FACULTEIT
TOEGEPASTE WETENSCHAPPEN

DEPARTEMENT
ELEKTROTECHNIEK – ESAT



KATHOLIEKE
UNIVERSITEIT
LEUVEN

The Generalised Mapping Regressor (GMR) neural network for inverse discontinuous problems

Thesis submitted to obtain the degree of
Master of Artificial Intelligence

chuan LU

Supervisor :
Prof. Dr. Ir. Sabine Van Huffel

1999 - 2000

Faculteiten Wetenschappen K.U.Leuven
en Toegepaste Wetenschappen Academiejaar 1999-2000

Student Name: Chuan LU

Title:

**The Generalised Mapping Regressor (GMR)
neural network for inverse discontinuous problems**

Abstract:

Feedforward neural networks are known to be universal approximators of nonlinear functions. Although for many functional (i.e. single-valued) approximation problems feedforward network can work well by minimizing a sum-of-squares error function, problems can arise when the least squares approach is applied to inverse and discontinuous problems, in which mapping is often multi-valued, or has structure varying for different regions of the input space.

The Generalised Mapping Regressor (GMR), as recently introduced by G. Cirrincione and M. Cirrincione, is a new method for solving these generalised mapping problems. The basic principle is the transformation of the function approximation problem into a pattern recognition problem under an unsupervised framework. The prominent feature of GMR are its capability to output all the solutions and their corresponding mapping branches and, if the case, the equilevel surfaces.

This thesis is to deal with implementation of the GMR algorithm. The branch and bound search technique is then applied to speed up its computation. The properties of GMR are further explored by means of experiments. The results show that GMR is powerful in high level modelling. Its results can be even more important if interpolation techniques are used; however, this is outside the scope of the thesis because of limits of space.

Thesis submitted to obtain
the degree of Master of Artificial Intelligence

Promotor: Prof. Dr. Ir. Sabine Van Huffel
Daily Supervisor: Dr. Ir. Giansalvo Cirrincione

Acknowledgements

Firstly, I wish to express my gratitude to my daily supervisor, Dr. Giansalvo Cirrincione. He generously spent so much time in explaining the knowledge to me in a vivid way with his enthusiasm and good humor, reviewing my manuscripts with great patience, and providing advice for my work.

I am grateful to my promotor, Prof. Sabine Van Huffel for her giving me introduction to this thesis, her valuable advice and kind help.

Finally, I would like to thank my family and friends for their continuous support. Many thanks go to my sister Yang for her great help and inspirational talks. Special thanks to my parents and my husband Shigang, for their endless encouragement and understanding.

Table of Contents

Abstract i
Acknowledgement ii
Table of Contents iii
List of Figures v
List of Tables vi

1 Introduction **1**

2 Neural Approaches to Function Approximation Problem **3**
2.1 Function Approximation 3
2.2 Neural Networks for Inverse and Discontinuous Problems - Modelling Conditional Distributions 4
2.2.1 Multi-layer Perceptrons 6
2.2.2 Mixture-of-experts 7

3 The EXIN Segmentation Neural Network **9**
3.1 The Progressive Learning Network (PLN) 9
3.2 The EXIN Segmentation Neural Network (EXIN SNN) 10
3.2.1 The Learning Algorithm 10
3.2.2 The Cleaning Algorithm 12

4 The Generalised Mapping Regressor Algorithm **13**
4.1 The Basic Ideas 14
4.2 The Algorithm 15
4.2.1 Learning (coarse-to-fine vector quantization) 15
4.2.2 Linking 16
4.2.3 Object Merging 19
4.2.4 Recalling 19
4.3 Summary 22

5 Using Branch and Bound to Accelerate GMR Computation **24**
5.1 Basic Branch and Bound Search 24
5.2 Using Branch and Bound in GMR 27
5.2.1 Implementation of Branch and Bound for Linking 27
5.2.1.1 Creation of the Neuron Tree During Learning Phase 28
5.2.1.2 Using Branch and Bound Search in Linking Phase 28

5.2.1.3 Experimental Results..... 30
5.2.2 Implementation of Branch and Bound in Learning..... 34
5.2.2.1 Using Branch and Bound in Relabelling..... 34
5.2.2.2 Experimental Results..... 35
5.3 Conclusions..... 36

6 Experiments..... 37
6.1 Learning..... 37
6.1.1 Stopping Criteria..... 37
6.1.2 Vigilance Threshold in Single Level Learning..... 40
6.1.3 Multilevel Learning..... 41
6.1.4 Neuron Cleaning..... 43
6.2 Linking and Merging..... 44
6.2.1 Linking Factor and Merging Threshold..... 44
6.2.2 Domain Factor..... 46
6.3 Sparse Region Approximation..... 47
6.4 Mapping of Noisy Training Set..... 49
6.5 Other Simulations for Mapping Approximation..... 51
6.6 Summary..... 55

