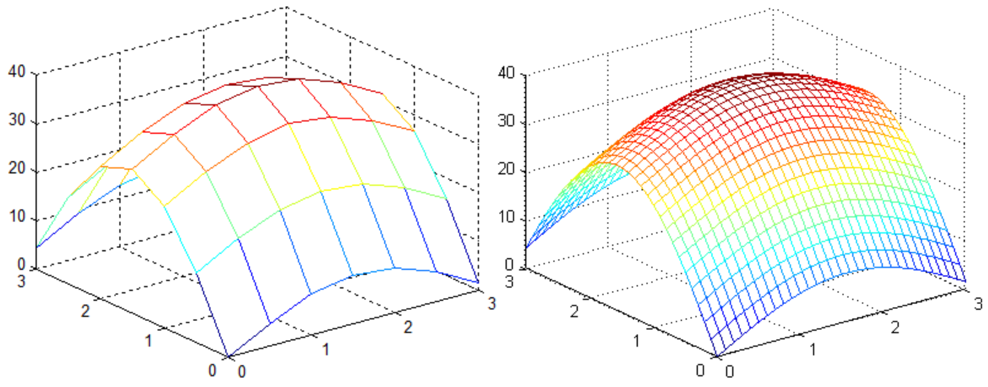
Fig 2.2 Two-link planar manipulator

As shown in Fig. 2.2, in the two-dimension space, the end effector coordinates of the manipulator is calculated by:

$$x = L_1 \cos\alpha + L_2 \cos(\alpha + \beta) \qquad (2\text{-}2)$$

$$y = L_1 \sin\alpha + L_2 \sin(\alpha + \beta) \qquad (2\text{-}3)$$

The data set of the two-dimensional forward-kinematics consist of 49 training patterns and 961 testing patterns which are generated from equations (2-2) and (2-3), with parameters α and β uniformly distributed in range [0, 3], and L_1=30, L_2=10. Figs. 2.3and 2.4 below visualizes the training/testing points in both x and y dimensions.



(a) (b)

Fig. 2.3 Data set in x-dimension: (a) 7×7=49 training patterns; (b) 31×31= 961 testing patterns

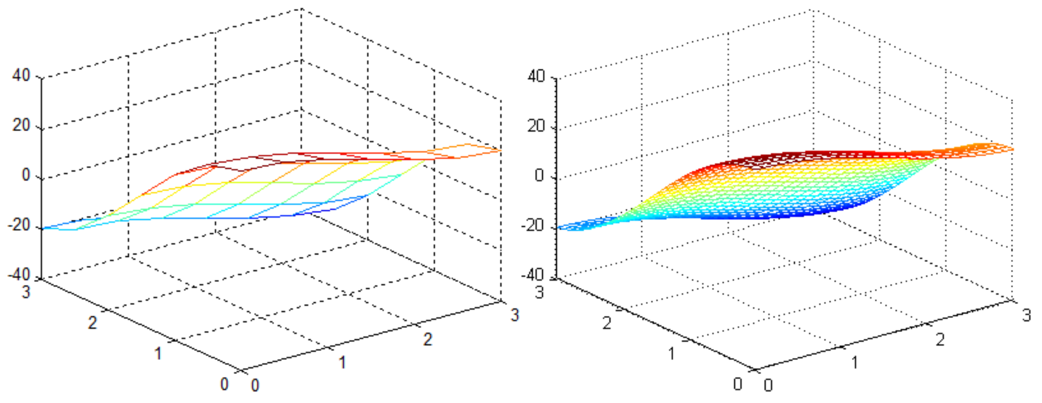


Fig. 2.4 Data set in y-dimension: (a) 7×7=49 training patterns; (b) 31×31= 961 testing patterns

For a traditional NN, all neurons are connected in FCC architectures with randomly generated initial weights between [-1, 1]. For RBF network, randomly selected patterns are used as initial centers, and the weights and widths are randomly generated between (0, 1]. For each architecture, the testing is repeated for 100 times and the averaged trajectories of training/testing errors are presented in Figs. 2.5 and 2.6 below.

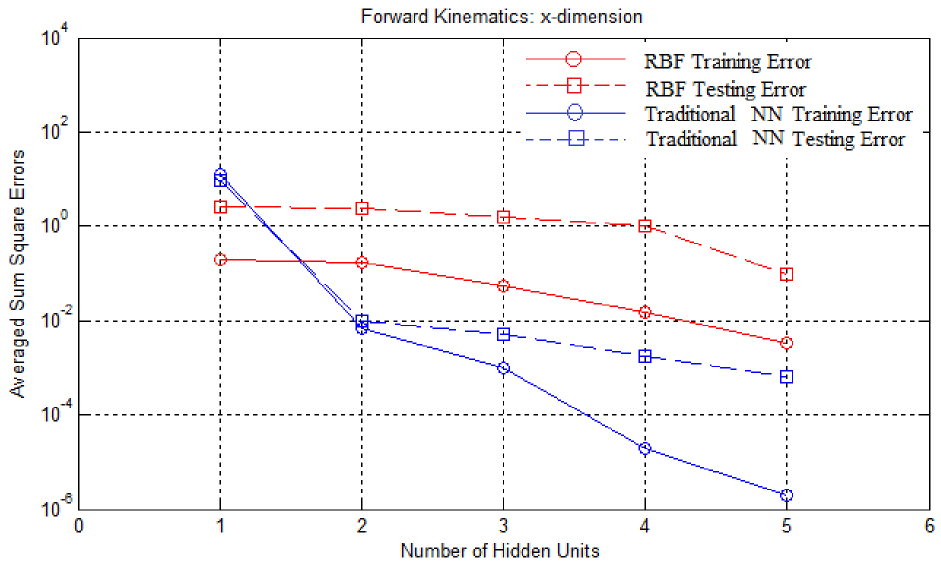


Fig. 2.5 X-dimension of forward kinematics: training/testing errors vs. the number of hidden units

As shown in Fig. 2.3, in x-dimension of kinematics, there is a big but not regular peak. Comparison results in Fig 2.5 show that RBF networks obtained smaller training/testing error than traditional neural networks at first. As the number of hidden units increase, traditional neural networks perform much better.

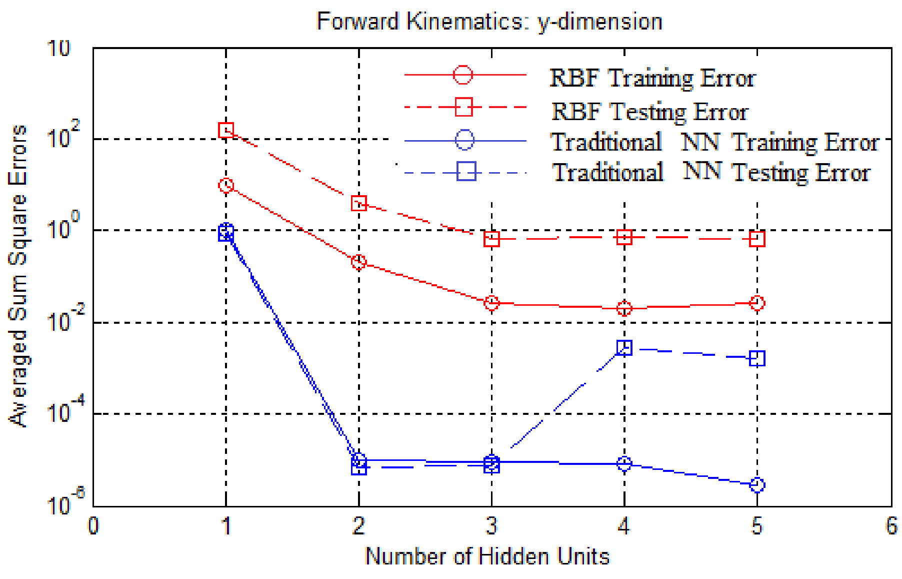


Fig. 2.6 Y-dimension of forward kinematics: training/testing errors vs. the number of hidden units

In y-dimension kinematics, there are no regular peaks and valleys (Fig. 2.4). As the comparison results shown in Fig. 2.6, traditional neural networks perform much better than RBF networks. As the number of hidden units increase, all errors decrease at first; however, the testing errors of trained traditional neural networks increase due to the over-fitting problem [27].

The second problem is a peaks function approximation problem, in which 20×20=400 points (Fig. 2.7a) are applied as a training set in order to predict the values of 100×100=10,000 points (Fig. 2.7b) in the same range. The surface is generated by MATLAB function peaks, and all training/testing points are uniformly distributed.

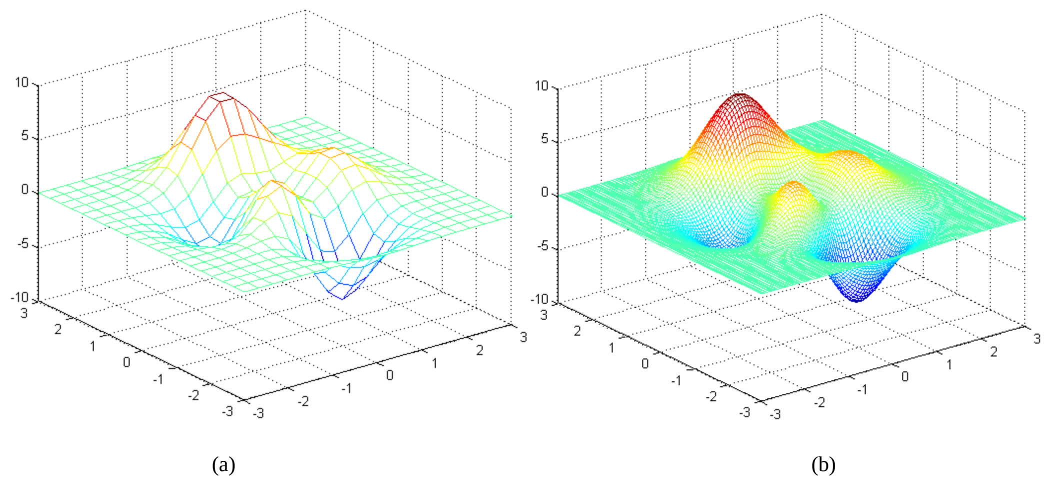


Fig.2.7 Peaks function approximation problem: (a) training data, 20×20=400 points; (b) testing data, 100×100=10,000 points.

Using RBF networks for approximating the peaks surface, since there are three peaks and two valleys, at least 5 hidden units are required.

For traditional neural networks, FCC architectures are applied for training. Table 1 presents the experimental results. One may notice that, with the same 5 hidden units, feedforward neural network got more than 2 times larger training errors and more than two orders of magnitude larger testing errors than radial basis function network.

Table 1 Comparison results Of Radial RBF Networks And Traditional Neural Networks On Peaks Surface Approximation Problem

Architectures	***Training Errors***	***Testing Errors***
RBF with 5 hidden units	0.0111	0.0120
FCC with 4 hidden units	0.1361	1.3706
FCC with 5 hidden units	0.0294	1.1834
FCC with 6 hidden units	0.0040	1.1689
FCC with 7 hidden units	0.0018	1.1713
FCC with 8 hidden units	0.0010	1.1728

Fig.2.8 below shows the generalization results of two types of neural networks, both of which have 5 hidden units. It can be seen that RBF has a better generalization ability than traditional NNs.

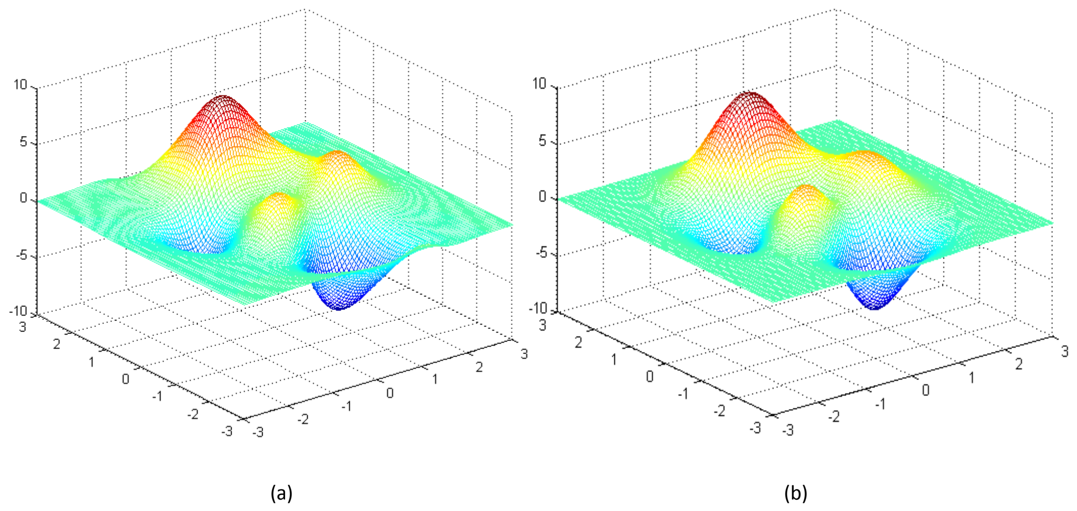


Fig. 2.8 Generalization results of neural networks with 5 hidden units: (a) traditional neural networks, testing error=1.1834; (b) RBF networks, testing error=0.0120

The third problem is the famous two-spiral problem which is always considered as a very complex benchmark to evaluate the power and efficiency of training algorithms and network architectures. As shown in Fig 2.9, the purpose of the two-spiral problem is to generate the 94 twisted two-dimension points into two groups, marked as +1(blue circles) and -1(red stars).

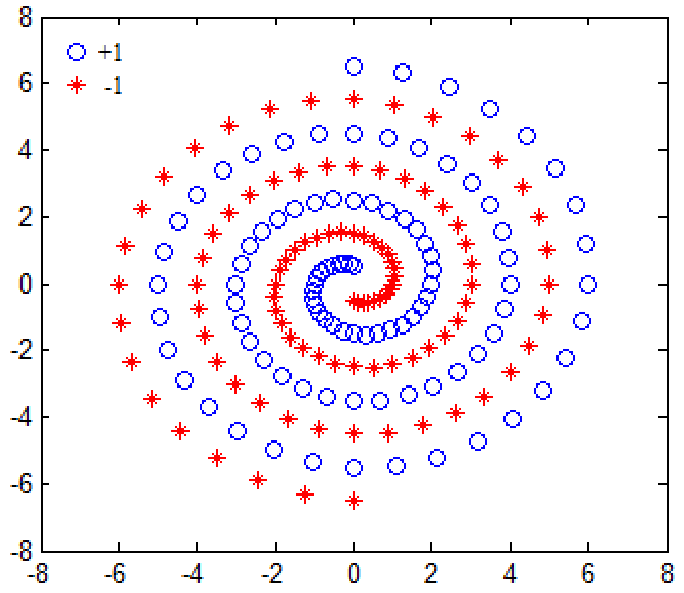


Fig.2.9 Two-spiral classification problem

Using traditional neural networks, the two-spiral problem can be solved very efficiently and the minimum number of required hidden units depends on network architectures. For example, using standard MLP architecture with one hidden layer, at least 33 hidden units are required for successful training. For MLP architecture with two hidden layers (assume they have the same number of neurons), at least 14 hidden units are required for convergence [28]. The most efficient architecture, FCC networks, can solve two-spiral problem with only 7 hidden units [28]. Fig. 2.10 shows the generalization results of 13 hidden units in FCC networks.

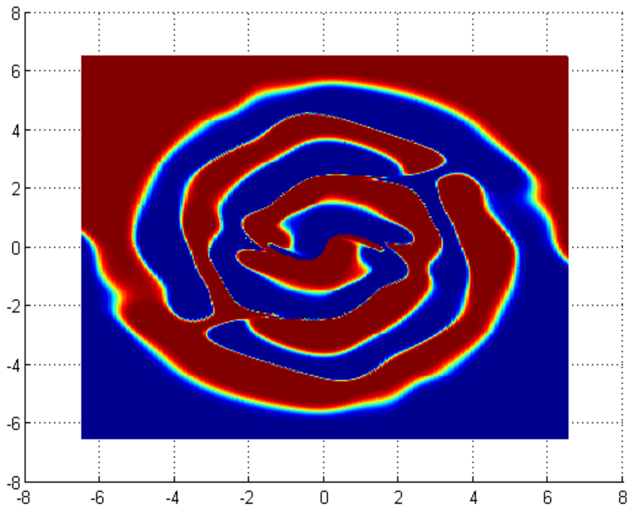


Fig. 2.10 Generalization result of FCC architecture with 13 hidden units

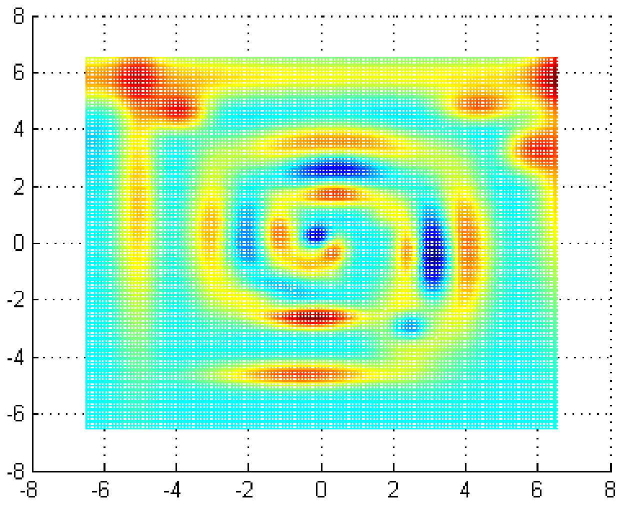


Fig. 2.11 Generalization result of RBF network with 40 hidden units

Using RBF networks, in order to reach the similar training error with the FCC architecture with 7 hidden units, at least 40 hidden units are required. The generalization result is shown in

Fig. 2.11.

For the two-spiral classification problem, one may notice that in order to get the similar classification results the RBF networks need many more number of hidden units than traditional NNs.

The fourth problem is a character image recognition problem. As shown in Fig 2.12, for each column, there are 10 character images from “A” to “J”, each of which consists of 8×7=56 pixels with normalized Jet degree between -1 and 1 (-1 for blue and 1 for red). The first column is the original image data without noise and used as training patterns; while the other 7 columns, from the 2nd column to the 8th column, are noised and used as testing patterns. The strength of noise is calculated by:

$$NP_i = P_0 + i \times \delta \tag{2-4}$$

where: P_0 is the original image in 1st column; NP_i is the image data with i-th level noise; i is the noise level from 1 to 7; δ is the randomly generated noise between [-0.5, 0.5].

The aim is to build neural networks based on the training patterns (1st column) and then test the networks with noised input data (from 2nd column to 8th column). For each noise level, the testing will be repeated for 100 times with randomly generated noise.

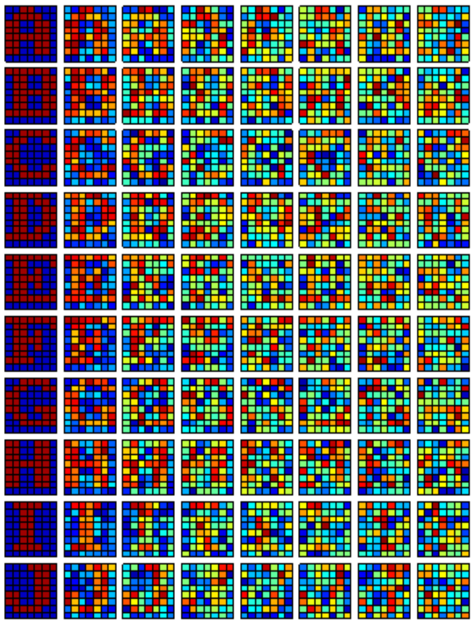


Fig 2.12 Character images with different noise levels from 0 to 7 in left-to-right order (one data set in 100 groups)

Using traditional neural networks, the MLP architecture 56-10 is applied for training. The testing results on the trained network are presented in Table 2 below. One may notice that recognition errors appear when patterns with 2nd level noises are applied.

Table 2 Success Rates Of The Trained Traditional Neural Network For Character Image Recognition

Data* *Char	***Noise level 1***	***Noise level 2***	***Noise level 3***	***Noise level 4***	***Noise level 5***	***Noise level 6***	***Noise level 7***
"A"	100%	99%	90%	70%	59%	44%	39%
"B"	100%	100%	100%	95%	94%	84%	81%
"C"	100%	98%	81%	52%	51%	45%	41%
"D"	100%	97%	84%	64%	56%	34%	33%
"E"	100%	100%	92%	70%	52%	48%	42%
"F"	100%	95%	83%	71%	49%	36%	35%
"G"	100%	96%	72%	58%	53%	34%	32%
"H"	100%	94%	60%	50%	32%	32%	23%
"I"	100%	100%	100%	95%	83%	76%	71%
"J"	100%	100%	96%	81%	70%	54%	46%

For RBF networks, 10 hidden units are chosen, and their centers are corresponding to 10 characters, respectively. Applying the testing patterns, the performance of the trained RBF network is shown in Table 3 below. One may notice that recognition errors appear until 3rd level noised patterns are applied.