7 Conclusions..... 56

References **59**

List of Figures

2.1	Structure of mixture density networks…….....	5
2.2	Examples of using MLP to model conditional distributions.....	6
2.3	Architecture of the mixture-of-experts modular network……	7
3.1	The Counterpropagation network (CPN) clustering.....	9
3.2	The Progressive Learning Network (PLN).....	10
3.3	Example of EXIN SNN learning.....	11
4.1	Illustration of a global mapper.....……..	13
4.2	Schematic illustration of the four phases in GMR.....……	14
4.3	Coarse to fine learning architecture in GMR.....……	16
4.4	Examples of linking step…….....	18
4.5	An illustrative example of using GMR for mapping approximation.....	21
5.1	Neuron tree structure built during learning phase of GMR.....	25
5.2	Branch and bound search rules.....	26
5.3	Examples of linking mistakes that appears when using δ-BnB and k-BnB in candidate steps…….....	29
5.4	Datasets, resulting neurons and linkings of experiments.....	31
6.1	Mapping result of a single level EXIN SNN learning.....	39
6.2	Mapping result of different vigilance threshold settings (fixed vs. decaying)…….....	41
6.3	Comparison of different mapping accuracies and computation time of different levels learning.....	43
6.4	Comparison of the linking and recalling results under different δ settings..	45
6.5	Comparison of recalling result with different ρ_f and k_{min} settings.....	46
6.6	Approximation of a function with sparse region.....	47
6.7	Comparison of mapping from data set with different noisy level.....	50
6.8	GMR gives a mapping from noisy TS (σ=0.15).....	50
6.9	Example of GMR approximation.....	51
6.10	Example of GMR approximation.....	52
6.11	GMR mapping of 8 sphere in a 3-D scene.....	53

List of Tables

4.1 The four phases of GMR network..... 22
5.1 Using branch and bound in GMR linking..... 32
5.2 Construct neuron tree structure for applying BnB in labelling by *k*-means / EXIN SNN clustering..... 35
6.1 Comparison of two stopping criteia in EXIN SNN: fixed epoch vs. ∇-stop..... 38
6.2 Mapping results with different vigilance threshold settings..... 40
6.3 Comparison of different levels EXIN SNN's learning..... 41
6.4 Linking and merging results of different δ settings..... 44
6.5 Mapping from data set with different noisy level..... 49

Chapter 1

Introduction

Feedforward neural networks are known to be universal approximators of nonlinear functions: any continuous multivariate function can be approximated by a single hidden layer feedforward neural network uniformly to any degree of accuracy, provided a sufficient number of hidden units is given. Rigorous proofs for the universality of feedforward neural networks employing continuous sigmoidal activation function have been given [5]. Moreover, the neural networks' ability to learn any relationship, however complex it is, has already been established in Kolmogorov's Theorem (1957) [6], [7].

In practical applications, many neural network applications fall to solve the corresponding inverse problems. Examples include the analysis of the spectral data, tomographic reconstruction, control of industrial plant, and robot kinematics. For such problems there exists a well-defined forward problem which characterized by a functional (i.e. single-valued) mapping. Often this corresponds to causality in a physical system. In the case of spectral reconstruction, for example, the forward problem corresponds to the evaluation of the spectrum when parameters (location, widths and amplitudes) of the spectral lines are prescribed. In practical applications we generally have to solve the corresponding inverse problem in which the roles of input and output variables are interchanged. In case of spectral analysis, this corresponds to the determination of the spectral line parameters from an observed spectrum. For inverse problems, the mapping can often be multi-valued, with values of the inputs for which there are several valid values for the outputs. For example, there may be several choices for the spectral line parameters which give rise to the same observed spectrum.

Although for many functional approximation problems feedforward network can work very well by minimizing a sum-of-squares error function, problems can arise when the least squares approach is applied to an inverse problem. As the least squares solution approximates the conditional average of the target data, this will frequently lead to extremely poor performance (since the average of several solutions is not necessarily itself a solution) [4]. As a fundamental consequence of using a sum-of-squares error function, modifying this kind of network architecture or the training algorithm cannot solve this problem. For those problems with high dimensional data,

visualization of which is not straightforward; it can be very difficult to locate the regions of input space for which the target data is multi-valued.

There exist some neural networks that can solve the inverse problem. For example, the neural networks with two hidden layers and threshold units can approximate the one-sided inverse of a continuous function [8]. The symmetric counterpropagation network [6] can also solve this inversion problem. The other interesting networks to solve this problem include some incremental radial basis neural networks for the function approximation such as the resource-allocating neural network (RAN) by Platt [9] and its improvement minimal RAN (M-RAN) [10] which uses a pruning strategy for its units.

The problem of discontinuities involves the mapping of which structure varies for different regions of the input space. Mixture-of-experts (ME) [11] architectures decompose the input space into different regions, assign an "expert" network to each of these different regions and then use a "gating" network to decide which experts should be used to determine the output. Hence ME is suitable for discontinuous problems.

Bishop [4] has provided a comprehensive description to these problems: he suggests going beyond the Gaussian description of the distribution of target variables, to find a more general model for the conditional density. This will be one of the topics of Chapter 2.

More recently, G.Cirrincione and M.Cirrincione (1998) [1], [2], [3] proposed the Generalised Mapping Regressor (GMR), which is an incremental self-organizing neural network with adaptive linking among neurons. By transforming the function approximation problem into a pattern recognition problem under an unsupervised framework, GMR is able to approximate every kind of function or relation, and simultaneously its inverse, if it exists, or the reverse relation. It also outputs all the solutions (even infinite), including their corresponding mapping branches and, if the case, the equilevel surfaces.

The goal of this thesis is then the implementation of the GMR algorithm, using tree search technique to speed up its computation, and exploring the properties of GMR based on experiments.

The thesis is organized as follows:

After introducing the mapping approximation problem, the existing neural approaches to the function approximation problem are briefly described in Chapter 2. The rest of the chapters will focus on GMR. Chapter 3 describes EXIN SNN, an important building block in GMR learning. Chapter 4 describes the algorithm of GMR. The use of Branch and Bound search technique to accelerate computation in GMR is explained in Chapter 5. Chapter 6 discusses the parameter setting and properties of GMR based on the experimental results. The final chapter gives the conclusions of this thesis.