Table 3 Success Rate Of The Trained RBF Network For Character Image Recognition

Data* / *Char	***Noise level 1***	***Noise level 2***	***Noise level 3***	***Noise level 4***	***Noise level 5***	***Noise level 6***	***Noise level 7***
“A”	100%	100%	100%	100%	100%	97%	97%
“B”	100%	100%	100%	99%	97%	96%	87%
“C”	100%	100%	99%	98%	90%	88%	80%
“D”	100%	100%	100%	98%	98%	95%	88%
“E”	100%	100%	100%	95%	94%	76%	76%
“F”	100%	100%	100%	97%	92%	83%	79%
“G”	100%	100%	99%	96%	88%	81%	77%
“H”	100%	100%	100%	100%	100%	98%	91%
“I”	100%	100%	100%	100%	100%	100%	97%
“J”	100%	100%	100%	100%	100%	99%	95%

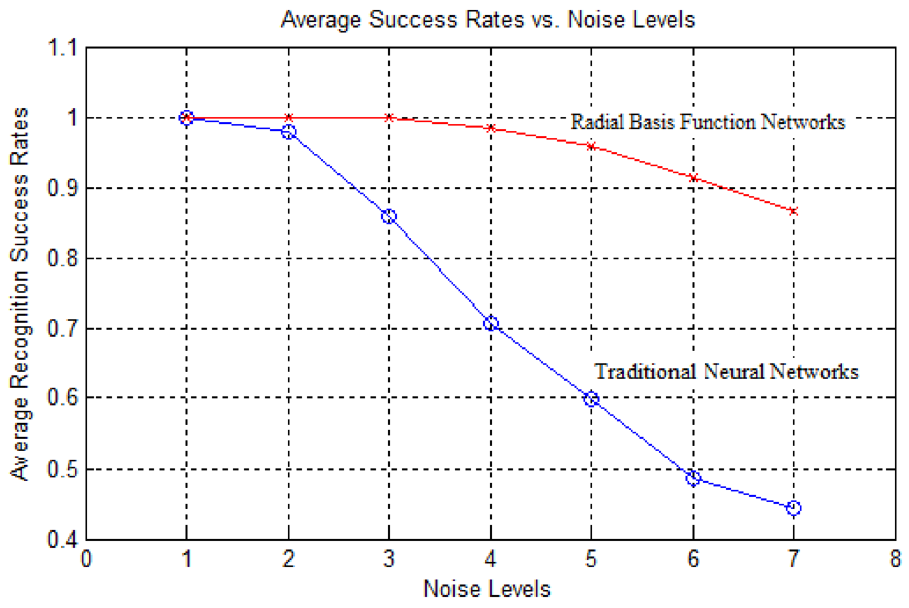


Fig.2.13 Average recognition success rates of traditional neural networks and RBF networks under different levels of noised inputs

Fig.2.13 shows the average success rate of two types of neural network architectures. It can be seen that RBF networks (red line) are more robust and have a better tolerant ability to input noises than traditional neural networks (blue line).

2.3 Conclusions

Comparing the results of the four examples and the properties of two types of neural networks, it can be concluded that:

(1) For function approximation problems, RBF networks are specially recommended for surface with regular peaks and valleys since efficient and accurate design can be obtained. RBF networks also have better generalization ability than traditional NNs. For surfaces without regular peaks and valleys, traditional neural networks are preferred as a general model.

(2) The RBF networks need larger size hidden units to solve the two spiral classification problems than traditional NNs. However, compared to traditional NNs, its structure is fixed and easier to design.

(3) For trained networks, RBF networks perform more robustly and tolerantly than traditional neural networks when dealing with noised input data set.

Chapter 3

Second Order Training of RBF networks

In original, people use unsupervised training process to construct the RBF networks [29]. But this is not proper for solving practical problems where there are usually hundreds of training patterns. In order to achieve more accuracy with less RBF units, the supervised training process was introduced for parameters adjusting. Based on gradient decent concept, lots of methods have developed to perform "deeper" training on RBF networks. Besides output weights, more parameters, such as centers and widths, are adjusted during the learning process. But first order gradient methods have very limited search ability and take a long time for convergence. In this chapter, we will propose an advanced training algorithm of RBF networks which is called second order gradient method. This method is derived from neuron-by-neuron (NBN) algorithm [26] and improved Levenberg Marquardt algorithm [30] used for traditional neural network training. In the proposed approach, all the parameters (as shown in Fig. 1), such as input weights, output weights, centers and widths, are adjusted by second order update rule. Furthermore, the proposed algorithm does not suffer from huge Jacobian matrix storage and its side effects, when training data is huge. With the proposed training algorithm, RBF networks can be designed very compactly; at the same time, the network performances, such as training speed and approximation accuracy, are improved.

Before presenting the computation fundamentals of radial basis function networks and Levenberg Marquardt algorithm, let us introduce the commonly used indices in this paper:

- i is the index of inputs, from 1 to I, where I is the number of input dimensions.
- p is the index of patterns, from 1 to P, where P is the number of input patterns.
- h is the index of hidden units, from 1 to H, where H is the number of units in the hidden layer.
- m is the index of outputs, from 1 to M, where M is the number of outputs.
- k is the index of iterations.

Other indices will be interpreted in related places.

3.1 Levenberg-Marquardt algorithm derivation

The Levenberg-Marquardt algorithm was first published by Kenneth Levenberg and then rediscovered by Donald Marquardt [31]. It is a blend of Gradient descent and Gauss-Newton iteration. We will introduce them respectively.

Gradient descent is the simplest, most intuitive technique to find minima in a function. Parameter updation is performed by adding the negative of the scaled gradient at each step:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha\, \mathbf{g}_k \tag{3-1}$$

Where w is the parameter to be adjusted, α is the learning constant gradient, $\boldsymbol{g}$ is defined as the first order derivative of total error function:

$$\boldsymbol{g} = \frac{\partial E(\boldsymbol{x}, \boldsymbol{w})}{\partial \boldsymbol{w}} = \begin{bmatrix} \frac{\partial E}{\partial w_1} & \frac{\partial E}{\partial w_2} & \cdots & \frac{\partial E}{\partial w_N} \end{bmatrix}^T \tag{3-2}$$

where E is the error. Simple gradient descent suffers from various convergence problems. Logically, we would like to take large steps down the gradient at locations where the gradient is small and conversely, take small steps when the gradient is large, so as not to rattle out of the minima. With the above update rule, we do the opposite way. For example, if there is a long and narrow valley in the error surface, it might oscillate and never be convergent because the gradient of the valley wall is so large and the error takes such large step concurrently that it will never reach the base of valley.

This situation can be improved upon by using curvature as well as gradient information, namely second derivatives. One way to do this is to use Newton's method [32]. Expanding the gradient vector $\boldsymbol{g}$ using Taylor series around the current state and take the first order approximation, we get:

$$\begin{cases} g_1 \approx g_{1,0} + \frac{\partial g_1}{\partial w_1}\Delta w_1 + \frac{\partial g_1}{\partial w_2}\Delta w_2 + \cdots + \frac{\partial g_1}{\partial w_N}\Delta w_N \\ g_2 \approx g_{2,0} + \frac{\partial g_2}{\partial w_1}\Delta w_1 + \frac{\partial g_2}{\partial w_2}\Delta w_2 + \cdots + \frac{\partial g_2}{\partial w_N}\Delta w_N \\ \cdots \\ g_N \approx g_{N,0} + \frac{\partial g_N}{\partial w_1}\Delta w_1 + \frac{\partial g_N}{\partial w_2}\Delta w_2 + \cdots + \frac{\partial g_N}{\partial w_N}\Delta w_N \end{cases} \tag{3-3}$$

If we set left hand of the function to be 0, we get the update rule for Newton's method:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \boldsymbol{H}_k^{-1}\boldsymbol{g}_k \tag{3-4}$$

where ***H*** is called Hessian matrix that is defined as:

$$\boldsymbol{H} = \begin{bmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_1 \partial w_N} \\ \frac{\partial^2 E}{\partial w_2 \partial w_1} & \frac{\partial^2 E}{\partial w_2^2} & \cdots & \frac{\partial^2 E}{\partial w_2 \partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 E}{\partial w_N \partial w_1} & \frac{\partial^2 E}{\partial w_N \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_N^2} \end{bmatrix} \tag{3-5}$$

The main advantage of this technique is rapid convergence. However, the rate of convergence is sensitive to the starting location. The calculation of Hessian matrix is very complicate. In order to simplify the process, Jacobian matrix ***J*** is introduced as:

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w_1} & \frac{\partial e_{1,1}}{\partial w_2} & \cdots & \frac{\partial e_{1,1}}{\partial w_N} \\ \frac{\partial e_{1,2}}{\partial w_1} & \frac{\partial e_{1,2}}{\partial w_2} & \cdots & \frac{\partial e_{1,2}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{1,M}}{\partial w_1} & \frac{\partial e_{1,M}}{\partial w_2} & \cdots & \frac{\partial e_{1,M}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{P,1}}{\partial w_1} & \frac{\partial e_{P,1}}{\partial w_2} & \cdots & \frac{\partial e_{P,1}}{\partial w_N} \\ \frac{\partial e_{P,2}}{\partial w_1} & \frac{\partial e_{P,2}}{\partial w_2} & \cdots & \frac{\partial e_{P,2}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{P,M}}{\partial w_1} & \frac{\partial e_{P,M}}{\partial w_2} & \cdots & \frac{\partial e_{P,M}}{\partial w_N} \end{bmatrix} \tag{3-6}$$

The relationship between Hessian matrix H and Jacobian matrix J is:

$$\boldsymbol{H} \approx \boldsymbol{J}^T \boldsymbol{J} \tag{3-7}$$

So function 3-4 can be rewritten as:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \left(\mathbf{J}_k^T \mathbf{J}_k\right)^{-1} \mathbf{J}_k^T \mathbf{e}_k \tag{3-8}$$

This is so called Gauss-Newton algorithm.

It can be seen that simple gradient descent and Gauss-Newton iteration are complementary in the advantages they provide. Levenberg proposed an algorithm based on this observation, whose update rule is a blend of the above and is given as:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \left(\mathbf{J}_k^T \mathbf{J}_k + \mu \mathbf{I}\right)^{-1} \mathbf{J}_k^T \mathbf{e}_k \tag{3-9}$$

where μ is always positive, called combination coefficient and $\boldsymbol{I}$ is the identity matrix.

This update rule is used as follows: if the error goes down following an update, it implies that our quadratic assumption on $\boldsymbol{g}$ is working and we reduce μ to reduce the influence of gradient descent. On the other hand, if the error goes up, we would like to follow the gradient more and so μ is increased by the same factor.

3.2 Improved Second Order (ISO) algorithm

3.2.1 The formulas of ISO algorithm in RBF networks

Following the computation procedure in LM algorithm, the update rule of RBF networks can be written as:

$$\Delta_{k+1} = \Delta_k - \left(J_k^T J_k + \mu_k I\right)^{-1} J_k^T e_k \tag{3-10}$$

where $\varDelta$ is the variable vector.

Function 3-10 can be replaced by:

$$\Delta_{k+1} = \Delta_k - \left(Q_k + \mu_k I\right)^{-1} g_k \tag{3-11}$$

Where: quasi Hessian matrix Q is directly calculated as the sum of $P\times M$ sub matrices $q_{p,m}$:

$$\mathbf{Q} = \sum_{p=1}^{P}\sum_{m=1}^{M}\mathbf{q}_{p,m} \qquad \mathbf{q}_{p,m} = \mathbf{j}_{p,m}^T \mathbf{j}_{p,m} \tag{3-12}$$

and gradient vector g is calculated as the sum of $P\times M$ sub vectors $\eta_{p,m}$:

$$\mathbf{g} = \sum_{p=1}^{P}\sum_{m=1}^{M}\boldsymbol{\eta}_{p,m} \qquad \boldsymbol{\eta}_{p,m} = \mathbf{j}_{p,m}^T e_{p,m} \tag{3-13}$$

Where: vector $j_{p,m}$ is one row of Jacobian matrix for pattern p associated with output m calculated by

$$\mathbf{j}_{p,m} = \left[\frac{\partial e_{p,m}}{\partial \Delta_1}, \frac{\partial e_{p,m}}{\partial \Delta_2} \cdots \frac{\partial e_{p,m}}{\partial \Delta_n} \cdots \frac{\partial e_{p,m}}{\partial \Delta_N}\right] \tag{3-14}$$

and the error $e_{p,m}$ is given by:

$$e_{p,m} = y_{p,m} - o_{p,m} \tag{3-15}$$

In the proposed algorithm, there are four types of variables: output weight matrix w, width vector σ, input weight matrix u and center matrix c. Therefore, the Jacobian row in (3-14) consists of four parts:

$$\mathbf{j}_{p,m} = \left[\cdots \frac{\partial e_{p,m}}{\partial w_{h,m}} \cdots \frac{\partial e_{p,m}}{\partial \sigma_h} \cdots \frac{\partial e_{p,m}}{\partial u_{i,h}} \cdots \frac{\partial e_{p,m}}{\partial c_{h,i}} \cdots \right] \tag{3-16}$$

3.2.2 **Computation of $\partial e_{p,m}/\partial w_{h,m}$**

The output weight matrix *w* presents the weight values on the connections between hidden layer and output layer, also including the bias weights on output units. So the output weight matrix *w* has $(H+1)\times M$ elements.

Using (3-15), the Jacobian element $\partial e_{p,m}/\partial w_{h,m}$ is calculated as:

$$\frac{\partial e_{p,m}}{\partial w_{h,m}} = -\frac{\partial o_{p,m}}{\partial w_{h,m}} \tag{3-17}$$

By combining with (1-5), equation (3-17) is rewritten as:

$$\frac{\partial e_{p,m}}{\partial w_{h,m}} = -\varphi_h(\mathbf{x}_p) \tag{3-18}$$

For bias weight $w_{0,m}$, related Jacobian element is calculated by

$$\frac{\partial e_{p,m}}{\partial w_{0,m}} = -1 \tag{3-19}$$

The width vector σ consists of the width of each RBF unit, so the total number of elements is *H*. For the RBF unit h, using (3-15) and the differential chain rule, the Jacobian element $\partial e_{p,m}/\partial \sigma_h$ is calculated as:

$$\frac{\partial e_{p,m}}{\partial \sigma_h} = -\frac{\partial o_{p,m}}{\partial \sigma_h} = -\frac{\partial o_{p,m}}{\partial \varphi_h(\mathbf{x}_p)} \frac{\partial \varphi_h(\mathbf{x}_p)}{\partial \sigma_h} \tag{3-20}$$

By combining with equations (1-4) and (1-5), (3-20) is rewritten as

$$\frac{\partial e_{p,m}}{\partial \sigma_h} = -\frac{w_{h,m}\varphi_h(\mathbf{x}_p)\|\mathbf{y}_{p,h} - \mathbf{c}_h\|^2}{\sigma_h^2} \tag{3-21}$$

where: the vector y$_{p,h}$ is defined as:

$$y_{p,h,i} = x_{p,i}u_{i,h} \tag{3-22}$$

The input weight matrix *u* describes the weights on the connections between input layer and the hidden layer, so the total number of elements is *I*×*H*.

Using (1-5) and the differential chain rule, the Jacobian element $\partial e_{p,m}/\partial u_{i,h}$ is calculated as

$$\frac{\partial e_{p,m}}{\partial u_{i,h}} = -\frac{\partial o_{p,m}}{\partial u_{i,h}} = -\frac{\partial o_{p,m}}{\partial \varphi_h(\mathbf{x}_p)}\frac{\partial \varphi_h(\mathbf{x}_p)}{\partial u_{i,h}} \tag{3-23}$$

By combing with equations (1-3), (1-4) and (1-5), (3-23) is rewritten as

$$\frac{\partial e_{p,m}}{\partial u_{i,h}} = \frac{2w_{h,m}\varphi_h(\mathbf{x}_p)x_{p,i}(x_{p,i}u_{i,h} - c_{h,i})}{\sigma_h} \tag{3-24}$$

The center matrix *c* consists of the centers of RBF units, and the number of elements is *H*×*I*. For RBF unit *h*, using (3-15) and the differential chain rule, the Jacobian element $\partial e_{p,m}/\partial c_{h,i}$ is calculated by:

$$\frac{\partial e_{p,m}}{\partial c_{h,i}} = -\frac{\partial o_{p,m}}{\partial c_{h,i}} = -\frac{\partial o_{p,m}}{\partial \varphi_h(\mathbf{x}_p)}\frac{\partial \varphi_h(\mathbf{x}_p)}{\partial c_{h,i}} \tag{3-25}$$

By combining with equations (1-3), (1-4) and (1-5), (3-25) is rewritten as:

$$\frac{\partial e_{p,m}}{\partial c_{h,i}} = -\frac{2w_{h,m}\varphi_h(\mathbf{x}_p)(x_{p,i}u_{i,h} - c_{h,i})}{\sigma_h} \tag{3-26}$$

With equations (3-18), (3-19), (3-21), (3-24) and (3-26), all the Jacobian row elements for output *m* when applying pattern *p* can be obtained. Then related sub quasi Hessian matrix $q_{p,m}$ and sub gradient vector $\eta_{p.m}$ can be computed by (3-12) and (3-13), respectively, so do the quasi Hessian matrix and gradient vector. Notice that, all patterns are independent, so the related memory for $j_{p,m}$, $q_{p,m}$ and $\eta_{p.m}$ can be reused.

3.3 Implementation of the ISO algorithm

In order to explain the training process of RBF networks using the proposed algorithm, let us use the parity-2 (XOR) classification problem as an illustration vehicle.

Fig. 3.1 shows the data set of XOR problem. The goal is to classify the 4 points (-1,-1), (-1,1), (1,-1) and (1,1) into two groups, which are marked by +1 and -1. Fig. 3.2 shows the minimum radial basis function network for solving the parity-2 problem.

In		out
-1	-1	+1
-1	+1	-1
+1	-1	-1
+1	+1	+1

Fig. 3.1 Data set of XOR problem

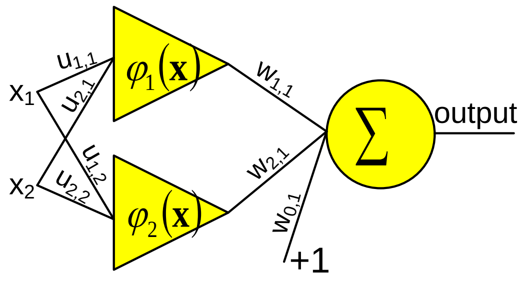


Fig. 3.2 RBF network for solving XOR problem

Implementing the proposed algorithm on the example, the training procedure can be organized in the following steps:

1. Initialization

As shown in Fig. 3.2, the initial conditions are set as: output weights $\boldsymbol{w}$=[$w_{0,1}$, $w_{1,1}$, $w_{2,1}$], widths of two RBF units $\boldsymbol{\sigma}$=[σ_1, σ_2], input weights $\boldsymbol{u}$=[$u_{1,1}$, $u_{2,1}$; $u_{1,2}$, $u_{2,2}$], and centers of two RBF units $\boldsymbol{c}$=[$c_{1,1}$, $c_{1,2}$; $c_{2,1}$, $c_{2,2}$].

In order to apply the update rule in (3-11), the variable vector $\boldsymbol{\Delta}_1$ is built by reforming the current parameters $\boldsymbol{w}$, $\boldsymbol{\sigma}$, $\boldsymbol{u}$ and $\boldsymbol{c}$ as: $\boldsymbol{\Delta}_1$=[$w_{0,1}$, $w_{1,1}$, $w_{2,1}$, σ_1, σ_2, $u_{1,1}$, $u_{2,1}$, $u_{1,2}$, $u_{2,2}$, $c_{1,1}$, $c_{1,2}$, $c_{2,1}$, $c_{2,2}$].

2. Error Evaluation

The root mean square error E is defined to evaluate the training procedure:

$$E = \sqrt{\frac{\sum_{p=1}^{P}\sum_{m=1}^{M} e_{p,m}^2}{P \times M}} \tag{3-27}$$

Where: *P* is the number of patterns and *M* is the number of outputs.

Applying the first pattern [-1, -1, 1] to RBF unit 1, the vector multiplication of x1=[-1, -1] and u1=[u_1,1, u_2,1] is obtained using (1-3)

$$\mathbf{y}_{1,1} = [-u_{1,1}, -u_{2,1}] \tag{3-28}$$

With (3-28), the Euclidean Norm of vector $\boldsymbol{y}_{1,1}$ and vector $\boldsymbol{c}_1$ is calculated as:

$$\left\|\mathbf{y}_{1,1} - \mathbf{c}_1\right\|^2 = \left(-u_{1,1} - c_{1,1}\right)^2 + \left(-u_{2,1} - c_{1,2}\right)^2 \tag{3-29}$$

Using (1-4) and (3-29), the output of the RBF unit 1 is calculated by:

$$\varphi_1(\mathbf{x}_1) = \exp\left(-\frac{\left\|\mathbf{y}_{1,1} - \mathbf{c}_1\right\|^2}{\sigma_1}\right) \tag{3-30}$$

Then by applying pattern [*-1, -1, 1*] to RBF unit 2, using similar computation with RBF unit 1, $\boldsymbol{y}_{1,2}$ and $\varphi_2(\boldsymbol{x}_1)$ are calculated.