Chapter 2

Neural Approaches to Function Approximation

2.1 Function Approximation

In general, neural function approximation is achieved by identifying the input-output relationship, described by a representative set of input-output pairs.

Approximation of functions in neural networks is based upon traditional mathematical curve fitting techniques such as least squares. Presented with a set of discrete values of a function some technique is employed to produce a function which best fits the presented data. In other words the goal of the neural network is to define a function which maps each input data point as closely as possible to its desired output value. Let $g{:}R^D \Rightarrow R$ be such a mapping. Given D-dimensional input vectors of $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, ..., $\boldsymbol{v}_N$ and their 1-dimensional desired output values $Z_1^{des}, Z_2^{des}, \cdots, Z_N^{des}$, a mapping function, $g(\ )$, is found which approximates $Z_k^{des} = f(\boldsymbol{v}_k)$. The best-fit function is an interpolating function.

The approach to approximation in a 3-layer feed-forward neural network is to break the function into pieces based on similarity of the input data during the clustering process. Then an interpolating function, $g_r(\ )$ where $1 \le r \le M$, is found for each piece or cluster. In this manner the entire function is approximated by combining the interpolating functions from each cluster. Some approximation techniques use only the cluster in which the input parameters fall to evaluate the approximate function value. Other techniques use a combination of the interpolating function's values of each cluster. In the latter case the contribution of the corresponding cluster usually weighs more heavily in the combination.

The approximated or interpolated values for the function are obtained from the best-fit mapping functions. The general form of the approximating function is predetermined and the best-fit coefficients of the mapping function are obtained through a mathematical technique such as least squares. The general form of approximating function can be as simple as a plane in D dimensions,

$g_r = s_1 v_1 + s_2 v_2 + \ldots + s_D v_D + b$, $\boldsymbol{s} = [s_1, s_2, ..., s_D]^T$ is the slope of the plane and *b* is the intercept. The neural network provides the optimum slope and intercept. A more complex approximating function could be the Gaussian or normal curve which has the general form:

$$g_r = e^{-\frac{\|\boldsymbol{v}-\boldsymbol{u}\|^2}{2(\sigma)^2}} \qquad (2.1)$$

where $\boldsymbol{u}$ is the center of the bell shaped curve, $\boldsymbol{v}$ is the input data, and σ controls the width of the bell or how rapidly the function drops to a near zero value. The values for $\boldsymbol{u}$ and σ are determined by the neural network.

Neural networks can be implemented with one unit for each input data vector rather than one unit representing each cluster of input data. This technique is more computationally demanding even though a clustering process is eliminated and may result in poor "generalization" characteristics. Poor generalization occurs when the neural network correctly maps the training data to interpolating functions but is ill-suited to approximate other sets of data from the sampled function. Poor generalization is even more likely if the input data contain noise, random unpredictable errors. Usually the more units implemented, the better the approximation if using noise-free data. Constraints on processing time and processor storage resources frequently make it impractical to implement one unit for each input data vector.

2.2 Neural Networks for Inverse and Discontinuous Problems – Modelling Conditional Distributions

The basic goal in training a feed-forward neural network can be viewed as that of modelling the statistical properties of the generator of the data, expressed in terms of a conditional distribution function *p(t|x)* [4]. For the sum-of–squares error function, this corresponds to modeling the conditional distribution of the target data in terms of a Gaussian distribution with a global variance parameter and an *x*-dependent mean. However, if the data has a complex structure, then this particular choice of distribution can lead to a very poor representation of the data. Next, the existing general neural frameworks which apply mixture models for modelling conditional probability distribution will be described.

Mixture models represent a distribution in terms of a linear combination of adaptive kernel functions. Applying this technique to the problem of modelling conditional distributions, the formula turns:

$$p(t|x) = \sum_{j=1}^{M} \alpha_j(x) \phi_j(t|x) \qquad (2.1)$$

where M is the number of components, or kernels, in the mixture. The parameters $\alpha_j(x)$ are called mixing coefficients, and can be regarded as prior probabilities (conditioned on x) of the target vector t having been generated from the jth component of the mixture. Note that the mixing coefficients are taken to be functions of the input vector x. The function $\phi_j(t|x)$ represents the conditional density of the target vector t for the jth kernel. Various choices for the kernel function are possible. However, the kernel function is normally a Gaussian of the form

$$\phi_j(t|x) = \frac{1}{(2\pi)^{c/2}\sigma_j^c(x)} \exp\left\{-\frac{\|t-\mu_j(x)\|^2}{2\sigma_j^2(x)}\right\} \qquad (2.2)$$

where the vector $\mu_j(x)$ represents the center of the jth kernel with components μ_{jk}, and c is the dimensionality of t.

For any given value of x, the mixture model (2.1) provides a general formalism for modelling an arbitrary conditional density function $p(t|x)$. If we take the various parameters of the mixture model, namely the mixing coefficients $\alpha_j(x)$, the means $\mu_j(x)$ and the variance $\sigma_j^2(x)$ to be governed by the outputs of a conventional neural network which takes x as its input. This technique was introduced in the form of *mixture-of-expert* model (Jacobs *et al*., 1991), as will be described soon. By choosing a mixture model with a sufficient number of kernel functions, and a neural network with a sufficient number of hidden units, this model can approximate as closely as desired any conditional density function $p(t|x)$ (see Fig 2.1).