Following the computation in equation (1-5), the network output is calculated as

$$o_{1,1} = w_{0,1} + w_{1,1}\varphi_1(\mathbf{x}_1) + w_{2,1}\varphi_2(\mathbf{x}_1) \tag{3-31}$$

Using equations (3-15) and (3-31), the error $e_{1,1}$ for the first pattern is computed by

$$e_{1,1} = 1 - o_{1,1} \tag{3-32}$$

Repeating the computation from (3-28) to (3-32) for other three patterns, the errors $e_{2,1}$, $e_{3,1}$ and $e_{4,1}$ are all obtained, then the root mean square error defined in (3-27) is calculated as:

$$E_1 = \frac{1}{2}\sqrt{e_{1,1}^2 + e_{2,1}^2 + e_{3,1}^2 + e_{4,1}^2} \tag{3-33}$$

3. Computation of quasi Hessian matrix and gradient vector

First of all, the quasi Hessian matrix ***Q₁*** and gradient vector ***g₁*** are initialized as zero:

$$\mathbf{Q}_1 = \mathbf{0}, \quad \mathbf{g}_1 = \mathbf{0} \tag{3-34}$$

Applying the first pattern [*-1, -1, 1*] and going through the computation from equations (3-28) to (3-32), parameters $\varphi_1(\mathbf{x}_1)$, $\varphi_2(\mathbf{x}_1)$, $o_{1,1}$ *and* $e_{1,1}$ are all obtained.

Using equations (3-18), (3-19), (3-21), (3-24) and (3-26), all the elements of Jacobian row for the first pattern can be calculated and built in the format of (3-16):

$\boldsymbol{j}_{1,1}$=[$\partial e_{1,1}/\partial w_{0,1}$, $\partial e_{1,1}/\partial w_{1,1}$, $\partial e_{1,1}/\partial w_{2,1}$, $\partial e_{1,1}/\partial \sigma_1$, $\partial e_{1,1}/\partial \sigma_2$, $\partial e_{1,1}/\partial u_{1,1}$, $\partial e_{1,1}/\partial u_{2,1}$, $\partial e_{1,1}/\partial u_{1,2}$, $\partial e_{1,1}/\partial u_{2,2}$, $\partial e_{1,1}/\partial c_{1,1}$, $\partial e_{1,1}/\partial c_{1,2}$, $\partial e_{1,1}/\partial c_{2,1}$, $\partial e_{1,1}/\partial c_{2,2}$].

Using (3-12), the sub quasi Hessian matrix $\boldsymbol{q}_{1,1}$ is calculated to update the quasi Hessian matrix $\boldsymbol{Q}_1$

$$\mathbf{q}_{1,1} = \mathbf{j}_{1,1}^T \mathbf{j}_{1,1} \qquad \mathbf{Q}_1 = \mathbf{Q}_1 + \mathbf{q}_{1,1} \tag{3-35}$$

Using (3-13), the sub gradient vector $\boldsymbol{\eta}_{1,1}$ calculated to update the gradient vector $\boldsymbol{g}_1$

$$\boldsymbol{\eta}_{1,1} = \mathbf{j}_{1,1}^T e_{1,1} \qquad \mathbf{g}_1 = \mathbf{g}_1 + \boldsymbol{\eta}_{1,1} \tag{3-36}$$

By repeating the computation (3-35) and (3-36) for the other three patterns, the accumulated results of matrix ***Q₁*** and vector ***g₁*** are the required quasi Hessian matrix and gradient vector.

4. Parameters Update

After the computation of quasi Hessian matrix $\boldsymbol{Q_1}$ and gradient vector $\boldsymbol{g_1}$, using (3-11), the updated parameter vector $\boldsymbol{\Delta_2}$ can be calculated as:

$$\mathbf{\Delta}_2^T = \mathbf{\Delta}_1^T + \left(\mathbf{Q}_1 + \mu_1 \mathbf{I}\right)^{-1} \mathbf{g}_1 \qquad (3\text{-}37)$$

From the new parameter vector $\boldsymbol{\Delta_2}$, the parameters $\boldsymbol{w}$, $\boldsymbol{\sigma}$, $\boldsymbol{u}$ and $\boldsymbol{c}$ can be extracted according with the order of constructing the parameter vector $\boldsymbol{\Delta_1}$ as was done previously.

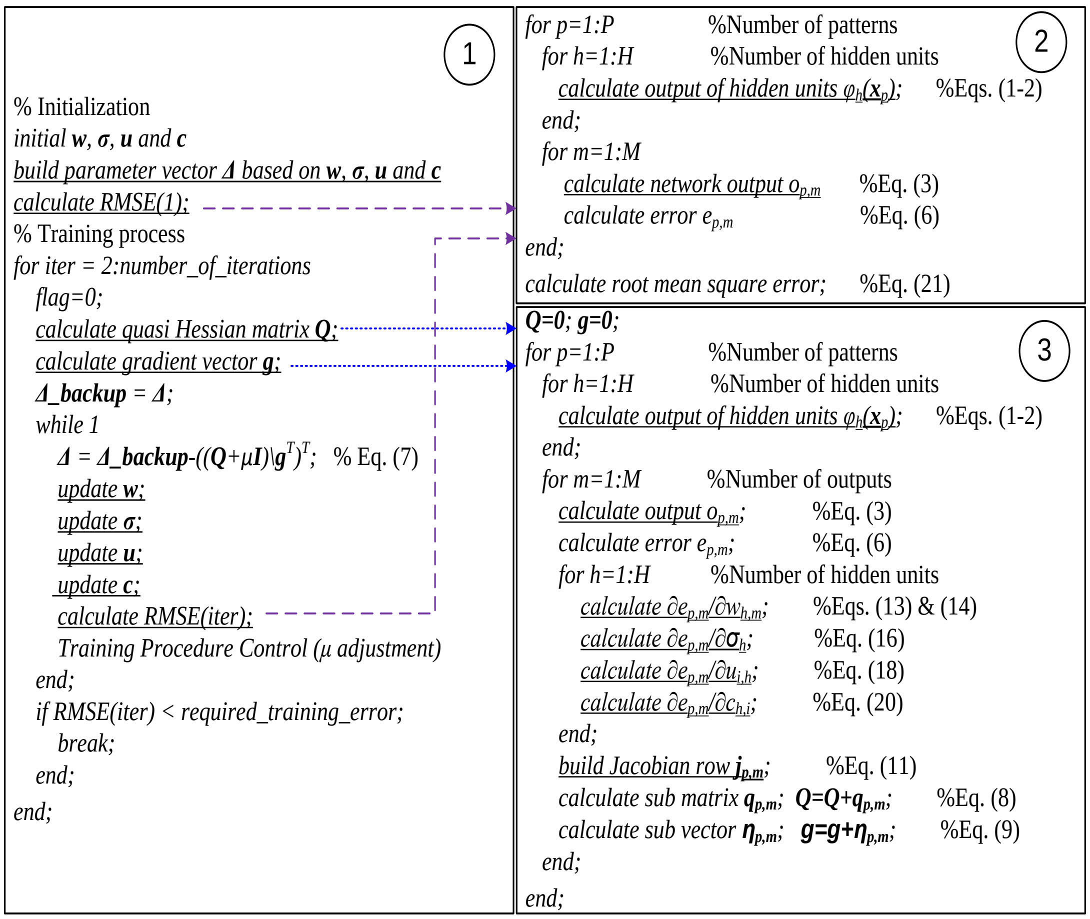
5. Training Procedure Control

With the updated parameters $\boldsymbol{w}$, $\boldsymbol{\sigma}$, $\boldsymbol{u}$ and $\boldsymbol{c}$, new root mean square error E_2 can be evaluated by following the procedure described above. Then the training process is controlled by the rules:

- If E_2 is less than the setting value, training converges.
- If $E_1 \geq E_2$, reduce parameter u_1 and keep the current parameter values (used for E_2 calculation). Then go through the above procedures for next iteration.
- If $E_1 < E_2$, increase parameter u_1 and recover previous parameter values (used for E_1 calculation). Then go through the above procedures. A counter (variable *flag* in Fig. 3.3) should be added here to help avoid dead loop.

Fig. 3.3 shows the pseudo code of the proposed algorithm with links to the equations given in previous sections. The block (1) in Fig. 3.3 is the main procedure for weight updating; the block (2) evaluates the training process according to root mean square error and the block (3) performs quasi Hessian matrix and gradient vector computation. Normally, the μ parameter in the proposed algorithm is initialed as 0.01.

One may notice that the proposed ISO algorithm for RBF network training takes the advantages of the LM algorithm for neural network training. However, because of the different activation functions, network architectures and parameters of the two network models, the Jacobian row computation and parameter update are very different as shown in Fig.3.3.



```
1
% Initialization
initial $\mathbf{w}$, $\boldsymbol{\sigma}$, $\mathbf{u}$ and $\mathbf{c}$
build parameter vector $\boldsymbol{\Delta}$ based on $\mathbf{w}$, $\boldsymbol{\sigma}$, $\mathbf{u}$ and $\mathbf{c}$
calculate RMSE(1);
% Training process
for iter = 2:number_of_iterations
   flag=0;
   calculate quasi Hessian matrix $\mathbf{Q}$;
   calculate gradient vector $\mathbf{g}$;
   $\boldsymbol{\Delta}$_backup = $\Delta$;
   while 1
      $\Delta = \boldsymbol{\Delta}$_backup-$((\mathbf{Q}+\mu\mathbf{I})\backslash\mathbf{g}^T)^T$;   % Eq. (7)
      update $\mathbf{w}$;
      update $\boldsymbol{\sigma}$;
      update $\mathbf{u}$;
      update $\mathbf{c}$;
      calculate RMSE(iter);
      Training Procedure Control ($\mu$ adjustment)
   end;
   if RMSE(iter) < required_training_error;
      break;
   end;
end;
```

```
2
for p=1:P            %Number of patterns
  for h=1:H          %Number of hidden units
    calculate output of hidden units $\varphi_h(\mathbf{x}_p)$;     %Eqs. (1-2)
  end;
  for m=1:M
    calculate network output $o_{p,m}$      %Eq. (3)
    calculate error $e_{p,m}$               %Eq. (6)
end;
calculate root mean square error;      %Eq. (21)
```

```
3
Q=0; g=0;
for p=1:P            %Number of patterns
  for h=1:H          %Number of hidden units
    calculate output of hidden units $\varphi_h(\mathbf{x}_p)$;     %Eqs. (1-2)
  end;
  for m=1:M          %Number of outputs
    calculate output $o_{p,m}$;          %Eq. (3)
    calculate error $e_{p,m}$;           %Eq. (6)
    for h=1:H        %Number of hidden units
      calculate $\partial e_{p,m}/\partial w_{h,m}$;      %Eqs. (13) & (14)
      calculate $\partial e_{p,m}/\partial \boldsymbol{\sigma}_h$;       %Eq. (16)
      calculate $\partial e_{p,m}/\partial u_{i,h}$;      %Eq. (18)
      calculate $\partial e_{p,m}/\partial c_{h,i}$;      %Eq. (20)
    end;
    build Jacobian row $\mathbf{j}_{p,m}$;        %Eq. (11)
    calculate sub matrix $\mathbf{q}_{p,m}$;  $\mathbf{Q}=\mathbf{Q}+\mathbf{q}_{p,m}$;      %Eq. (8)
    calculate sub vector $\boldsymbol{\eta}_{p,m}$;  $\mathbf{g}=\mathbf{g}+\boldsymbol{\eta}_{p,m}$;      %Eq. (9)
  end;
end;
```

Fig.3.3 Pseudo code (following MATLAB syntax) of the proposed algorithm to train radial basis function networks. Block (1) is the procedure for weight updating; block (2) evaluates the root

mean square error; block (3) is used for quasi Hessian matrix and gradient vector computation.

3.4 Experimental Result with algorithm comparison

Several practical issues are presented to test the performance of the improved second order (ISO) algorithm, from the point of training speed, required hidden units, training error and generalization error. In this part, at first,the ISO algorithm is compared with several algorithms introduced in Chapter 1, including GGAP, MRAN, RANEKF, RAN and GGAP-GMM. Next, four cases with different training parameters are compared based on two practical problems. Finally, the proposed ISO algorithm is compared with first order gradient algorithm and Gauss-Newton method, based on MATLAB PEAK problem.

The testing environment of the proposed algorithm consists of: Windows 7 Professional 32-bit operating system; AMD Athlon (tm) ×2 Dual-Core QL-65 2.10GHz processor; 3.00GB (2.75GB usable) RAM; MATLAB 2007b platform. All the attributes in the data set are normalized (divided by the maximum value of the attribute) in range [0, 1], and 100 trials are repeated under similar conditions for each study case, when applying the proposed algorithm for training/testing.

3.4.1 Comparing with other algorithms

In the performed experiments, three practical problems, including Boston housing problem, abalone age prediction and fuel consumption prediction from [33], are applied to test the performance of the proposed ISO algorithm, by comparing with other five algorithms. Each testing case for ISO algorithm is repeated for 100 trials and testing results of other algorithms are from [24] [25].

The Boston Housing problem has total of 506 observations, each of which consists of 13 input attributes (12 continuous attributes and 1 binary-valued attribute) and 1 continuous output attribute (the median value of owner-occupied homes). For each trial, 481 randomly selected observations are going to be applied for training, and the rest 25 observations will be used to test the trained radial basis function network. The experiment results are shown in Table 4. Fig. 3.4 shows the training/testing RMS error trajectories when increasing the number of RBF units.

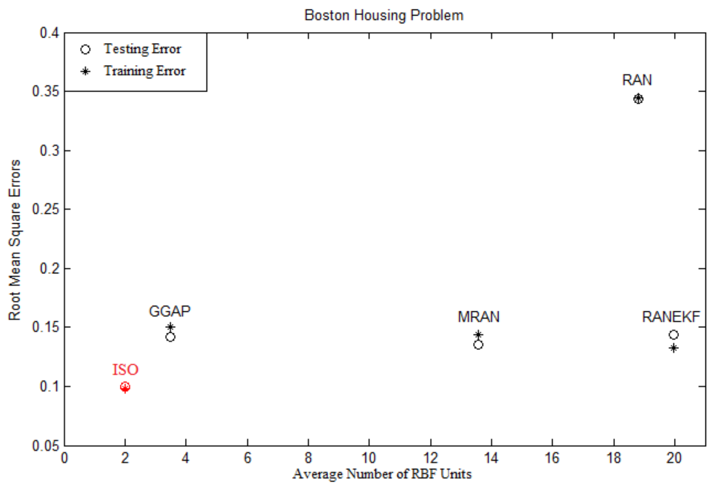


Fig. 3.4 RMS errors vs. average number of RBF units for Boston housing problem

TABLE 4 PERFORMANCE COMPARISON FOR BOSTON HOUSING PROBLEM

Algorithms	CPU Time (s)		Training RMSE		Testing RMSE		Number of RBF Units (Mean)
	Mean	Dev	Mean	Dev	Mean	Dev	
GGAP	1.2399	0.2812	0.1507	0.0128	0.1418	0.0466	3.5
MRAN	12.731	2.2585	0.1440	0.0108	0.1356	0.0411	13.58
RANEKF	22.572	6.4159	0.1328	0.0086	0.1437	0.0464	19.98
RAN	4.2664	0.4846	0.3449	0.0620	0.3432	0.0770	18.8
ISO	0.6192	0.0591	0.1327	0.0471	0.1403	0.0553	1
ISO	1.1103	0.1596	0.0996	0.0383	0.1018	0.0430	2
ISO	1.5349	0.1688	0.0904	0.0318	0.0926	0.0369	3

As shown in Fig. 3.4, for the Boston housing problem, the ISO algorithm can obtain smaller training/testing errors than other four algorithms with only 2 RBF units.

The Abalone problem consists of 4,177 observations, each of which consists of 7 continuous input attributes and 1 continuous output attribute (age in years). For each trial, 3,000 randomly selected observations are applied as training data and the remaining 1,177 observations are applied for testing. The experimental results are presented in Table 5. Fig. 3.5 shows the relationship between training/testing RMS errors and the average number of RBF units.

As shown in Fig.3.5, the proposed ISO algorithm can reach similar or smaller training/testing errors with other algorithms using significantly compact RBF network consisting of 4 RBF units.

TABLE 5 PERFORMANCE COMPARISON FOR ABALONE AGE PREDICTION

Algorithms	CPU Time (s)		Training RMSE		Testing RMSE		Number of RBF Units (Mean)
	Mean	Dev	Mean	Dev	Mean	Dev	
GGAP-GMM	/	/	0.08	/	0.0850	0.0027	5.13
GGAP	83.784	73.401	0.0963	0.0061	0.0966	0.0068	23.62
MRAN	1500.4	134.08	0.0836	0.0039	0.0837	0.0042	87.571
RANEKF	90806	18193	0.0738	0.0042	0.0794	0.0053	409
RAN	105.17	6.1714	0.0931	0.0091	0.0978	0.0092	345.58
ISO	**5.9672**	**0.7495**	**0.0792**	**0.0109**	**0.0792**	**0.0101**	**4**
ISO	**8.4672**	**0.9885**	**0.0778**	**0.0066**	**0.0762**	**0.0085**	**5**
ISO	**11.625**	**1.7499**	**0.0748**	**0.0047**	**0.0738**	**0.0035**	**6**

/ data not available in the literature [25]

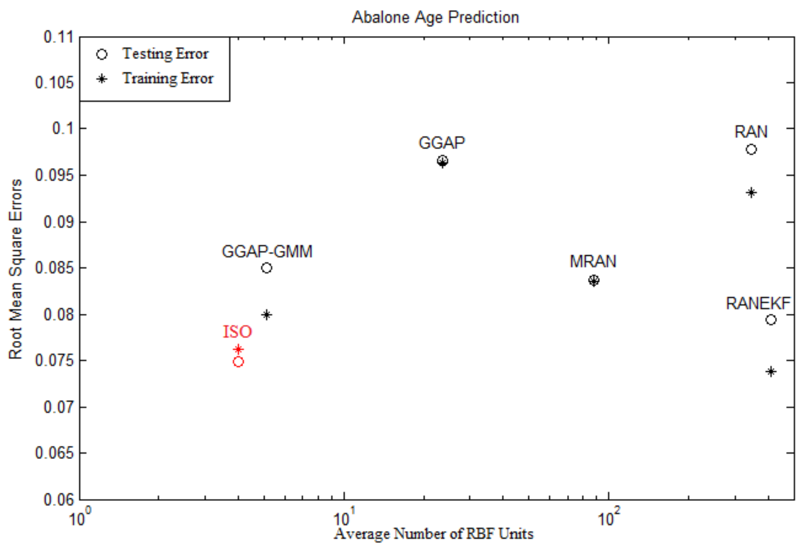


Fig. 3.5 RMS errors vs. average number of RBF units for abalone age prediction problem

The Auto MPG problem has 398 patterns. Each pattern consists of seven continuous input attributes and one continuous output attribute (the fuel consumption in mile-per-gallon). For each trial, 320 randomly selected patterns are applied for training and the remaining 78 patterns are applied for testing. The experimental results are shown in Table 6. Fig. 3.6 presents the changing of training/testing RMS errors as the average number of RBF units increases.

TABLE 6 PERFORMANCE COMPARISON FOR FUEL CONSUMPTION PREDICTION OF AUTOS

Algorithms	CPU Time (s)		Training RMSE		Testing RMSE		Number of RBF Units (Mean)
	Mean	Dev	Mean	Dev	Mean	Dev	
GGAP-GMM	/	/	0.11	/	0.1167	0.0134	3.6
GGAP	0.4520	0.0786	0.1144	0.0132	0.1404	0.0270	3.12
MRAN	1.4644	0.2453	0.1086	0.0100	0.1376	0.0226	4.46
RANEKF	1.0103	0.1694	0.1088	0.0117	0.1387	0.0289	5.14
RAN	0.8042	0.1417	0.2923	0.0808	0.3080	0.0915	4.44
ISO	**0.2105**	**0.0107**	**0.0975**	**0.0463**	**0.0995**	**0.0433**	**1**
ISO	**0.3043**	**0.0248**	**0.0784**	**0.0294**	**0.0817**	**0.0289**	**2**
ISO	**0.5922**	**0.0683**	**0.0622**	**0.0077**	**0.0645**	**0.0094**	**3**

/ data not available in the literature [25]

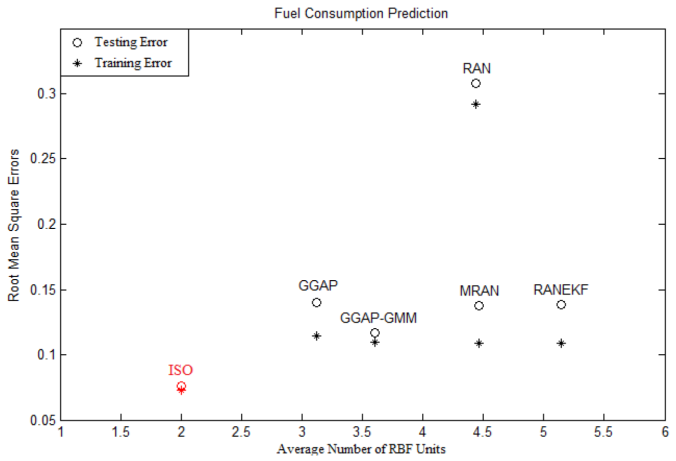


Fig. 3.6 RMS errors vs. average number of RBF units for fuel consumption prediction problem

As the comparison results shown in Fig. 3.6, for the fuel consumption prediction problem, the proposed ISO method can reach smaller training/testing errors than other algorithms, with very compact RBF network consisting of 2 RBF units. One may also notice that the ISO method got better generalization ability on this problem, since the differences between training errors and testing errors are much smaller than other algorithms.

The comparison results presented in Tables 1-3 show that the proposed algorithm can reach similar or smaller training/testing RMS errors with much less number of RBF units and less training time than other algorithms.

3.4.2 Performance with different training parameters

In the experiments conducted, two of the most popular data sets in [33], wine classification and car evaluation, will be applied to test the proposed ISO algorithm.

In the wine problem, there are three types of wines required to be classified according with other 13 attributes. There are 178 observations in total.

In the car evaluation problem, 6 input attributes are used to evaluate the degree of satisfaction about the car. String data are replaced by natural numbers 1, 2, 3…. There are 1,728 observations in total.

With the two benchmark problems, the following four cases with different variables are tested, using the proposed ISO algorithm:

- Case 1: only output weight matrix $\boldsymbol{w}$ is updated;
- Case 2: only input weight matrix $\boldsymbol{u}$ is updated;
- Case 3: output weight matrix $\boldsymbol{w}$, width vector $\boldsymbol{\sigma}$ and center matrix $\boldsymbol{c}$ are updated;
- Case 4: output weight matrix $\boldsymbol{w}$, width vector $\boldsymbol{\sigma}$, input weight matrix $\boldsymbol{u}$ and center matrix $\boldsymbol{c}$ are all updated.

All the input weights, output weights and widths are randomly generated in range (0, 1). All centers are randomly selected from the training dataset. The maximum iteration is 100 and each testing case is repeated for 50 trials.

As shown in Figs. 3.7 and 3.8, several observations can be concluded:

(1) As the number of RBF units increases, except the case 2, the training errors decrease stably.

(2) Case 2 (squares in line) shows the worst performance which is mainly depends on the initial conditions. So adjusting input weights only is not helping for RBF networks design.

(3) For the same number of RBF units, case 4 (circles in line) mostly gets smaller training errors than other three cases.

(4) As the number of RBF units increases, the difference of training results between case 3 and case 4 becomes small.

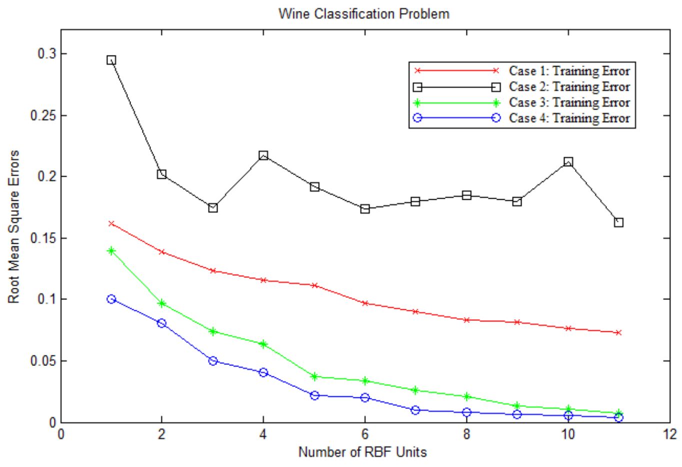


Fig. 3.7 Root mean square errors vs. number of RBF units for wine classification problem

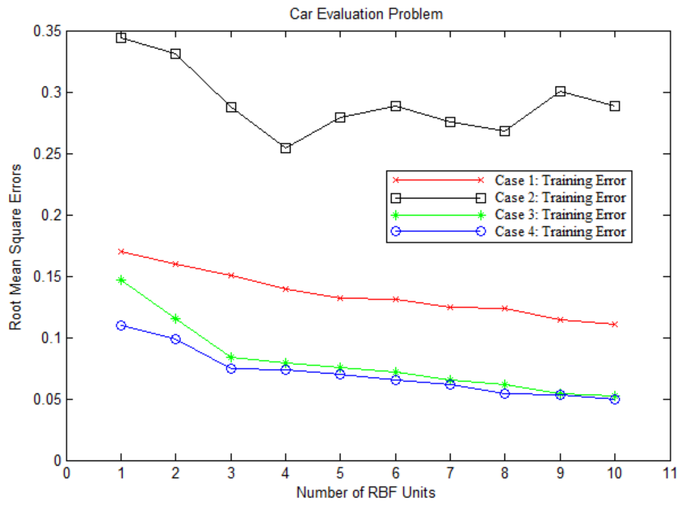
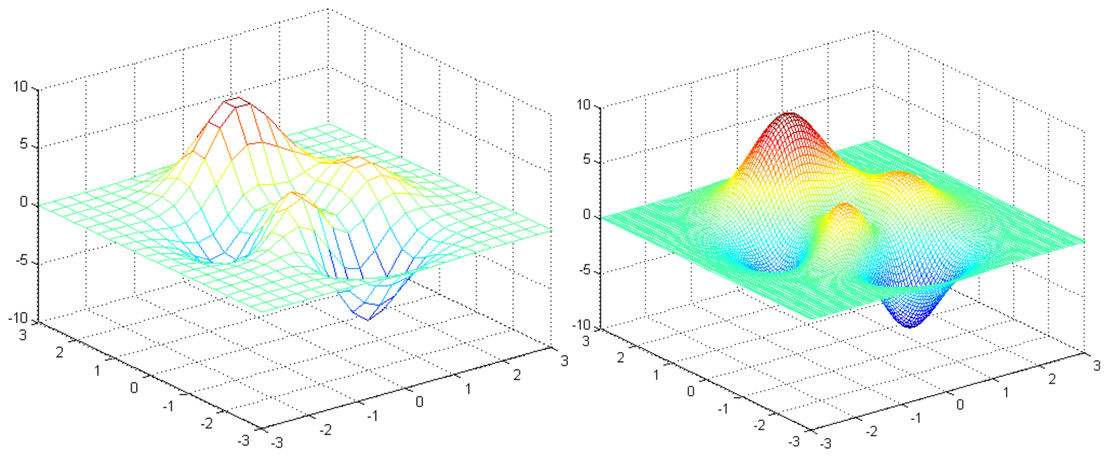


Fig. 3.8 Root mean square errors vs. number of RBF units for car evaluation problem

3.4.3 Training speed comparison

In the experiment conducted, the proposed ISO method is compared with the first order gradient method (with momentum) [34] and enhanced Gauss-Newton method, by solving the PEAK function approximation problem. Notice that, the original Gauss-Newton method [35] seldom converges because Hessian matrix is mostly not invertible for complex error surfaces. In the experiment, the Gauss-Newton method is enhanced by adding a constant value to the diagonal elements of Hessian matrix when it is not invertible.

The purpose is to approximate the surface in Fig. 3.9b using the surface in Fig. 3.9a. All the data comes from MATLAB PEAKS function. RBF network with 5 hidden units is applied to do the approximation. The maximum number of iteration is 200 for ISO algorithm, 10,000 for first order gradient method and 1,000 for the enhanced Gauss-Newton algorithm. All the parameters, including input weights, output weights, centers and widths are adjusted by the three algorithms.



(a) (b)

Fig. 3.9 Peaks function approximation problem: (a) training data, *20×20=400* points; (b) testing data, *100×100=10,000* points.

With the same initial centers, widths and input weights, but randomly generated output weights, the training RMS error trajectories of the three algorithms for 10 trials each are shown in Fig. 3.10. One may notice that the proposed ISO algorithm (black dot-line) costs significantly less iterations than first order gradient method (red solid-line) and converges to much smaller RMS errors than both first order gradient method and enhanced Gauss-Newton method (blue dash-line), in the limited iterations (when training errors get saturated).

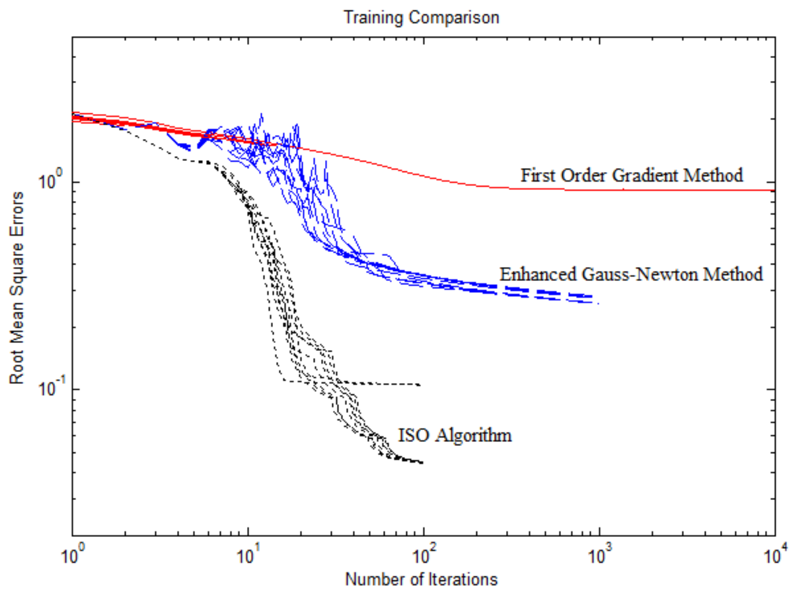


Fig. 3.10 Training RMS error trajectories: red solid-line is for first order gradient method, blue dash-line is for enhanced Gauss-Newton method and black dot-line stands for ISO algorithm (10 trials for each algorithm)

Figs. 3.11-3.13 show the best error surfaces of the approximating results we have tried using first order gradient method (Fig. 3.11), enhanced Gauss-Newton method (Fig. 3.12) and the proposed ISO algorithm (Fig. 3.13), respectively.

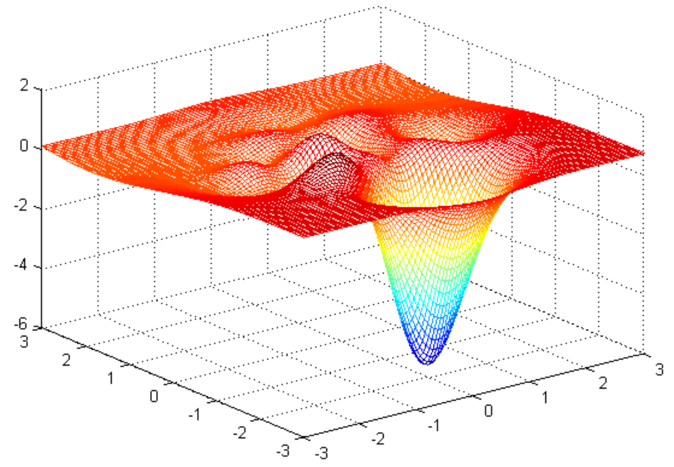


Fig. 3.11 Error surface of the approximating result using first order gradient method, with

E_{Train}=0.8911 and E_{Test}=0.9192

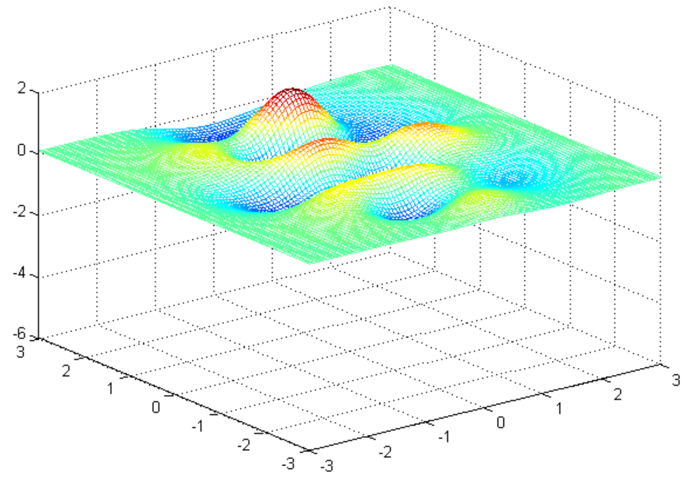


Fig. 3.12 Error surface of the approximating result using enhanced Gauss-Newton method, with

E_{Train}=0.2769 and E_{Test}=0.2869

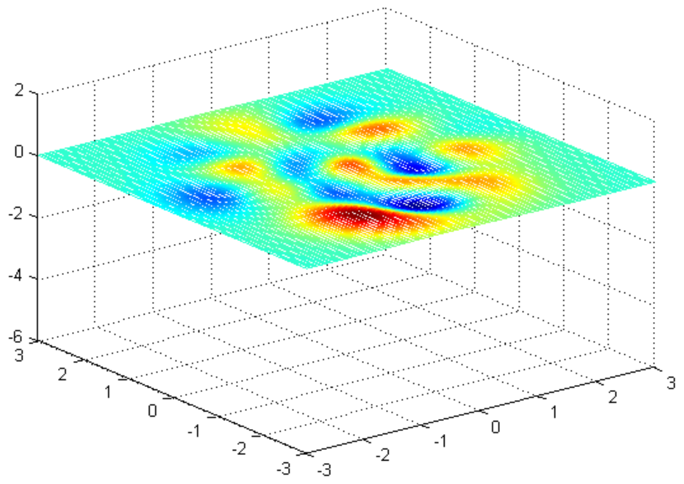


Fig. 3.13 Error surfaces of the approximating result using the proposed ISO method with E_{Train}=0.0424 and E_{Test}=0.0440

One may notice that, with the smallest training error, the proposed ISO algorithm also gets the smaller approximation errors (better generalization results) than both first order gradient method and enhanced Gauss-Newton method. Obviously, because of its limited search ability, first order gradient method is not able to adjust the centers of RBF units properly.

3.5 Conclusion

In this chapter, we introduced an advanced ISO algorithm which is derivate d from Levenberg-Marquardt algorithm for training RBF networks. During the implementation of the ISO proposed algorithm, matrix operations are replaced by vector operations, which lead to significant memory reduction and speed benefit. Also, all parameters including input weights, output weights, centers and widths are all adjusted during the training process. Variable space with higher dimensions makes the designed radial basis function networks perform more compact and accurate for approximation when the network size is small. Compact radial basis function networks benefit the design in two aspects. First of all, compact architecture could be more efficient for hardware implementation. On the other hand, the less number of RBF units used for design, the better generalization ability the trained networks can obtain. The ISO exhibits the powerful search ability and fast convergence in the presented examples.

Chapter 4

Incremental Design of RBF networks

As we mentioned in chapter 1, the architecture of RBF networks is fixed to three-layer, so it is easy to design. However, sometime it may need large size hidden units to solve the problem. The networks applied for training should be as compact as possible. For RBF networks, not only the number of RBF units has to be decided, but also the center and width of each RBF unit have to be properly chosen. In this chapter, we will introduce the error correction algorithm which is proposed to find proper initial centers of RBF units and design compact RBF networks. The performance of the proposed algorithm is also evaluated.

4.1 Error correction algorithm

The basic idea of the ErrCor algorithm is to use kernel function with peak/valley shape to compensate the biggest error in error surface step by step, so as to reduce the approximation errors gradually.

4.1.1 Graphical Interpretation

In order to illustrate the error correction process, let us have an example to approximate the peak surface in MATLAB. The destination surface consists of 900 points (as training data), obtained by command *peaks(30)*, is shown in Fig. 4.1.

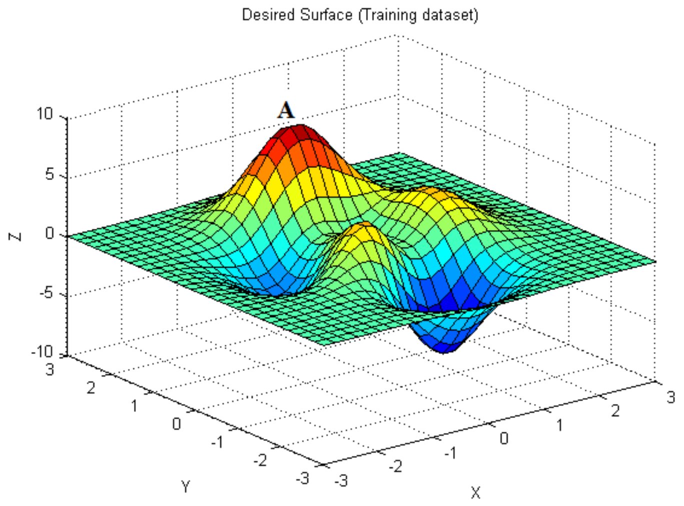
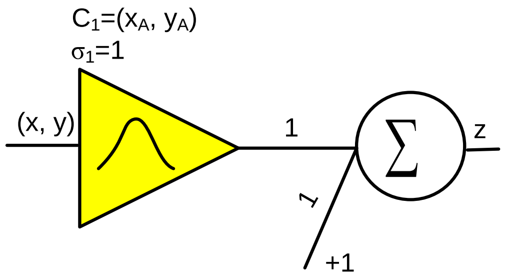
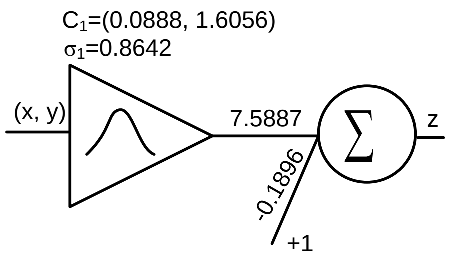


Fig. 4.1 Peak surface (900 points)

At the beginning, since the number of RBF units is “0”, it is reasonable to assume that all the actual outputs from RBF network are “0”. In this case, the desired surface in Fig. 4.1 can be also considered as the error surface which is obtained by (3-15).

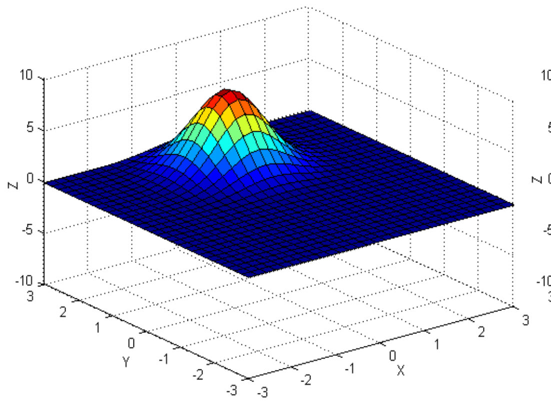
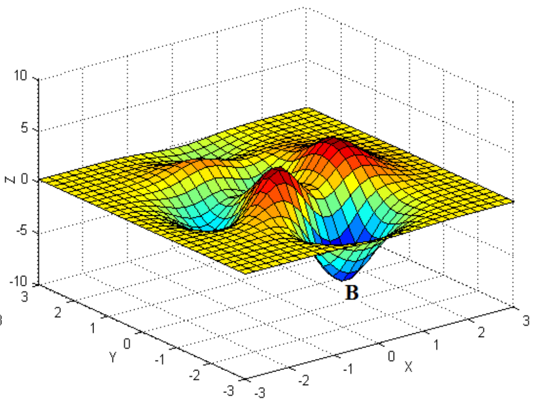
By going through the data of error surface in Fig. 4.1, the location (x_A=-0.1034, y_A=1.5517) of the highest peak (marked as ***A***) is found. Then, the first RBF unit can be added with initial center (x_A, y_A), as shown in Fig. 4.2a. By applying Levenberg Marquardt algorithm (introduced in chapter 3) for parameter adjustment, the trained network is shown in Fig. 4.2b. Based on the training results, the outputs of the RBF network (Fig. 4.2b) are visualized in Fig. 4.3a and the error surface (Fig. 4.3b) is obtained as the difference between Fig. 4.1 and Fig. 4.3a. Comparing the error surfaces in Figs. 4.1 and 4.3b, one may notice that, the highest peak (marked as ***A***) in Fig. 4.1 is eliminated from Fig. 4.3b.