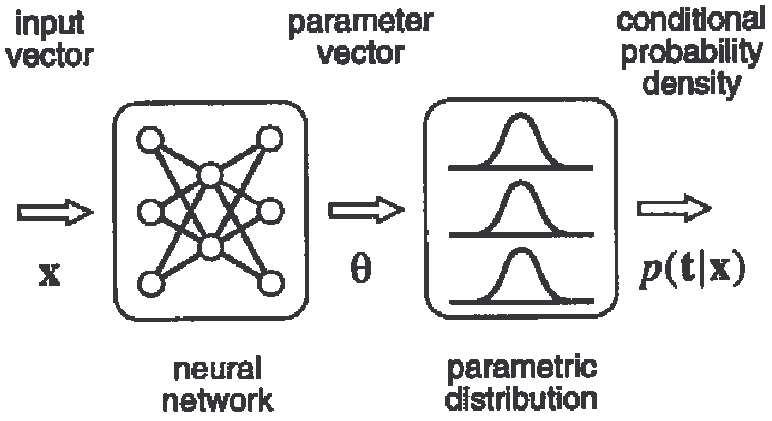


Fig. 2.1 General conditional probability densities $p(t|x)$ by considering a parametric model for the distribution of t whose parameters are determined by the outputs of a neural network which takes x as its input vector.

2.2.1 Multi-layer Perceptrons

The neural network in Fig 2.1 can be any standard feed-forward network structure with universal apporximation capabilites. Bishop [4] gave an example of using a multi-layer perceptron (MLP), with a single hidden layer of sigmoidal units and an output layer of linear units. For M components in the mixture model (2.1), the network will have M outputs denoted by z_j^{α} which determine the mixing coefficients, and $M \times c$

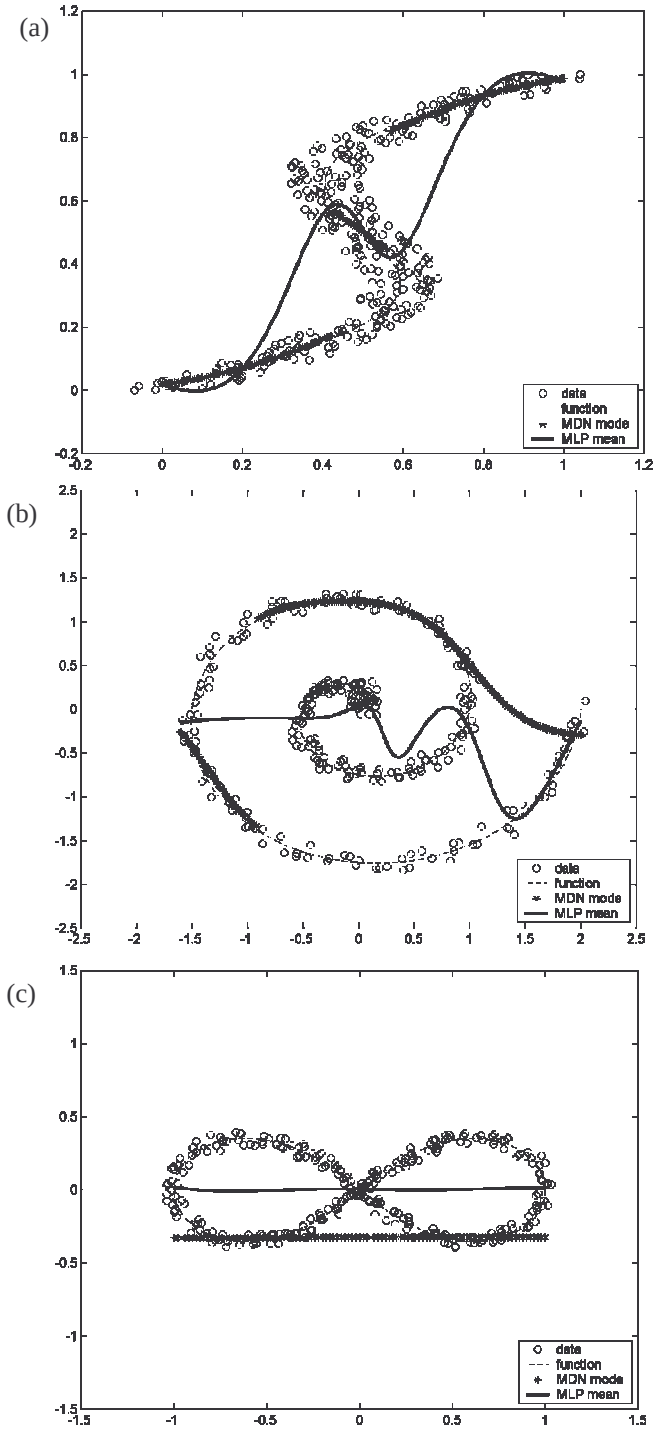


Fig.2.2 Examples of using MLP to model conditional distributions. The bold lines are plots of the central value of the most probable kernel as a function of x from a MLP network. This gives a discontinuous functional mapping from x to t. The solid curves show the conditional average of target data trained by sum-of-squares technique.

outputs denoted by z^{μ}_{jk} which determine the components μ_{jk} of the kernel centeres $\boldsymbol{\mu}_j$. The total number of network outputs is given by $(c+2)\times M$, as compared with the usual c outputs for a netwoork used with a sum-of-squares error function.

Fig 2.2 gives some examples of using the above MLP to model the conditional distributions in multi-value problems [3]. The circles denote the training set. The most probable branch which has the greatest associated 'probability mass' is represented in bold line compared with the solid curve which denotes the conditional average of the target data trained by the sum-of-squares technique. As shown in these examples, the 'most probable' branches cannot give good representation of the mapping with complex structures. It's even worse for some mappings such as shown in Fig 2.2 (b) and (c).