(a) (b)

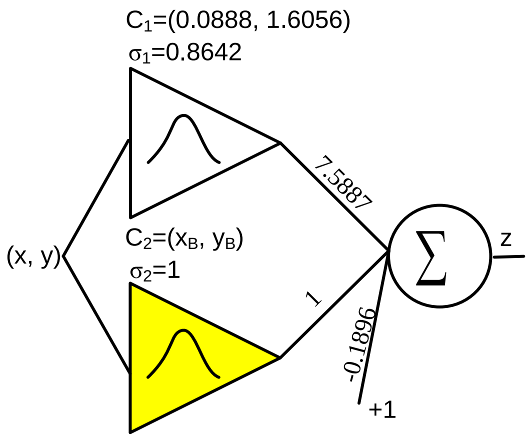
Fig. 4.2 RBF network with 1 RBF unit: (a) initialed RBF network; (b) trained RBF network. Yellow RBF unit is newly added.

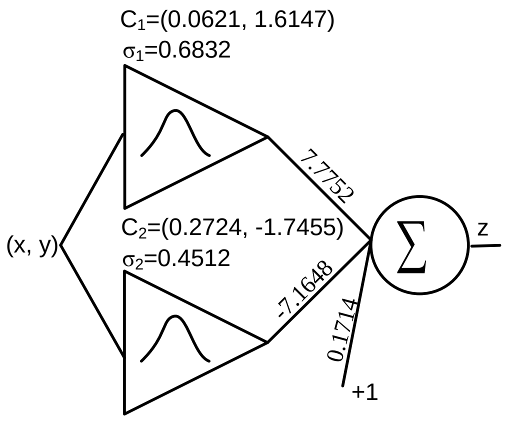
(a) (b)

Fig. 4.3 Result surfaces of the RBF network in Fig. 4.2b with 1 RBF unit: (a) actual output surface; (b) error surface

By observing the data of the error surface in Fig. 4.3b, it is obtained that the lowest valley (marked as ***B***) has the coordinates (x_B=0.3103, y_B=-1.5517). Then, the second RBF unit is added with initial center (x_B, y_B) and the parameters of the first RBF unit keeps the same as the training results from last step, as shown in Fig. 4.4a. After training process, the network parameters can be adjusted as shown in Fig. 4.4b. Fig. 4.5 presents the actual outputs and errors obtained from the trained network (Fig. 4.4b). Again, one may notice that, the lowest valley (marked as ***B***) in error surface Fig. 4.3b is eliminated from Fig. 4.5b.

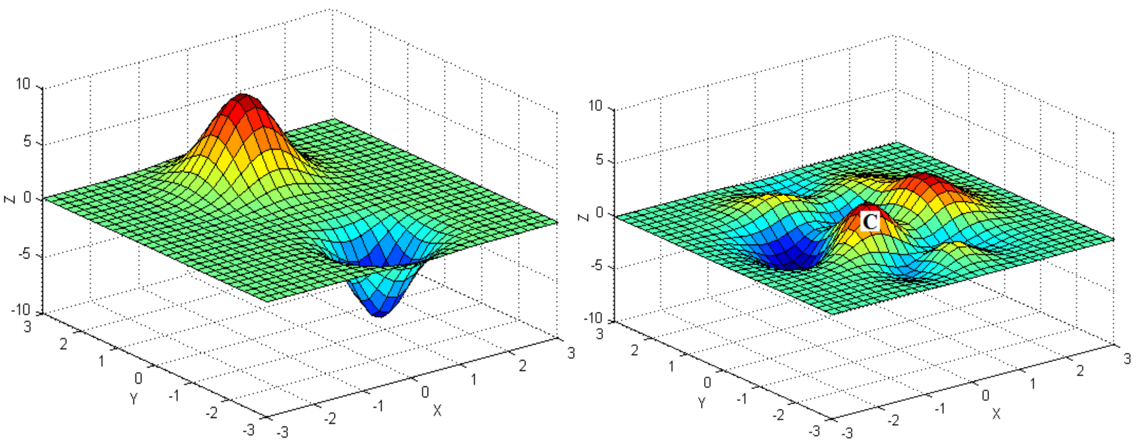


(a)



(b)

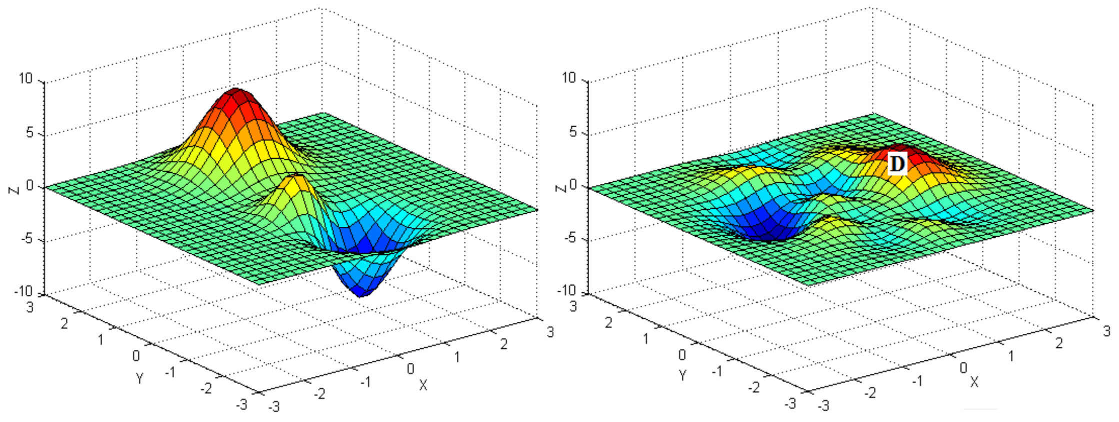
Fig. 4.4 RBF network with 2 RBF units: (a) initialed RBF network; (b) trained RBF network. Yellow RBF unit is newly added



(a) (b)

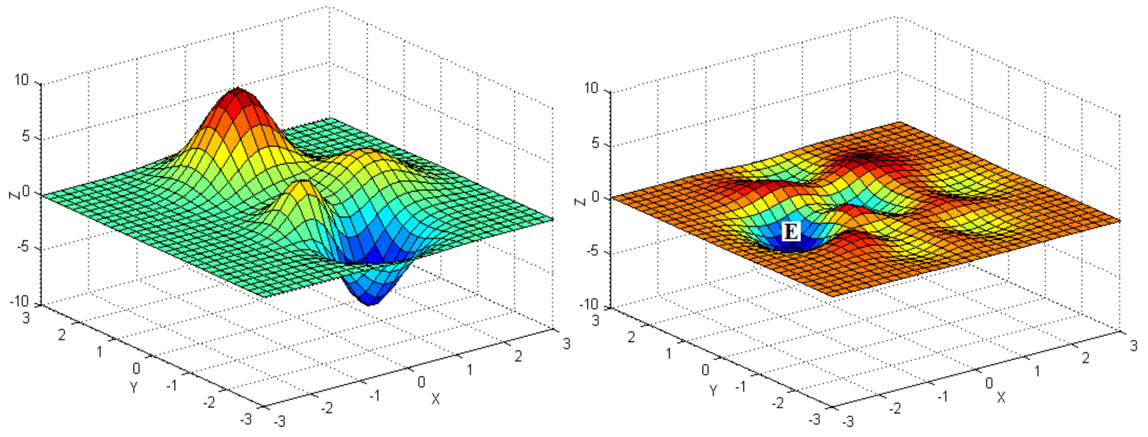
Fig. 4.5 Result surfaces of the RBF network in Fig. 4.4b with 2 RBF units: (a) actual output surface; (b) error surface

By repeating the error correction process above, the result surfaces shown in Figs. 4.6, 4.7 and 4.8 can be obtained from the trained RBF networks consisting of 3, 4 and 5 hidden units, as shown in Figs. 4.9b, 4.10b and 4.11b, respectively.



(a) (b)

Fig. 4.6 Result surface of the RBF network in Fig. 4.9b with 3 RBF units: (a) actual output surface; (b) error surface



(a) (b)

Fig. 4.7 Result surface of the RBF network in Fig. 4.10b with 4 RBF units: (a) actual output surface; (b) error surface

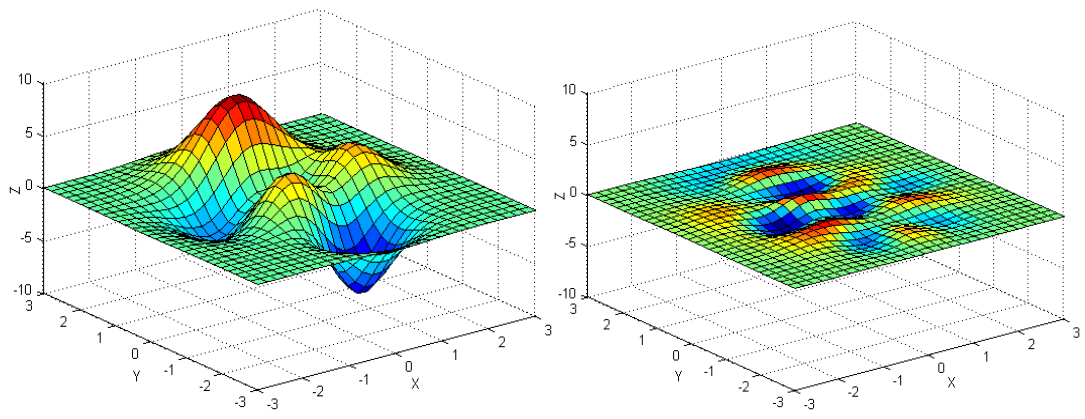
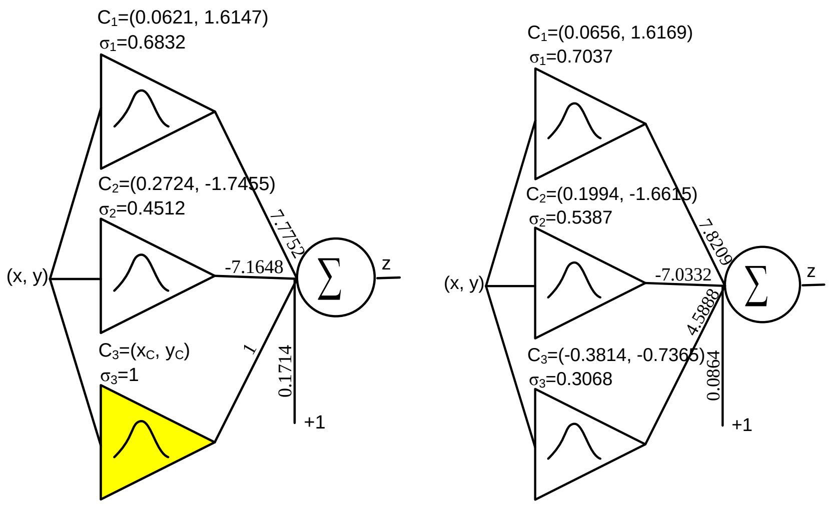


Fig. 4.8 Result surface of the RBF network in Fig. 4.11b with 5 RBF units: (a) actual output surface; (b) error surface



(a) (b)

Fig. 4.9 RBF network with 3 RBF units: (a) initialed RBF network; (b) trained RBF network. Yellow RBF unit is newly added

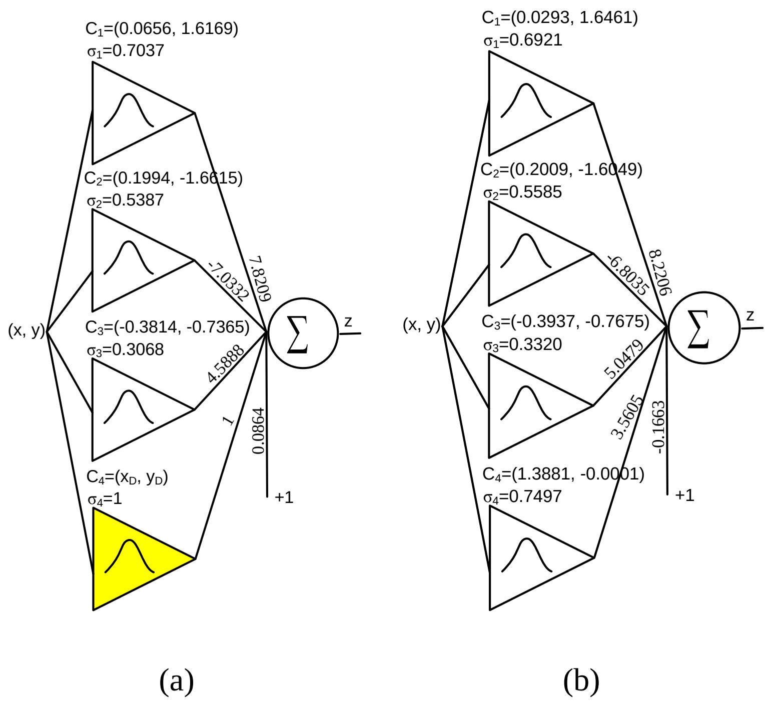
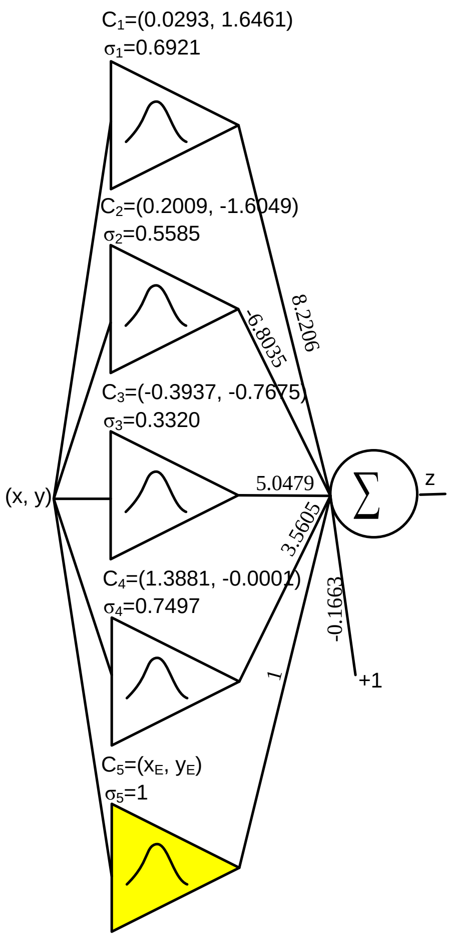
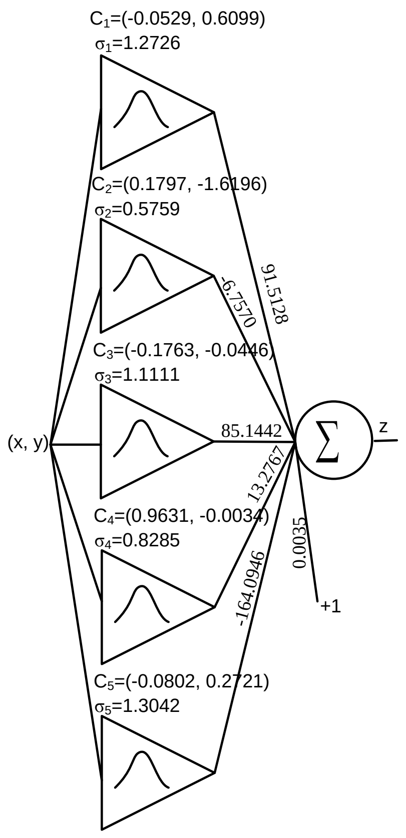


Fig. 4.10 RBF network with 4 RBF units: (a) initialed RBF network; (b) trained RBF network. Yellow RBF unit is newly added





(a) (b)

Fig. 4.11 RBF network with 5 RBF units: (a) initialed RBF network; (b) trained RBF network. Yellow RBF unit is newly added

From Figs. 4.6, 4.7 and 4.8, it can be seen that:

- The highest peak in error surface Fig. 4.5b located at (x_C=-0.5172, y_C=-0.7241) (marked as ***C***) is eliminated from error surface Fig. 4.6b, by the initialed and then trained RBF network in Fig. 4.9.

- The highest peak in error surface Fig. 4.6b located at (x_D=1.3448, y_D=-0.1034) (marked as ***D***) is eliminated from error surface Fig. 4.7b, by the initialed and then trained RBF network in Fig. 4.10.
- The lowest valley in error surface Fig. 4.7b located at (x_E=-1.3448, y_E=0.1034) (marked as ***E***) is eliminated from error surface Fig. 4.8b, by the initialed and then trained RBF network in Fig. 11.

For RBF networks design in Figs. 4.2, 4.4, 4.9, 4.10 and 4.11, notice that, yellow units are newly added and initialed by related peaks (or valleys) as centers; while other units are initialized as the training results of last steps.

4.1.2 Feature Study

The training algorithms can be usually evaluated from three aspects: (1) stability; (2) network efficiency; (3) time efficiency. Based on the peak surface approximation problem, let us give a rough evaluation on the properties of the proposed ErrCor algorithm.

Fig. 4.12 shows the training process of each the RBF network. It can be noticed that, following the error correction procedure, the training errors are reduced stably as the increase of RBF units; the more RBF units are added, the smaller training errors can be obtained.

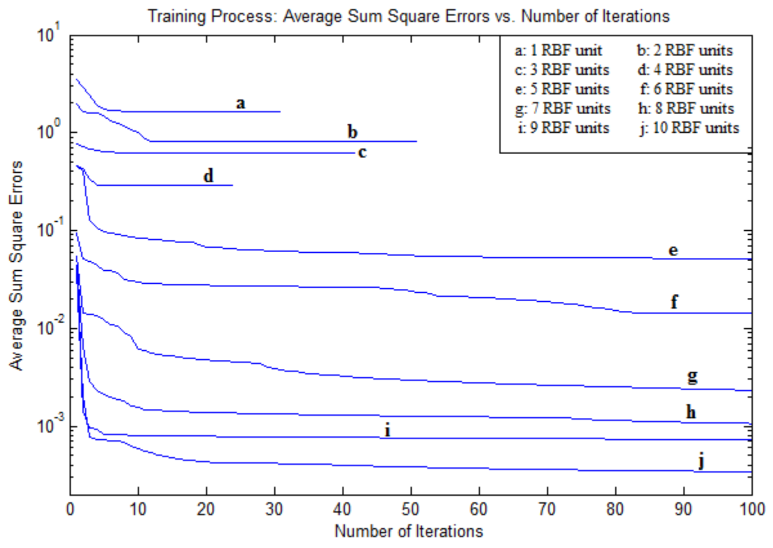


Fig. 4.12 Training Process: average sum square errors vs. number of iterations

In order to evaluate the network efficiency and time efficiency, two different network construction strategies are compared: (a) ErrCor Algorithm; (b) Pick up initial centers randomly from training dataset. The testing results are averaged from 100 trials for all the testing networks.

In Fig. 4.13, each curve represents the relationship between the training average sum square errors and the number of RBF units.

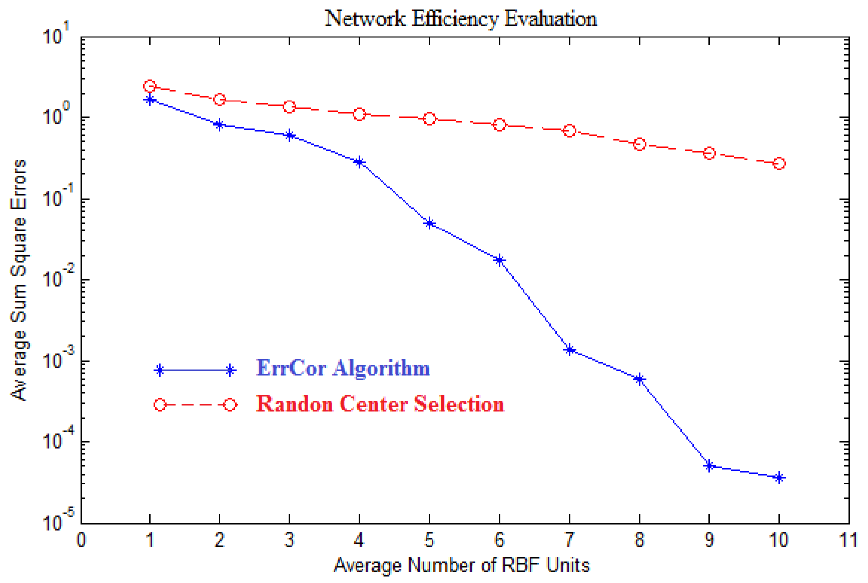


Fig. 4.13 Network efficiency evaluation: average sum square errors vs. number of RBF units. Blue solid line is for (a) ErrCor algorithm; while red dash line is for (b) the random center selection strategy

It can be noticed that, the ErrCor algorithm is network efficient in constructing RBF networks. The ErrCor (blue solid line) cost less number of RBF units than the random center selection strategy (red dash line), in order to reach the similar training error levels.

In Fig. 4.14, the two curves show the relationship between the training average time and the number of RBF units.