2.2.2 Mixture of Experts

The fundamental problem for mixture-of-experts (ME) (Jacobs *et al.*, 1991) [11] is to learn a mapping in which the structure of the mapping varies for different regions of the input space. The ME approach provides a mechanism for partitioning the solution to a problem between several networks. Its architecture is shown in Fig 2.3.

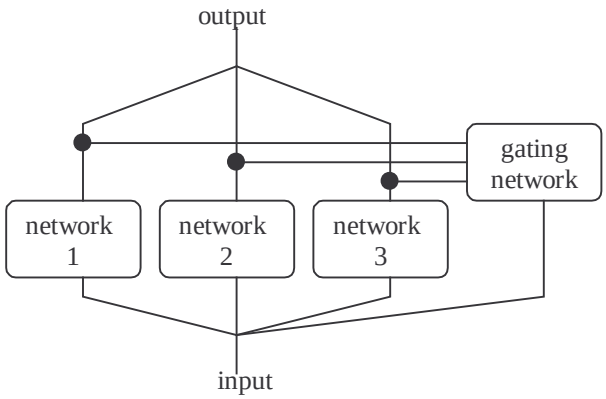


Fig. 2.3 Architecture of the mixture-of-experts modular network. The gating network acts as a switch and, for any given input vector, decides which of the expert networks will be used to determine the output.

It uses a separate expert network to determine the parameters of each kernel, and a further gating network is used to determine the coefficients. When the trained network is used to make predictions for new inputs, the input vector is presented to the gating network and the largest output is used to select one of the expert networks. The input vector is then presented to this expert network whose output $\mu_i(x)$ represents the prediction of the complete system for this input. This corresponds to the selection of the most probable branch of the conditional distribution on the assumption of weakly overlapping Gaussians.

The hierarchical mixture-of-experts (HME) (Jordan and Jacobs, 1994) [12] is the extension of the mixture-of-experts model by considering a hierarchical system in which each expert network can itself consist of a mixture-of-experts model with its own gating network. This can be repeated at any number of levels, leading to a tree structure. An application of ME and HME to speech recognition can be found in [13].

From now on, the GMR architecture will be focused on. Although GMR shares some ideas such as using divide-and-conquer strategy in learning with the above cited networks, it uses a linking procedure to keep the topological information of the data set; in the recalling phase it can output all the solutions and its corresponding mapping branches. It's proposed to solve more generalised mapping approximation problem.

Chapter 3

The EXIN Segmentation Neural Network

The EXIN Segmentation Neural Network (EXIN SNN) (G. Cirrincione, 1998) [1] is one of the important building blocks for GMR. It is a particular self-organizing neural network, basically a simplification of the Progressive Learning Network (PLN) [1] which is derived from the Counter-propagation network (CPN) [6].

3.1 The Progressive Learning Network (PLN)

The Counterpropagation neural network (CPN), introduced by Hecht-Nielsen (1987, 1988) [16], is a clustering network, that in its feedforward version, works as a neural statistically equiprobable look-up table. In the first training phase, the CPN splits up the input space into clusters by an iterative Kohonen learning process. Subsequently, to each cluster in the input space, represented by the weight vector w_k, the corresponding cluster in the output space, represented by the weight vector w_k, is associated. In the recall phase, when a generic input vector x is presented, the closest cluster centroid w_k in the input space is selected by using the Euclidean distance metric; the estimated output y is then chosen as the corresponding cluster centroid v_k in the output space. The accuracy of the achieved function mapping thus depends on the number of PE's in the hidden layer. Moreover, the Kohonen learning moves the hidden layer PE's weights in order to reproduce the probability distribution of the training set (TS). The PE's are then thickened in those regions of the input space where also many samples of the TS exist and loosened elsewhere.

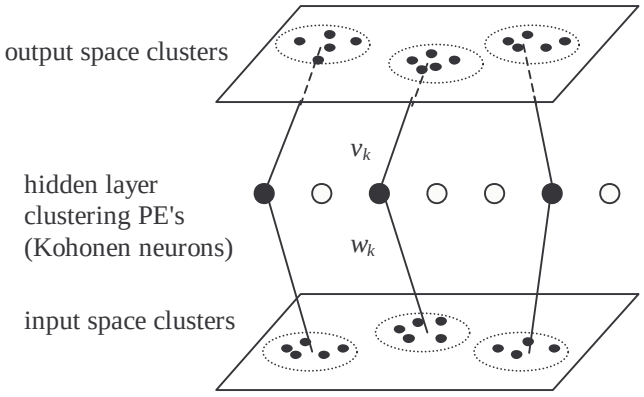


Fig.3.1 The Counterpropagation network (CPN) clustering

Although the CPN allows reduced training time compared to the MLP , problems can arise when the number of the hidden layer PE's has not been adequately chosen.

Further more, the CPN does not allow incremental learning, due to the iterative nature of the Kohonen learning process.

The Progressive Learning Network (PLN) has been developed to overcome the difficulties of CPN in on-line applications. PLN is made up of three layers of processing elements (PE's). The input and output layers, consisting of PE's with linear transfer function, have the task of normalising and are fully connected to the hidden layer, which is composed of competitive PE's with linear transfer fucntion and has no predefined dimensions. The number of its PE's is increased or decreased automatically by the ANN according to the required mapping accuracy. Its training requires only one epoch. The algorithm of PLN has two operating modes, learning and recalling, that can be started independently. Moreover, there is a merging algorithm, periodically activated during the learning and aimed at regulating the distribution of the PE's in the hidden layer. Next, the PLN variant , the EXIN SNN will be introduced.

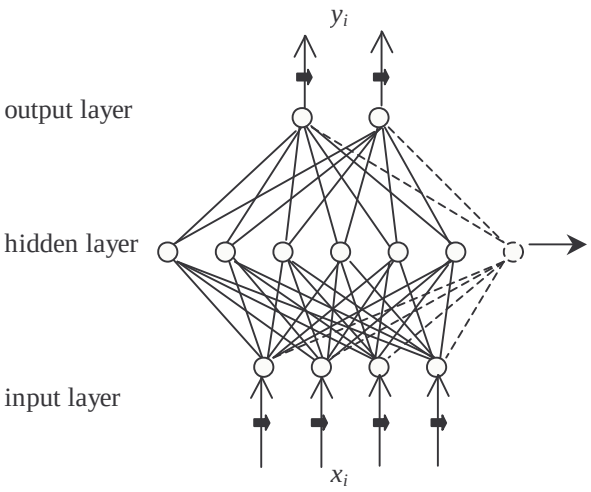


Fig. 3.2 The Progressive Learning Network (PLN)

3.2 The EXIN Segmentation Neural Network (EXIN SNN)

The EXIN SNN works as an unsupervised network, hence doesn't need the weights from the hidden layer to the output layer. The network parameter of the modified unsupervised PLN are: K, dimension of the hidden layer; w_k, weight vector connecting the input layer to the kth PE of the hidden layer; s_k state variable for the neuron k, equal to the number of samples indentified by the kth PE of the hidden layer; ρ, input space vigilance threshold (positive scalar). The number K of PE's of the hidden layer is initially set to zero. The index i stands for the ith presented sample. The EXIN SNN learning is given by the following procedure.

3.2.1 The Learning Algorithm

1. Present the first vector of the traning set (TS) x_1, let $K = 1$ and assign the vector representing the input as weight of the first PE: $w_1 = x_1$. Let $i = 2$.
2. Present the ith sample x_i and compute the K Euclidean distances δ_k between the vector x_i and the weights w_k of the K PE's of the hidden layer.

3. Sort out the δ_k's in increasing order and select the closest neuron as winner, whose weight is w_w.
4. If $\delta_w > \rho_x$, go to 6.
5. Update the weights and the state of the wth PE according to :

$$w_w = \frac{s_w w_w + x_i}{s_w + 1} \qquad (3.1)$$
$$s_w = s_w + 1$$

and go to 7.

6. Create a new PE setting:

$$K = K + 1, w_k = x_i, s_k = 1$$

7. Let $i = i + 1$ and go to 2.

After learning (one epoch), the modified PLN has clustered the input data according to the given attribute vectors. Fig 3.3 gives an example of the EXIN SNN learning.

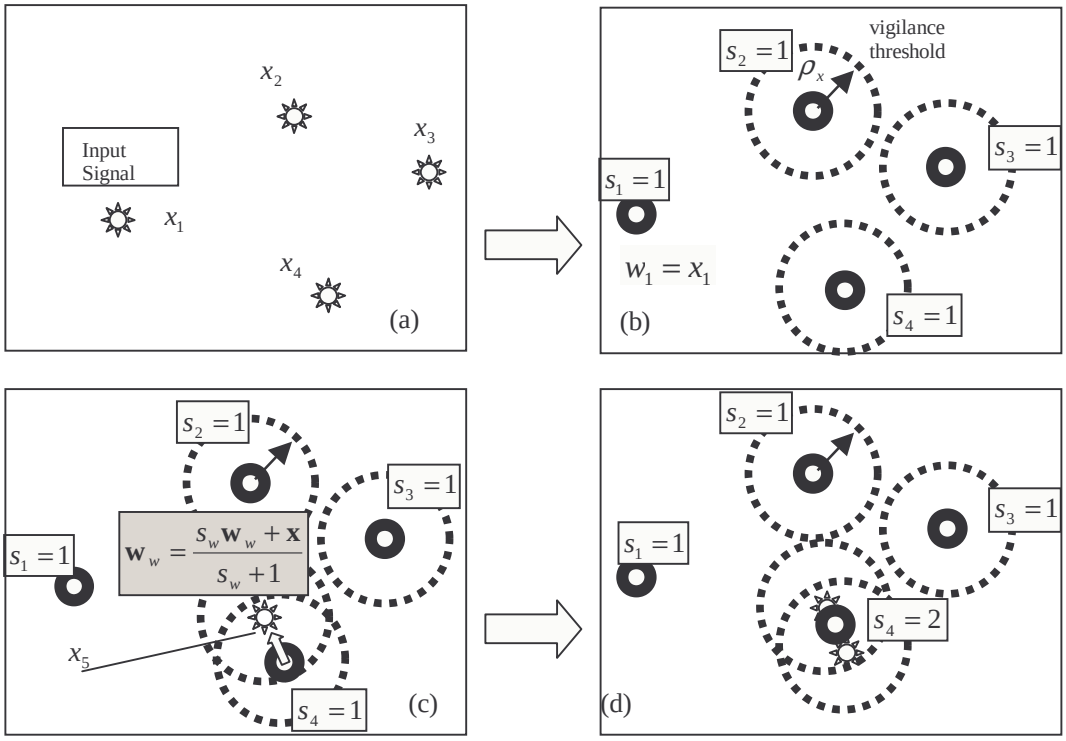


Fig. 3.3 Example of EXIN SNN learning
Figs. (a) and (b) show the first 4 input signals (stars) which have caused the creation of 4 neurons (solid circles), as no neuron is within the vigilance threshold of the input signal. The neuron weight is identical to the input signal at first.
Figs. (c) and (d) show that neuron 4 is within the vigilance threshold of the 5th input signal x_5, thus w_5 is updated according to eq. (3.1) $w_w = \frac{s_w w_w + x_i}{s_w + 1}$.