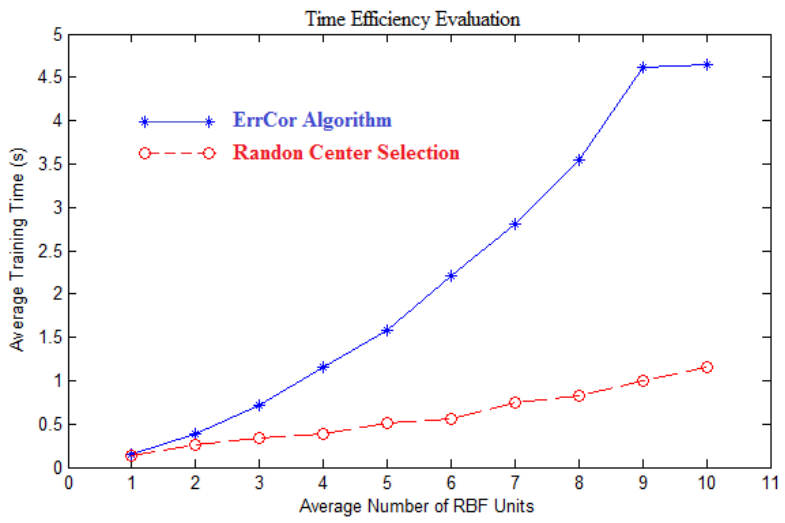


Fig. 4.14 Time efficiency evaluation: average training time costs vs. number of RBF units. Blue solid line is for (a) ErrCor algorithm; while red dash line is for (b) the random center selection strategy

It can be seen that, the ErrCor algorithm (blue solid line) costs more training time than the random center selection strategy (red dash line) when the sizes of RBF networks are the same. The slow computation of the ErrCor algorithm is due to the greedy search: each time a new RBF unit is added, the whole RBF network has to be retained.

4.1.3 The implementation of ErrCor Algorithm

Generally, the proposed ErrCor algorithm can be organized as the pseudo code shown in

Fig.4.15.

```
Initialization;
actual_output=0;
for n=1:maximum_RBF_unit
    err_surf=abs(desired_output – actual_output);
    find the index index_ of the maximum value in vector err_surf;
    set center of newly added RBF unit as x(index_,:);
    set output weight of newly added RBF unit as 1;
    set width of newly added RBF unit as 1;
    if n > 1
        initial other RBF units as the training results of step n-1;
    end;
    evaluate error(1);
    for iter = 2:maximum_iteraiton
        calculate quasi Hessian matrix Q and gradient vector g;
        update parameters;              %Eq. (3)
        evaluate error(iter);
        if (error(iter)-error(iter-1)) < minimum_difference
           break;
        end;
    end;
    If error(iter) < desired_error
        break;
    end;
    update actual_output;
end;
```

Fig. 4.15 Pseudo code of the proposed ErrCor algorithm

In the implementation of ErrCor algorithm, if there are duplicated highest peaks (or lowest valleys), the one with the smallest index will be selected as the initial center of the next newly added RBF unit.

4.2 Performance evaluation of ErrCor algorithm

Several examples are presented to test the performance of the proposed ErrCor algorithm. The experiments are organized in three parts: (1) comparison with other algorithms; (2) duplicate patterns test; (3) noise patterns test.

The testing environment of the proposed algorithm consists of: Windows 7 Professional 32-bit operating system; AMD Athlon (tm) ×2 Dual-Core QL-65 2.10GHz processor; 3.00GB (2.75GB usable) RAM.

4.2.1 Comparison with Other Algorithms

Four experiments in this part are conducted to test the network efficiency and computation efficiency of the proposed ErrCor algorithm, by comparing with several other algorithms:

- In the experiment ***A***, the proposed ErrCor algorithm is compared with GGAP algorithm, MRAN algorithm, RANEKF algorithm and RAN algorithm.
- In the experiments ***B*** and ***C***, besides the four algorithms in experiment ***A***, the recently developed GGAP-GMM algorithm is added into comparison.
- In the experiment ***D***, the proposed ErrCor algorithm is compared with the algorithms in literature [37-39].

A. Function Approximation

In this experiment, the proposed algorithm is applied to design RBF networks to approximate the following rapidly changing function

$$y(x) = 0.8\exp(-0.2x)\sin(10x) \tag{4-1}$$

In this problem, there are 3000 training patterns with x-coordinates uniformly distributed in range [0, 10]. The testing data set consists of 1500 patterns with x-coordinates randomly generated in the same range [0, 10].

Fig.4.16 shows the training results of proposed ErrCor algorithm and other four algorithms. One may notice that the proposed ErrCor algorithm can reach the similar training/testing error level with much less number of RBF units (18 RBF units). As the number of RBF units increases, the training/testing errors decrease steadily.

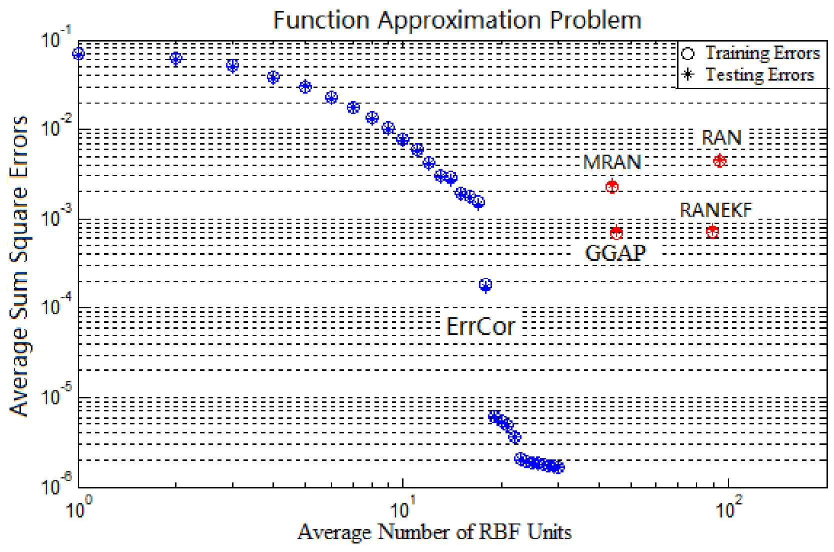


Fig. 4.16 Function approximation problem: training/testing average sum square errors vs.

average number of RBF units

Figs. 4.17-4.20 show the testing results of the proposed ErrCor algorithm, with the number of RBF units equal to 5, 10, 15 and 20, respectively.

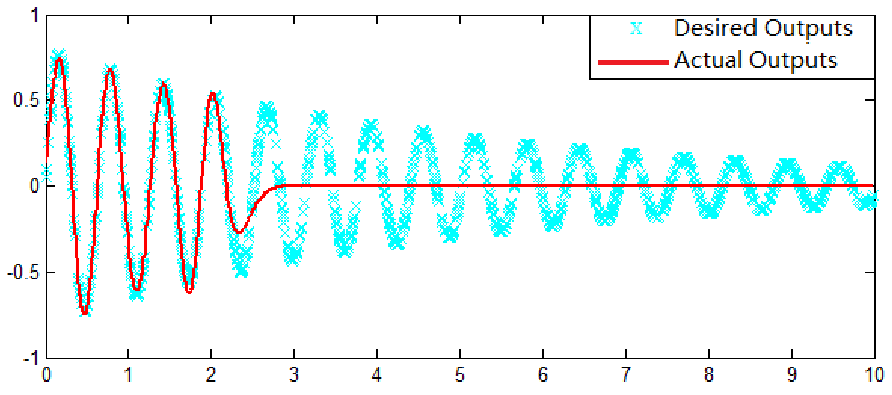


Fig.4.17 Testing results of ErrCor algorithm with 5 RBF units; E_{Train}=2.963×10^{-2} and E_{Test}=3.036×10^{-2}

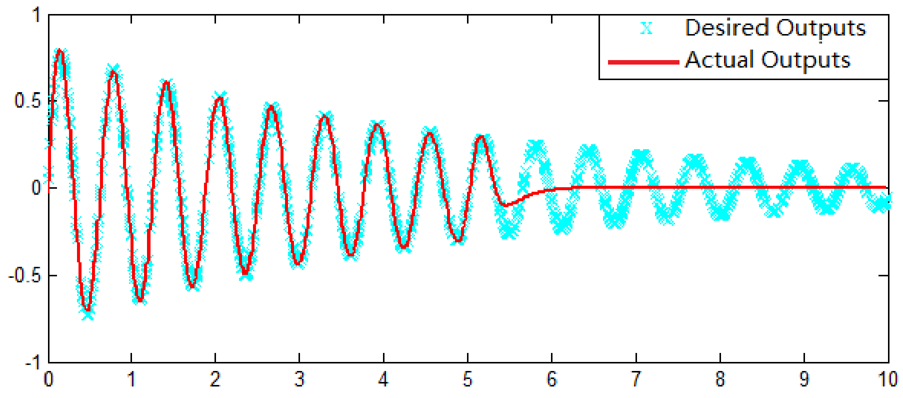


Fig. 4.18 Testing results of ErrCor algorithm with 10 RBF units; $E_{Train}=7.846\times10^{-3}$ and $E_{Test}=7.516\times10^{-3}$

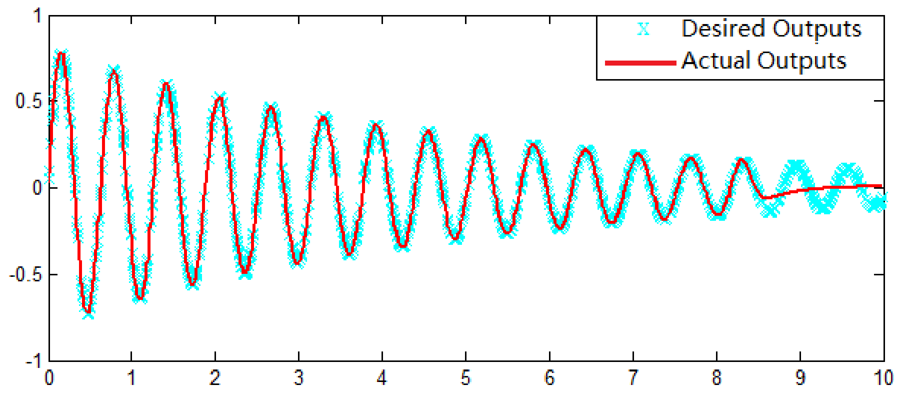


Fig. 4.19 Testing results of ErrCor algorithm with 15 RBF units; $E_{Train}=1.905\times10^{-3}$ and $E_{Test}=1.862\times10^{-3}$

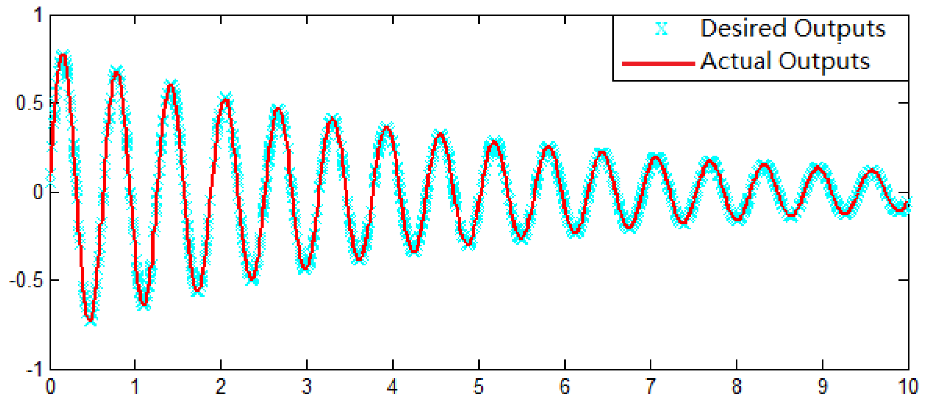


Fig. 4.20 Testing results of ErrCor algorithm with 20 RBF units; $E_{Train}=5.428\times10^{-6}$ and $E_{Test}=5.347\times10^{-6}$

Fig.4.21 presents the comparison of average computation time (blue bars) and average testing errors (red bars). For the proposed ErrCor algorithm, the computation time is counted until the RBF network with 18 units (with smaller training/testing errors than other algorithms) gets trained.

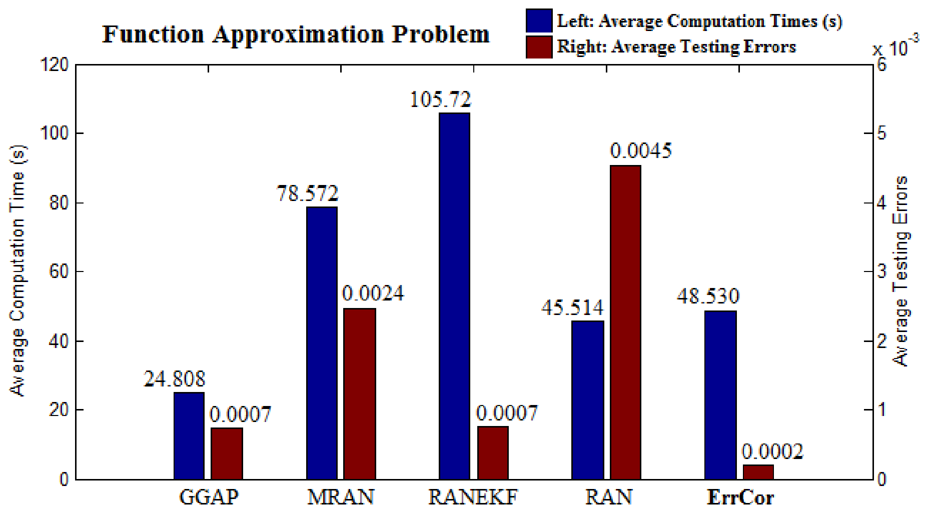


Fig. 4.21 Computation time comparison for function approximation problem

From Fig.4.21, one may notice that, the proposed ErrCor algorithm consumes more time than GGAP algorithm and RAN algorithm to reach the smallest testing error in the five algorithms. The slow convergence of the proposed ErrCor algorithm is mainly due to the retraining process for each newly added RBF unit.

B. Abalone Age Prediction

The abalone age prediction is a practical benchmark in [39] and it consists of 4,177 observations, where 3,000 randomly selected observations are applied as training data set and the remaining 1,177 observations are used to test the trained RBF networks. For each observation, there are 7 continuous input attributes and 1 continuous output attribute (age in years). Fig. 4.22 shows the experimental results of the proposed ErrCor algorithm and other five algorithms.

It can be noticed from Fig. 4.22 that, for the abalone age prediction, the proposed ErrCor algorithm reaches smaller training/testing errors with more compact RBF architecture (4 RBF units) than other five algorithms.

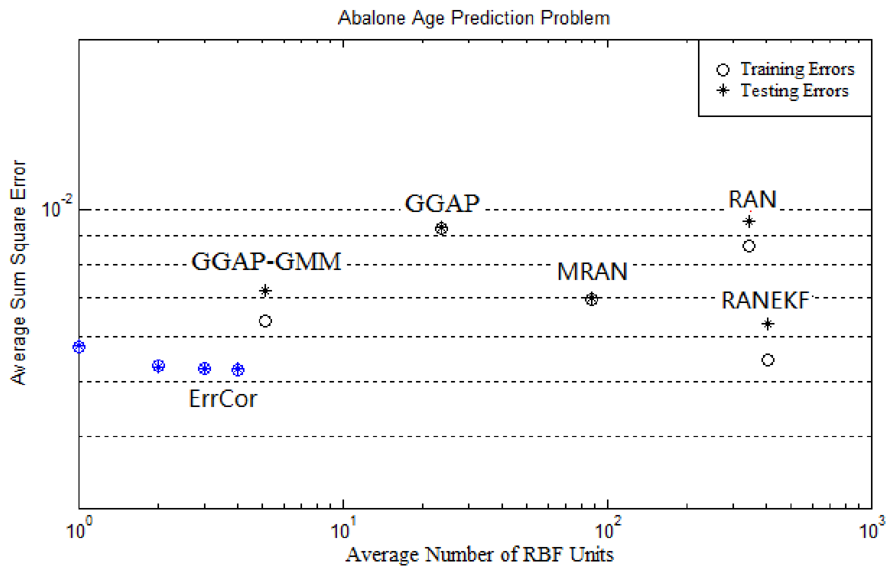


Fig. 4.22 Abalone age prediction problem: training/testing average sum square errors vs. average number of RBF units

Fig. 4.23 shows the comparison of average computation time (blue bars) and average testing errors (red bars) between the proposed ErrCor algorithm and other four algorithms. For ErrCor algorithm, the number of RBF units is increased up to 4 from scratch.

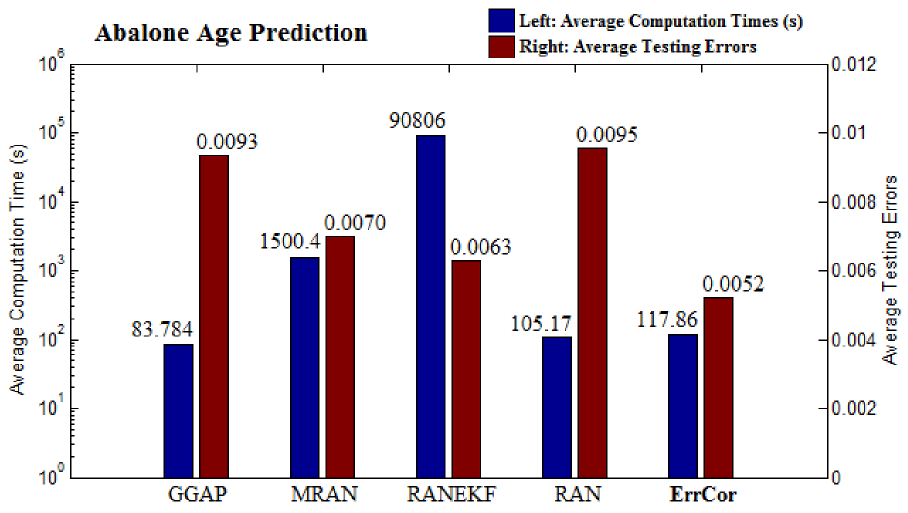


Fig. 4.23 Computation time comparison for abalone age prediction problem.

Again, the proposed ErrCor algorithm computes slower than GGAP algorithm and RAN algorithm, but much faster than MRAN algorithm and RANEKF algorithm. The average testing error of the ErrCor algorithm is smaller than other four algorithms.

C. Fuel Consumption Prediction

The fuel consumption prediction is another benchmark from [33] and it consists of 398 observations, each of which has seven continuous input attributes and one continuous output attribute (the fuel consumption in mile/gallon). For each trial, 320 randomly selected observations are applied training and the remaining 78 observations are applied to test the trained RBF networks. Fig.4.24 shows the relationship between the training/testing average sum square errors and the average number of RBF units.

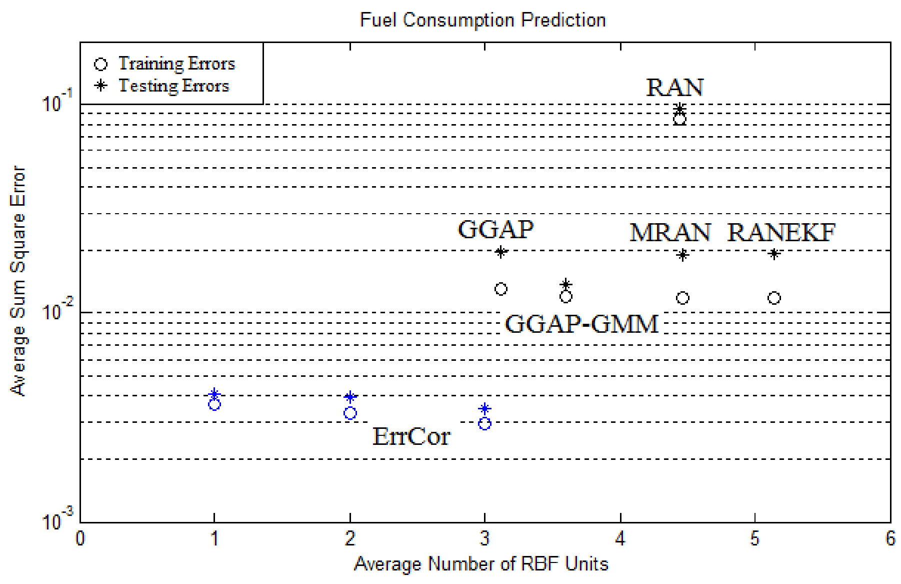


Fig.4.24 Fuel consumption prediction problem: training/testing average sum square errors vs. average number of RBF units

With the experimental results presented in Fig. 4.24, one may notice that the ErrCor algorithm can get smaller training/testing errors than other five algorithms with only one RBF unit. The training/testing errors decrease as the number of RBF units increases.

Fig. 4.25 presents the average computation time (blue bars) and testing errors (red bars) of the five algorithms. For the proposed ErrCor algorithm, only one RBF unit is applied for network construction.

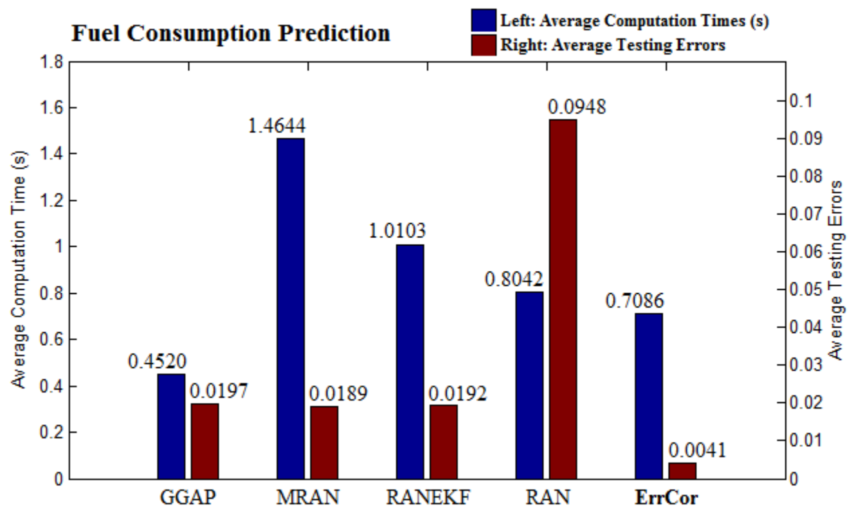


Fig.4.25 Computation time comparison for fuel consumption prediction.

It can be seen that, for the fuel consumption prediction, the proposed ErrCor algorithm still works less efficiently than GGAP algorithm, but faster than the other three algorithms. The average testing error of the ErrCor algorithm is the smallest in the five algorithms.

D. Two-Spiral Classification Problem

Two-spiral classification problem is often considered as a very complex benchmark to evaluate the efficiency of learning algorithms and network architectures. The problem is described in chapter 2.

Applying the ErrCor algorithm, Fig. 4.26 shows all the 30 retraining processes when starting the network construction from scratch.

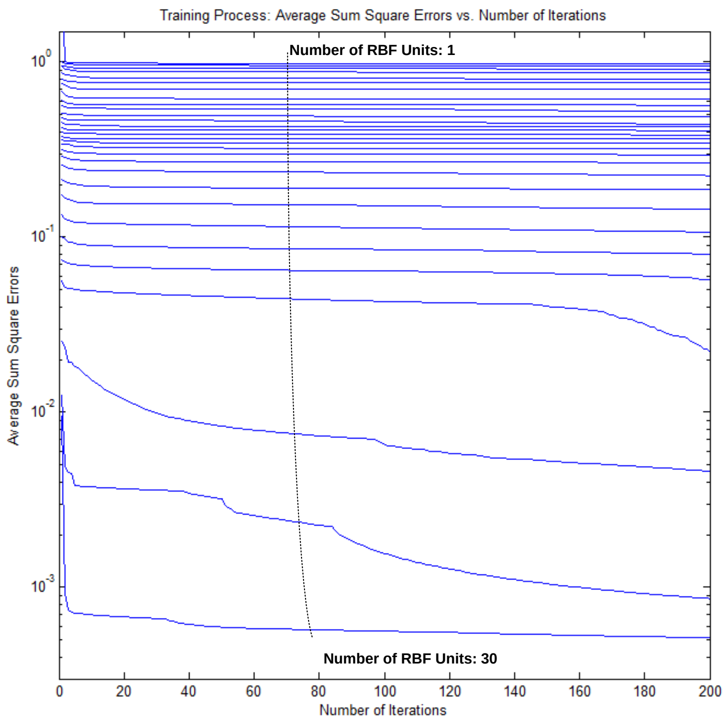
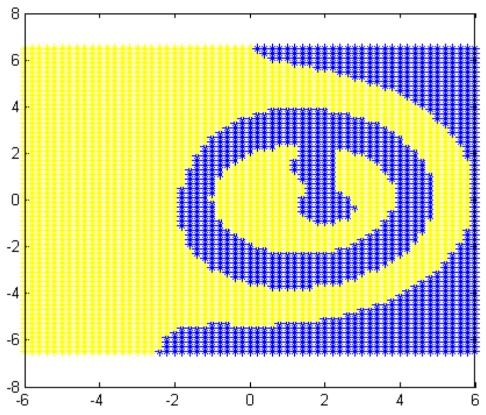


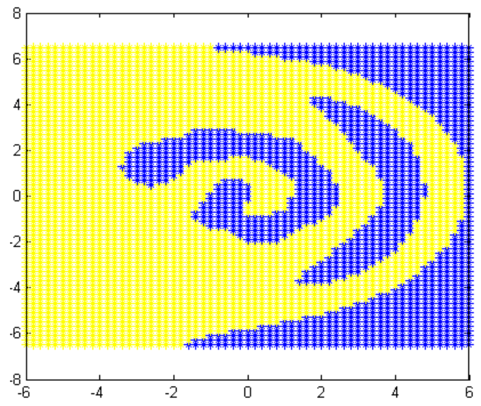
Fig.4.26 Training process: average sum square errors vs. number of iterations. The number of RBF units is increased from 1 to 30

One may notice that, each newly added RBF unit contributes the error reduction during the training process. After a total of 30×200=6,000 iterations, the RBF network with 30 RBF units approaches to the training error level 0.005.

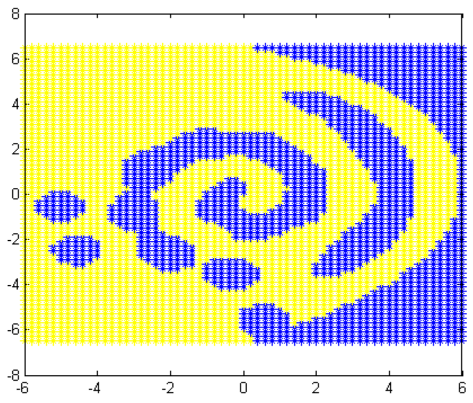
Figs. 4.27a-4.27e show the generalization results of the proposed ErrCor algorithm, with number of RBF units equal to 6, 12, 18, 24 and 30.



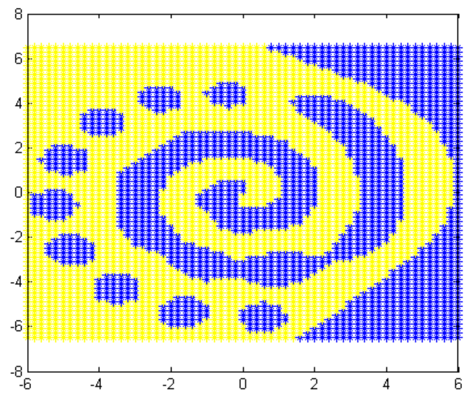
(a)



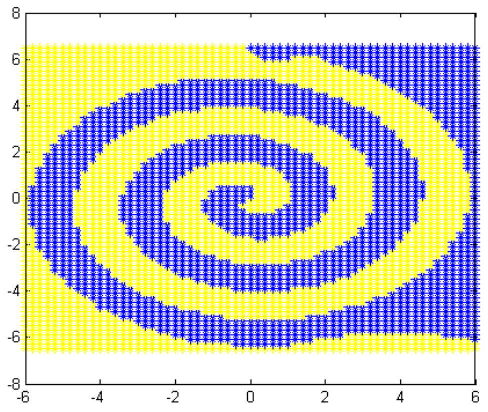
(b)



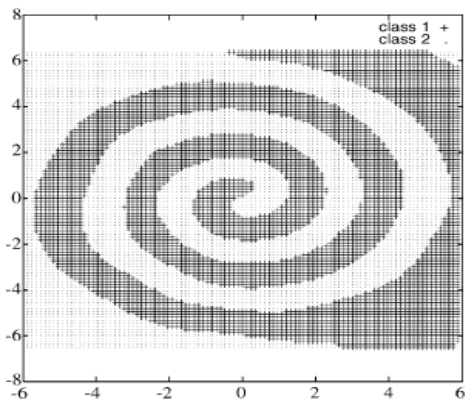
(c)



(d)



(e)



(f)

Fig. 4.27 Generalization results of the two-spiral problem with different number of RBF units, trained by the ErrCor algorithm: (a) 6 RBF units, ErrCor algorithm, E_{Train}=0.7617; (b) 12 RBF units, ErrCor algorithm, E_{Train}=0.4412; (c) 18 RBF units, ErrCor algorithm, E_{Train}=0.3206; (d) 24 RBF units, ErrCor algorithm, E_{Train}=0.1070; (e) 30 RBF units, ErrCor algorithm, E_{Train}=0.0005; (f) 70 RBF units described in [38].

The RBF-MLP networks proposed in [36] required at least 74 RBF units to solve the two-spiral problem. Using the orthonormalization procedure [37], the two-spiral problem can be solved with at least 64 RBF kernel functions. It was reported in [38] that the two-spiral problem was solved using 70 hidden RBF units and the average computation time is 120 seconds. The separation result is shown in Fig. 4.27f. When reaching the similar training error level, the proposed ErrCor algorithm can solve the two-spiral problem with only 30 RBF units (as shown in Fig. 4.27e) and the average time cost is 56.734 seconds.

4.2.2 Duplicate Patterns Test

The ability to handle training a dataset with duplicated patterns is tested by the conducted experiment. The training dataset comes from the two-spiral problem in the last section, but they are modified by adding 10 patterns which are randomly selected from the original 194 patterns. Therefore, there are 204 patterns in the modified two-spiral dataset, including 10 duplicated patterns.

In order to test the ability of the ErrCor algorithm to train dataset with duplicated patterns, the experiment is arranged in two cases for comparison.

- Case 1: apply original two-spiral dataset (without duplication) for training
- Case 2: apply the modified two-spiral dataset (with 10 duplicated patterns) for training

Fig. 4.28 shows the network construction process using the ErrCor algorithm for the two cases.

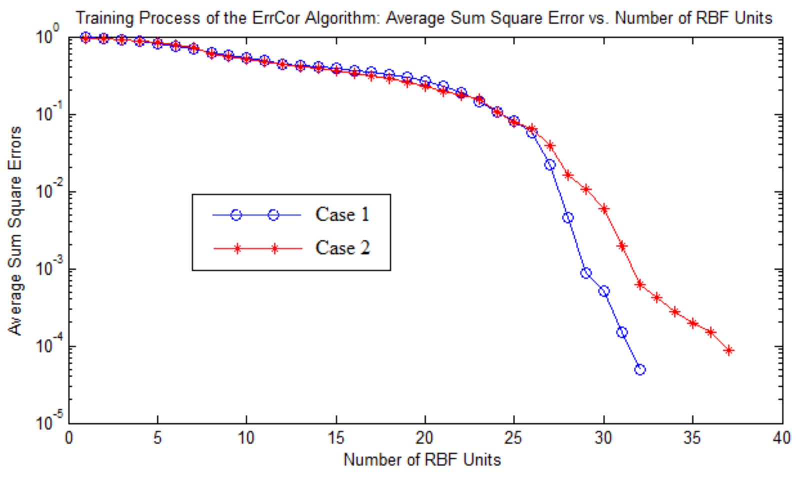
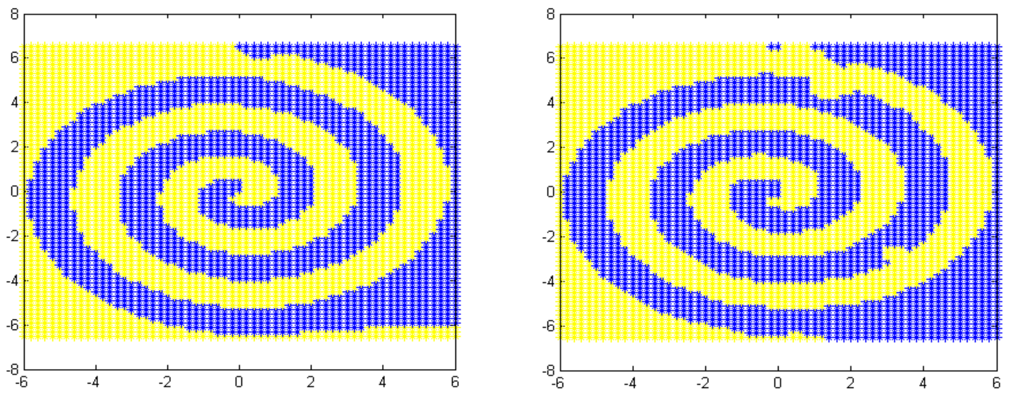


Fig. 4.28 Training process of the ErrCor algorithm using different dataset

From the experimental results in Fig. 4.28, it can be noticed that the ErrCor algorithm can handle the dataset with duplicated patterns correctly. In order to reach the similar desired training level (average sum square error=0.0001), 5 more RBF units are required for network construction using the modified two-spiral dataset.

Fig. 4.29 shows the generalization results of the designed RBF networks in the two cases.



(a) (b)

Fig. 4.29 Generalization results of the two trained RBF networks by different dataset: (a) case 1: original two-spiral dataset; (b) case 2: modified two-spiral dataset

One may notice that the RBF network designed by the modified dataset (with 10 duplicated patterns) gets slightly worse generalization results (Fig.4.29b) than that obtained by the original dataset (without duplicated pattern) in Fig. 4.29a.

4.2.3 Noise Patterns Test

The image recognition problem is applied to test the ability of the proposed ErrCor algorithm to handle the training datasets contaminated by noise. Fig.4.30a shows the original images, each of which consists of 7×8=56 pixels. The color of pixels is scaled by the Jet degree: from -1 (blue) to 1 (red). Fig.4-30b shows the 6 groups of noised images. Each group consists of 20 images with the same noise level. The noised images are generated by the formula:

$$NP_i = P_0 + i \times \delta \tag{4-2}$$

where: NP_i are the noised image data in Fig. 4-30b; P_0 is the original image data in Fig. 4.30a; i is the noise level from 1 to 6; δ is the randomly generated noise in range [-0.25, 0.25].

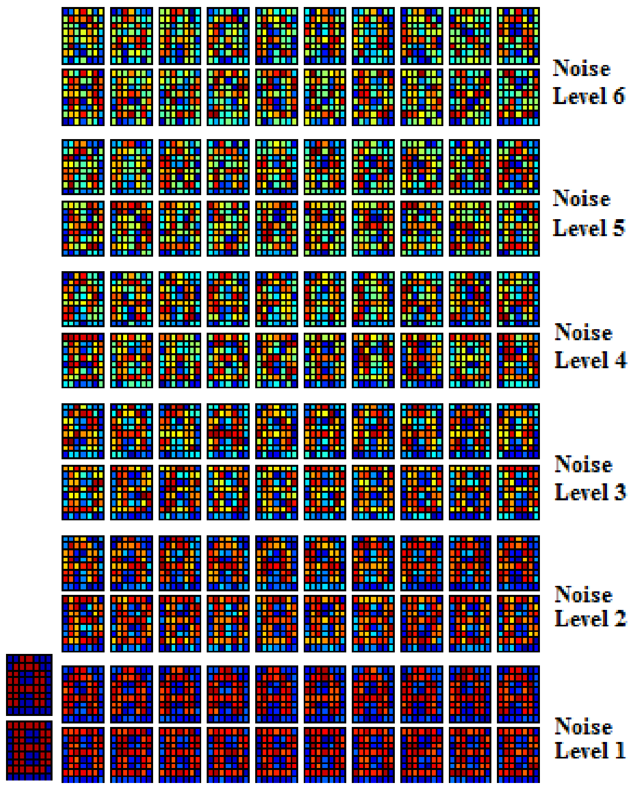


Fig. 4.30 Dataset of the noised patterns test: (a) original images; (b) noised images

In the experiment the ErrCor algorithm is applied to train the 6 groups of noised images separately. The desired training sum square error (SSE) is set as 0.0001. Fig.4.31 presents the minimum sizes of RBF networks required for convergence.

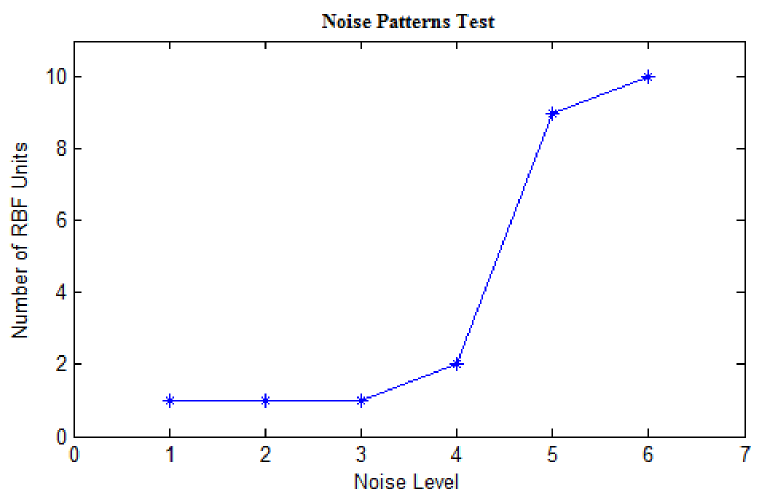


Fig. 4.31 Noise patterns test results: noise level vs. the number of RBF units cost required for convergence

Based on the results shown in Fig.4.31, it can be noticed that as the noise level increases, more RBF units are required to reach the desired training error. When the noise range (calculated by $i \times \delta$ in (4-2)) is no more than the original data range [-1, 1] (the noise level is less than or equal to 4), ErrCor algorithm illustrates its good tolerance to noise data because almost the same size (1 or 2 RBF units) of RBF networks can be designed to solve the problem.

4.3 Conclusion

In this chapter, we present an error correction (ErrCor) algorithm to perform incremental design of RBF networks. During the training process, the number of RBF units is increased one by one until the training evaluation reaches desired accuracy. The initial center of the newly added RBF unit is properly selected based on the location of highest peak/lowest valley in error surface; while for the other RBF units, the initial conditions are copied from the training results of the last step. Parameter adjustments, including weights, centers, and widths are performed by the Levenberg Marquardt algorithm. Taking the advantages of second order algorithms, the proposed algorithm converges fast in each step of building the desired RBF networks. The experiment results show the network efficiency and robustness of the proposed ErrCor algorithm.

Chapter 5

Conclusions

This dissertation is dedicated to the development of new algorithms to construct RBF networks. The improved second-order algorithm was proposed to apply in RBF training process. It not only used the second-order algorithm, but also introduced all the parameters, including input weights, output weights, centers and widths to be adjusted. The ISO algorithm exhibits good search ability, fast convergence, and high network efficiency. The proposed ErrCor algorithm which performed an incremental design of RBF network solved the problem of deciding the number of RBF units and initial parameters of kernel function in RBF network. The algorithm shows higher network efficiency and robustness compared with other algorithms in the references. By combining these two algorithms together, the desired RBF network with proper size and initial conditions could be successfully constructed.

REFERENCES

[1] T. Hrycej, “Modular Learning in Neural Networks: A Modularized Approach to Neural Network Classification”, John Wiley&Sons, 1992

[2] P.Yohhan,”Adaptive pattern recognition and neural networks”, Reading, Addison-Wesley Publishing, 1989.

[3] T.H. Smith, “A self-tuning EWMA controller utilizing artificial neural network function approximation techniques”, IEEE Transactions on Components, Packaging and Manufacturing Technology, Part C, Vol 20(2), pp. 121-132, 1997.

[4] D. Wang, J. Huang, “Neural network-based adaptive dynamic surface control for a class of uncertain nonlinear systems in strict-feedback form”, IEEE Transactions on Neural Networks, Vol 16(1), pp.195-202, 2005.

[5] M. Bratislav, A. Marija, S, Zoran, D. Nebojsa and S. Maja, “Application of neural networks in spatial signal processing”, 11th Symposium on Neural Network Applications in Electrical Engineering, pp. 5-14, 2012.

[6] P.Goel, G. Dedeoglu, S.I. Roumeliotis and G.S. Sukhateme, “Fault detection and identification in a mobile robot using multiple model estimation and neural network”, IEEE conference on Robotics and Automation, Vol 3, pp. 2302-2309, 2000.

[7] T.M. Cover, “Geometrical and Statistical properties of systems of linear inequalities with

applications in pattern recognition”, IEEE Transactions on Electronic Computers,pp: 356-334, 1965.

[8] J. Moody and C. Darken D. Touretzky , G. Hinten and T. Sejnowski "Learning with localized receptive fields", Proc. 1988 Connectionist Models Summer School, 1988.

[9] S. Chen, C. F. N. Cowan, and P. M. Grant, "Orthogonal least squares learning algorithm for radial basis function networks", IEEE Trans. Neural Networks, vol. 2, pp.302 -309, 1991.

[10] D. Wettschereck and T. Dietterich, J. E. Moody, S. J. Hanson, and R. P. Lippmann, "Improving the performance of radial basis function networks by learning center locations", Advances in Neural Information Processing Systems 4, pp.1133 -1140 1992.

[11] N. B. Karayiannis and G. W. Mi, "Growing radial basis neural networks: Merging supervised and unsupervised learning with network growth techniques", IEEE Trans. Neural Networks, vol. 8, pp.1492 -1506, 1997.

[12] Huang, G.-B., Saratchandran, P. and Sundararajan, N.: An efficient sequential learning algorithm for Growing and Pruning RBF (GAP-RBF) networks, IEEE Transactions on Systems, Man and Cybernetics-Part B: Cybernetics, Vol 34(6), pp.2284–2292, 2004.