3.2.2 Cleaning algorithm

The aim of the cleaning is to merge some PE's that become too close, i.e. below the vigilance threshold: $\lambda\rho_x$ where λ is a predefined constant not greater than 1, during the learning process. The general form of cleaning is the following:

1. Let *i* and *j* two PE's of the network.
2. Compute the Euclidean distance D_{ij} between the weights w_i and w_j of the *i*th and *j*th PE's.
3. if $D_{ij} > \lambda\rho_x$, go to 6.
4. If $s_i < s_j$, then exchange $w_i \leftrightarrow w_j$, $s_i \leftrightarrow s_j$.
5. Update the ith PE according to:

$$w_i = w_i + s_j \frac{w_j - w_i}{s_i + s_j} \qquad (3.2)$$

6. Remove the neuron *j*.
7. Choose a new *i* and *j* pair of PE's and go to 1.

Chapter 4

The Generalised Mapping Regressor Algorithm

GMR (G. Cirrincione, M. Cirrincione 1998) [1], [2] can be defined as a global mapper with the following desirable characteristics:

- It solves the inverse discontinuous problems.
- It approximates every kind of mapping (function or relation) in both senses, i.e. $M(x,y) : x \in \Re^m \leftrightarrow y \in \Re^n$.
- It is able to find all the solutions to a given input, and the corresponding branch of the mapping. Its input could be every possible collection of components of x and y, and its output is the estimation of the remaining components (Fig.4.1).
- It yields the equilevel hypersurfaces.
- It is an open architecture, under which many different neural techniques can be chosen so as to improve its performance.

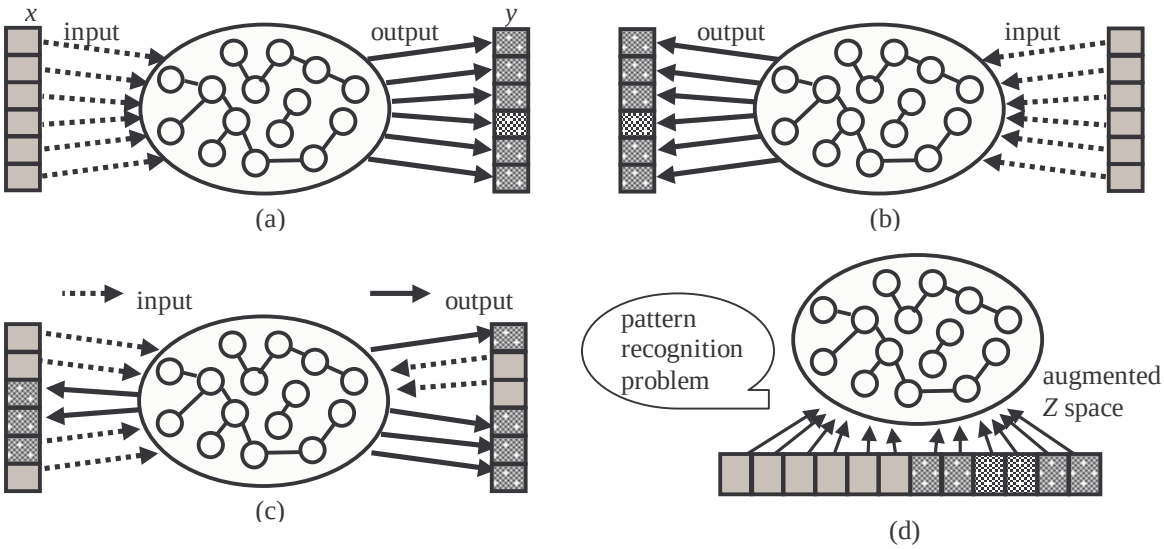


Fig. 4.1 Illustration of a global mapper.
(a)(b)(c) GMR can approximate every kind of mapping. Its input could be every possible collection of components of x and y, and its output is the estimation of the remaining components.
(d) GMR transforms the function approximation problem into a pattern recognition problem via the augmented Z space.

These features justify the name: Generalised Mapping Regressor (GMR). Generally, GMR is an incremental self-organizing competitive neural network with adaptive linking among neurons; both the neuron weights and the linking among neurons are used for computation of the output of the network.

4.1 The Basic Ideas

The basic idea of GMR is to transform the function approximation problem to a pattern recognition problem under an unsupervised framework. Hence, a coarse-to-fine strategy of covering of the mapping is used. Different techniques can be used for this purpose in order to improve the interpolation and extrapolation properties of the network. However, GMR suffers from the curse of dimensionality, i.e. high dimensionality of the input and output spaces require too many neurons for a dense covering. However, this can be alleviated by preprocessing data with some other techniques to convert the high-dimensional space into a lower dimensional space.

Another important principle is the generation of the linking matrix, which carries topological information between neurons. The linking criterion implies a preference of connections directionally biased by the winner neuron and the input data at each iteration. The branch information for each solution and the possible equilevel surfaces are given by *linking tracking*.

The originality and power of GMR lies mainly in its open architecture. Within this framework, many different neural algorithms can be used in adaptation to different problems.