[13]S.I. Ch’ng, K.P.Seng and L.M. Ang, “Adaptive momentum Levenberg-Marquardt RBF for face recognition”, 2012 IEEE International Conference on Circuits and Systems, pp.126-131,2012.

[14] K. Meng, Z. Y. Dong, D. H. Wang and K. P. Wong, "A Self-Adaptive RBF Neural Network Classifier for Transformer Fault Analysis," IEEE Trans. on Power Systems, vol. 25, issue 3, pp. 1350-1360, 2010.

[15] S. Huang and K. K. Tan, "Fault Detection and Diagnosis Based on Modeling and Estimation Methods," IEEE Trans. on Neural Networks, vol. 20, issue 5, pp. 872-881, 2009.

[16] L. Cai, A. B. Rad and W. L. Chan, "An Intelligent Longitudinal Controller for Application in Semiautonomous Vehicles," IEEE Trans. on Industrial Electronics, vol. 57, no. 4, pp. 1487-1497, 2010.

[17] C. C. Tsai, H. C. Huang and S. C. Lin, "Adaptive Neural Network Control of a Self-Balancing Two-Wheeled Scooter," IEEE Trans. on Industrial Electronics, vol. 57, no. 4, pp. 1420-1428, 2010.

[18] L. Cai, A. B. Rad and W. L. Chan, "An Intelligent Longitudinal Controller for Application in Semiautonomous Vehicles," IEEE Trans. on Industrial Electronics, vol. 57, no. 4, pp. 1487-1497, 2010.

[19] C. C. Tsai, H. C. Huang and S. C. Lin, "Adaptive Neural Network Control of a Self-Balancing Two-Wheeled Scooter," IEEE Trans. on Industrial Electronics, vol. 57, no. 4, pp. 1420-1428, 2010.

[20] K. B. Cho and B. H. Wang, "Radial basis function based adaptive fuzzy systems and their applications to system identification and prediction", Fuzzy Sets Syst., vol. 83, pp.325 -339, 1996.

[21] J. Platt, "A resource-allocating network for function interpolation", Neural Computa., vol. 3, pp.213 -225 1991.

[22] V. Kadirkanianathan and M. Niraujan, “A function estimation approach to sequential learning with neural networks”, Neural Computation, Vol 5(6):954–975, 1993.

[23] L. Yingwei, N. Sundararajan, and P. Saratchandran, “A sequential learning scheine for function approximation using minimal radial basis function neural networks”, Neural Computations, Vol 9 (2) pp.461–478, 1997.

[24] G.-B. Huang , P. Saratchandran and N. Sundararajan "A generalized growing and pruning RBF (GGAP-RBF) neural network for function approximation", IEEE Trans. Neural Networks, vol. 16, no. 1, pp.57 -67 2005.

[25] M. Bortman and M. Aladjem, "A growing and pruning method for radial basis function networks", IEEE Trans. Neural Networks, vol. 20, no. 6, pp.1039 -1045 2009.

[26] H. Yu and B. M. Wilamowski, "Fast and efficient and training of neural networks," in Proc. 3nd IEEE Human System Interaction Conf. HSI 2010, Rzeszow, Poland, May 13-15, pp. 175-181 2010.

[27] B. M. Wilamowski, “Neural Network Architectures and Learning Algorithms: How Not to Be Frustrated with Neural Networks,” IEEE Industrial Electronics Magazine, vol. 3, no. 4, pp. 56-63, Dec. 2009.

[28] B. M. Wilamowski and H. Yu, "Neural Network Learning Without Backpropagation," IEEE Trans. on Neural Networks, vol. 21, no.11, pp. 1793-1803, Nov. 2010.

[29] L. Tarassenko and S. Roberts, "Supervised and unsupervised learning in radial basis function classifiers", Proc. Inst. Elect. Eng.-Visual Image Signal Processing, vol. 141, no. 4, pp.210 -216 1994.

[30] B. M. Wilamowski and H. Yu, "Improved Computation for Levenberg Marquardt Training," IEEE Trans. on Neural Networks, vol. 21, no. 6, pp. 930-937, June 2010.

[31] D.W. Marquardt, “An algorithm for least squares estimation of nonlinear parameters”, SIAM J. Appl. Math. 11, pp. 431–441, 1963.

[32] P.T. Harker and B. Xiao, “Newton's method for the nonlinear complementarity problem: A B-differentiable equation approach,” Mathematical Programming, Vol 48 pp:339–357, 1990.

[33] C. Blake and C. Merz, UCI Repository of Machine Learning Databases, Dept. Inform. Comput. Sci., Univ. California, Irvine, 1998.

[34] H. Yu and B. M. Wilamowski, "Efficient and Reliable Training of Neural Networks," in Proc. 2nd IEEE Human System Interaction Conf. HSI 2009, Catania, Italy, pp. 109-115, 2009.

[35] H. Yu and B. M. Wilamowski, “Levenberg–Marquardt Training” Industrial Electronics Handbook, vol. 5 – Intelligent Systems, 2nd Edition, chapter 12, pp. 12-1 to 12-15, CRC Press 2011.

[36] N. Chaiyaratana and A. M. S. Zalzala, "Evolving Hybrid RBF-MLP Networks Using Combined Genetic/Unsupervised/Supervised Learning," UKACC International Conference on Control '98, vol. 1, pp. 330-335, Swansea, UK, Sep. 01-04, 1998.

[37] W. Kaminski and P. Strumillo, "Kernel Orthonormalization in Radial Basis Function Neural Networks," IEEE Trans. on Neural Networks, vol. 8, no. 5, pp. 1177-1183, Sep. 1997.

[38] Neruda and P. Kudová, "Learning Methods for Radial Basis Function Networks," Future Generation Computer Systems, vol. 21, issue. 7, pp. 1131-1142, , July 2005.

APPENDIX

Appendix 1: Improved Second Order Algorithm Implementation of Training RBF networks

```
function [a1, a2, a3] = ISO method
clear all; format long;
%% generate training patterns
[x1,y1,z1]=peaks(10);
x1 = x1(1,:);
y1 = y1(:,1);
for i = 1:length(x1)
        for j = 1:length(y1)
                inputs((i-1)*length(y1)+j,1)=x1(i);
                inputs((i-1)*length(y1)+j,2)=y1(j);
                outputs((i-1)*length(y1)+j,1)=z1(i,j);
        end;
end;
[m,n] = size(inputs);
figure(1);clf;
surf(x1,y1,z1);
%% set the number of RBF units
number_of_hidden_unit = 10;
%% initial parameter generation
[weights_input, weights_output, widths, centers] =
generate_initial_parameters(number_of_hidden_unit,inputs);
%% combination of parameters
para_cur = parameter_combination(weights_output, widths, weights_input, centers);
%% other parameters
I = eye(length(para_cur));
maximum_iteration = 300;
maximum_error = 0.001;
mu = 0.01;
%% training process
[SSE(1)] = calculate_SSE(weights_input, weights_output,widths,centers,inputs,outputs);
fprintf('iteration = 1, SSE = %6.10f\n',SSE(1));
tic
for iter = 2:maximum_iteration
        jw = 0;
        [gradient, hessian] = calculate_gradient(weights_input, weights_output, widths, centers,
inputs, outputs );
        para_back = para_cur;
        while 1
                para_cur = para_back - ((hessian+mu*I)\gradient')';
                [weights_output, widths, weights_input, centers] =
parameter_divison(para_cur,number_of_hidden_unit,inputs);
                [SSE(iter)] = calculate_SSE(weights_input,
weights_output,widths,centers,inputs,outputs);
                if SSE(iter) <= SSE(iter-1)
                        if mu > 10^-20;
                                mu = mu/10;
                        end;
                        break;
                end;
                if mu < 10^20
                        mu = mu*10;
                end;
                jw = jw + 1;
                if jw > 5
                        break;
                end;
        end;
        fprintf('iteration = %d, SSE = %6.10f\n',iter, SSE(iter));
        if SSE(iter) < maximum_error
                break;
        end;
end;
%% plot the error curve
SSE(iter)
figure(2);clf;
loglog(1:iter,SSE);
%% plot the test patterns
[x1_,y1_,z1_]=peaks(30);
x1_ = x1_(1,:);
y1_ = y1_(:,1);
```

```
for i = 1:length(x1_)
        for j = 1:length(y1_)
                inputs_((i-1)*length(y1_)+j,1)=x1_(i);
                inputs_((i-1)*length(y1_)+j,2)=y1_(j);
        end;
end;
output_ = verification(weights_input, weights_output, widths, centers, inputs_);
for i = 1:length(x1_)
        for j = 1:length(y1_)
                test_output(i,j) = output_((i-1)*length(y1_)+j);
        end;
end;
figure(3);clf;
surf(x1_,y1_,test_output);
a1 = 1; a2 = 1; a3 = 1;
%% gradient computation
function [gradient, hessian] = calculate_gradient(ww, weights, widths, centers, inputs, outputs)
        [p1,p2] = size(weights);
        [p3,p4] = size(centers);
        [p5,p6] = size(widths);
        [p7,p8] = size(ww);
        g_weight = zeros(p1,p2);
        g_center = zeros(p3,p4);
        g_width = zeros(p5,p6);
        g_ww = zeros(p7,p8);
        gradient = zeros(1,p1*p2+p3*p4+p5*p6+p7*p8);
        hessian = zeros(p1*p2+p3*p4+p5*p6+p7*p8,p1*p2+p3*p4+p5*p6+p7*p8);
%          gradient  = zeros(1,p1*p2+p3*p4+p7*p8);
%          hessian  = zeros(p1*p2+p3*p4+p7*p8,p1*p2+p3*p4+p7*p8);
        [m,n] = size(inputs);
        for i = 1:m
                net = weights(1);
                for j = 1:p3
                        node(j) = exp(-sum((ww(j,:).*inputs(i,:)-centers(j,:)).^2)/widths(j));
                        net = net + node(j)*weights(j+1);
                end;
```

```
            % for g_weight
            out = net;
            de = 1;
            err = outputs(i,1) - out;
            J_weight(1) = -de;
            for j = 2:p2
                J_weight(j) = J_weight(1)*node(j-1);
            end;
            % for g_center
            for j = 1:p3
 J_center(j,:) = (-1)*weights(j+1)*node(j)*2*(ww(j,:).*inputs(i,:)-centers(j,:))./widths(j);
 J_width(j) = (-1)*weights(j+1)*node(j)*sum((ww(j,:).*inputs(i,:)-centers(j,:)).^2)/widths(j)^2;
                for k = 1:n
                  J_ww(j,k) =
(-1)*weights(j+1)*node(j)*(-1)/widths(j)*2*(ww(j,k)*inputs(i,k)-centers(j,k))*inputs(i,k);
                end;
            end;
            J = parameter_combination(J_weight, J_width, J_ww, J_center);
            gradient = gradient + err*J;
            hessian = hessian + J'*J;
        end;
%% error computation
function [SSE] = calculate_SSE(ww, weights,widths,centers,inputs,outputs)
        [m,n] = size(inputs);
        [p,q] = size(centers);
        SSE = 0;
        for i = 1:m
            count = weights(1);
            for j = 1:p
                count = count +
weights(j+1)*exp(-sum((ww(j,:).*inputs(i,:)-centers(j,:)).^2)/widths(j));
            end;
            SSE = SSE + (count - outputs(i,1))^2;
        end;
        SSE = sqrt(SSE/m);
%% generate weights
```

```
function [weights_input, weights_output, widths, centers] =
generate_initial_parameters(num,data)
        [row,col] = size(data);
        weights_input = ones(num,col);
        weights_output = ones(1,num+1);
        widths = ones(1,num);
        for i = 1:num
                ind(i) = mod(floor(20000*rand(1)),row)+1;
                if i > 2
                        while 1
                                flag = 0;
                                for j = 1:(i-1)
                                        if ind(i) == ind(j)
                                                flag = 1;
                                                break;
                                        end;
                                end;
                                if flag == 0
                                        break;
                                end;
                                ind(i) = mod(floor(20000*rand(1)),row)+1;
                        end;
                end;
                centers(i,:) = data(ind(i),:);
        end;
%% parameter combination
function [vector] = parameter_combination(weights_output, widths, weights_input, centers)
        [p1,p2] = size(weights_input);
        [p3,p4] = size(centers);
        vector = [weights_output widths reshape(weights_input',1,p1*p2)
reshape(centers',1,p3*p4)];
%       vector = [weights_output reshape(weights_input',1,p1*p2) reshape(centers',1,p3*p4)];
%% parameter division
function [weights_output, widths, weights_input, centers] = parameter_divison(vector, num,
data)
        [row, col] = size(data);
        for i = 1:(num+1)
                weights_output(1,i) = vector(1,i);
        end;
        for i = 1:num
                widths(1,i) = vector(1, num+1+i);
        end;
        for i = 1:num
                for j = 1:col
                        weights_input(i,j) = vector(1,2*num+1+(i-1)*col+j);
%                            weights_input(i,j)  = vector(1,num+1+(i-1)*col+j);
                end;
        end;
        for i = 1:num
                for j = 1:col
                        centers(i,j) = vector(1,2*num+1+num*col+(i-1)*col+j);
%                            centers(i,j)  = vector(1,num+1+num*col+(i-1)*col+j);
                end;
        end;
%          widths  = calculate_width(centers);
%% verification process
function [output] = verification(weights_input, weights_output, widths, centers, testing_input)
        [m,n] = size(testing_input);
        [p,q] = size(centers);
        for i = 1:m
                count = weights_output(1);
                for j = 1:p
                        count = count +
weights_output(j+1)*exp(-sum((weights_input(j,:).*testing_input(i,:)-centers(j,:)).^2)/widths(j));
                end;
                output(i,1) = count;
        end;
%% calculate width
function [widths] = calculate_width(centers)
        [m,n] = size(centers);
        for i = 1:m
                d = 0;
```

```
            for j = 1:m
                d = d + sum((centers(j,:)-centers(i,:)).^2);
            end;
            widths(1,i) = sqrt(d)/m;
        end;
```

Appendix 2: Implementation of Error Correction Algorithm of RBF network

```
function [a1, a2, a3] = ISO_Training
clear all; format long;

x1=0:0.1:5;
for i = 1:length(x1)
  inputs(i,1)=x1(i);
  outputs(i,1)=0.8*exp(-x1(i)/2)*sin(3*x1(i));
end;

[m,n] = size(inputs);
actual_output_ = zeros(size(outputs));

centers = [];
weights_input = [];
weights_output = [1];
widths = [];
number_of_hidden_unit = 0;

for kkk = 1:10
        SSE = [];
        [maxi_, index_1] = max(abs(outputs-actual_output_));
        number_of_hidden_unit = number_of_hidden_unit + 1;
        centers = [centers; inputs(index_1,:)];
        weights_input = [weights_input; ones(1,n)];
        weights_output = [weights_output, 1];
        widths = [widths, 1];
        para_cur = parameter_combination(weights_output, widths, centers);
        % para_cur = weights_output;
        I = eye(length(para_cur));
        % other parameters
        maximum_iteration = 100;
        maximum_error = 0.00001;
        mu = 0.01;
        % training process
        [SSE(1)] = calculate_SSE(weights_input, weights_output,widths,centers,inputs,outputs);
        fprintf('Number of RBF units = %d, iteration = 1, SSE = %6.10f\n',kkk,SSE(1));
        for iter = 2:maximum_iteration
                jw = 0;
                [gradient, hessian] = calculate_gradient(weights_input, weights_output, widths, centers,
inputs, outputs );
                para_back = para_cur;
                while 1
                        para_cur = para_back - (inv(hessian+mu*I)*gradient')';
                        [weights_output, widths, centers] =
parameter_divison(para_cur,number_of_hidden_unit,inputs)
                        [SSE(iter)] = calculate_SSE(weights_input,
weights_output,widths,centers,inputs,outputs);
                        if SSE(iter) <= SSE(iter-1)
                                if mu > 10^-20;
                                        mu = mu/10;
                                end;
                                break;
                        end;
                        if mu < 10^20
                                mu = mu*10;
                        end;
                        jw = jw + 1;
                        if jw > 5
                                break;
                        end;
                end;
                fprintf('Number of RBF units = %d, iteration = %d, SSE = %6.10f\n',kkk,iter,
SSE(iter));
                if abs(SSE(iter-1)-SSE(iter)) < 0.0000000001
                        break;
```

```
            end;
            if SSE(iter) < maximum_error
                break;
            end;
        end;
          [actual_] = verification(weights_input, weights_output, widths, centers, inputs);
           figure(5);
          plot(x1,actual_,'ro');
           axis([0 5 -1 1]);
           figure(6);
           plot(x1, outputs-actual_,'ro');
           axis([0 5 -1 1]);

        if SSE(iter) < maximum_error
            break;
        end;

        [actual_output_] = verification(weights_input, weights_output, widths, centers, inputs);

          pause(2);
end;
a1 = 1; a2 = 1; a3 = 1;
%% gradient computation
function [gradient, hessian] = calculate_gradient(ww, weights, widths, centers, inputs, outputs)
        [p1,p2] = size(weights);
        [p3,p4] = size(centers);
        [p5,p6] = size(widths);
        [p7,p8] = size(ww);
        g_weight = zeros(p1,p2);
        g_center = zeros(p3,p4);
        g_width = zeros(p5,p6);
        g_ww = zeros(p7,p8);
        gradient = zeros(1,p1*p2+p3*p4+p5*p6);
        hessian = zeros(p1*p2+p3*p4+p5*p6,p1*p2+p3*p4+p5*p6);
%       gradient  = zeros(1,p1*p2+p3*p4+p5*p6+p7*p8);
%       hessian  = zeros(p1*p2+p3*p4+p5*p6+p7*p8,p1*p2+p3*p4+p5*p6+p7*p8);
```

```
    [m,n] = size(inputs);
    for i = 1:m
        net = weights(1);
        for j = 1:p3
            node(j) = exp(-sum((ww(j,:).*inputs(i,:)-centers(j,:)).^2)/widths(j));
            net = net + node(j)*weights(j+1);
        end;
        % for g_weight
        out = net;
        de = 1;
        err = outputs(i,1) - out;
        J_weight(1) = -de;
        for j = 2:p2
            J_weight(j) = J_weight(1)*node(j-1);
        end;
        % for g_center
        for j = 1:p3
            J_center(j,:) =
(-1)*weights(j+1)*node(j)*2*(ww(j,:).*inputs(i,:)-centers(j,:))./widths(j);
            J_width(j) =
(-1)*weights(j+1)*node(j)*sum((ww(j,:).*inputs(i,:)-centers(j,:)).^2)/widths(j)^2;
%              for  k  =1n
%                  J_ww(j,k)  =
(-1)*weights(j+1)*node(j)*(-1)/widths(j)*2*(ww(j,k)*inputs(i,k)-centers(j,k))*inputs(i,k);
%              end;
        end;
        J = parameter_combination(J_weight, J_width, J_center);
%          J  = parameter_combination(J_weight,  J_width,  J_ww,  J_center);
        gradient = gradient + err*J;
        hessian = hessian + J'*J;
    end;
%% error computation
function [SSE] = calculate_SSE(ww, weights,widths,centers,inputs,outputs)
    [m,n] = size(inputs);
    [p,q] = size(centers);
    SSE = 0;
```

```
        for i = 1:m
                count = weights(1);
                for j = 1:p
                        count = count +
weights(j+1)*exp(-sum((ww(j,:).*inputs(i,:)-centers(j,:)).^2)/widths(j));
                end;
                SSE = SSE + (count - outputs(i,1))^2;
        end;
        SSE = SSE/m;
%% parameter combination
% function [vector] = parameter_combination(weights_output, widths, weights_input, centers)
function [vector] = parameter_combination(weights_output, widths, centers)
        [p3,p4] = size(centers);
        vector = [weights_output widths reshape(centers',1,p3*p4)];
%       [p1,p2] = size(weights_input);
%       [p3,p4] = size(centers);
%       vector = [weights_output widths reshape(weights_input',1,p1*p2)
reshape(centers',1,p3*p4)];
%% parameter division
function [weights_output, widths, centers] = parameter_divison(vector, num, data)
% function [weights_output, widths, weights_input, centers] = parameter_divison(vector, num,
data)
        [row, col] = size(data);
        for i = 1:(num+1)
                weights_output(1,i) = vector(1,i);
        end;
        for i = 1:num
                widths(1,i) = vector(1, num+1+i);
        end;
        for i = 1:num
                for j = 1:col
                        centers(i,j) = vector(1,2*num+1+(i-1)*col+j);
                end;
        end;
%% verification process
function [output] = verification(weights_input, weights_output, widths, centers, testing_input)
```

```
    [m,n] = size(testing_input);
    [p,q] = size(centers);
    for i = 1:m
        count = weights_output(1);
        for j = 1:p
            count = count +
weights_output(j+1)*exp(-sum((weights_input(j,:).*testing_input(i,:)-centers(j,:)).^2)/widths(j));
        end;
        output(i,1) = count;
    end;
```