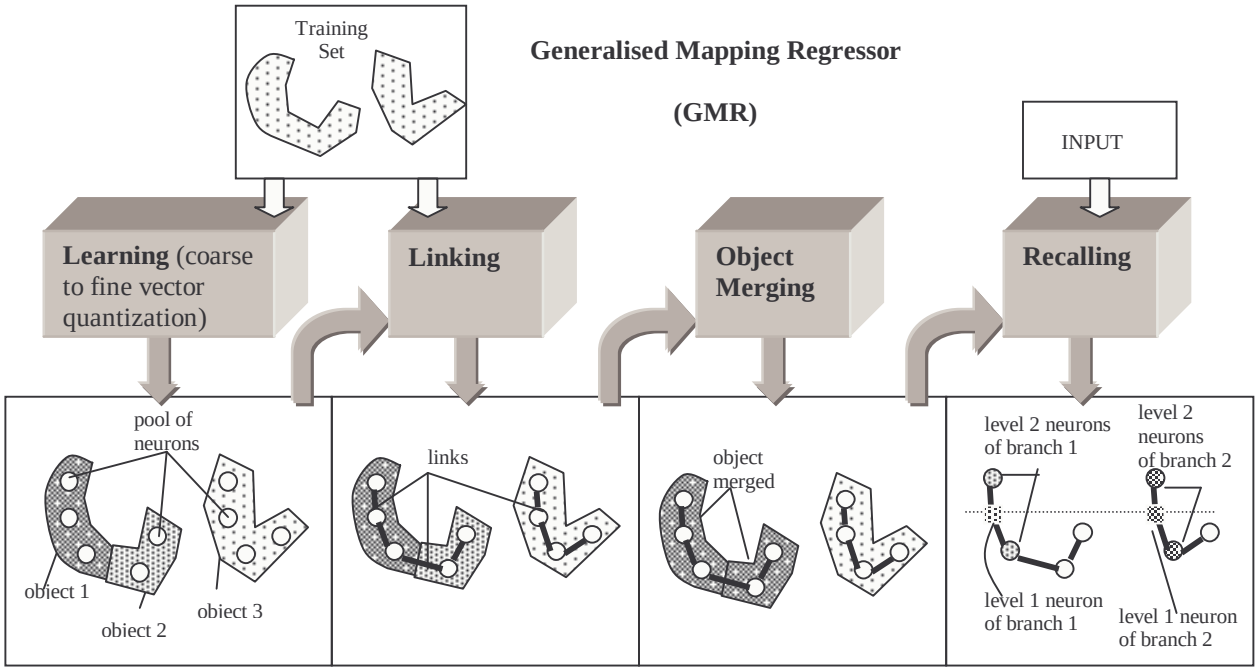


Fig. 4.2 Schematic illustration of the 4 phases in GMR.
The 4 diagrams on the bottom show an example of the 4 phases.

4.2 Algorithm

As shown in Fig 4.2, the GMR approach can be identified by four phases: learning, linking, merging, and recalling. The algorithms of each phase and its processing results (schematically illustrated in the bottom 4 diagrams in Fig 4.2) will be explained in the following four sections.

4.2.1 Learning (coarse-to-fine vector quantization)

The GMR approach requires the unsupervised learning of the given mapping. In order to do so, for the *i*th example (x_i, y_i), build the augmented vector $z_i = \left[x_i^T, y_i^T\right]^T \in \Re^{m+n}$. The overall TS is represented by the matrix $Z \in \Re^{N\times(m+n)}$ where N is the number of input data and $Z = \left[z_1^T, ..., z_N^T\right]^T$. GMR performs a vector quantization of the augmented subspace; the resulting clusters are representations of the branches of the mapping. The quantization can be obtained by using different neural approaches, which must be incremental, i.e. the number of neurons is not predefined but changes according to the complexity of the mapping to be approximated. In this thesis, EXIN SNN is implemented due to it's speed. Other more sophisticated techniques can be suggested in order to have an accurate quantization together with good smoothing (interpolative) properties (note that EXIN SNN yields a pointwise covering and so needs particulare linear or nonlinear techniques for interpolation in the recall phase): as suggested in [2], either RAN or M-RAN, which introduce a smoothing because of their Gaussian kernels. Other alternative clustering neural networks include Growing Neural Gas [13], Growing Neural Grid [14], Self Organization Map [15].

The learning phase can be divided in two subphases, coarse quantization and fine quantization (Fig 4.3). Coarse quantization is obtained by using a network, called *primary*. This network is EXIN SNN where a high value for vigilance threshold ρ_1 is chosen. More than one epoch is allowed for training. The resulting neurons are called *object* neurons.

In the second subphase, each input data is presented to the network for labelling according to its winning object neuron (the closest in Euclidean distance to the input). In the end, every object neuron is associated to the set of inputs for which it has won (Voronoi set). This set represents the *domain* of the object neuron. Apparently, the TS has been divided into mutually exclusive (no overlap) subsets (or domains) as many as the object neurons, and their sum is the original training set. Every subset is considered as TS for a *secondary* EXIN SNN. Hence the same number of EXIN SNN's as the object neurons is used in parallel with the aim of quantizing each object domain. The vigilance threshold for this intradomain learning, say ρ_2, should be lower than ρ_1. The value of this final vigilance threshold is determined by the required resolution. Finally, the neural network consists of a pool of neurons generated by the

secondary learning. Each final neuron is labelled as belonging to its corresponding object neuron (the pool of neurons doesn't include the object neurons). These labelled neurons represent a discrete Voronoi tesselation of the input space.

This coarse-to-fine vector quantization can be recursively repeated using other levels, although in practice two or three levels are enough.

As the EXIN SNN is incremental, pruning techniques sometimes are needed. For example the neuron cleaning which has been introduced in Chapter 3 can help to prune the redundant information.

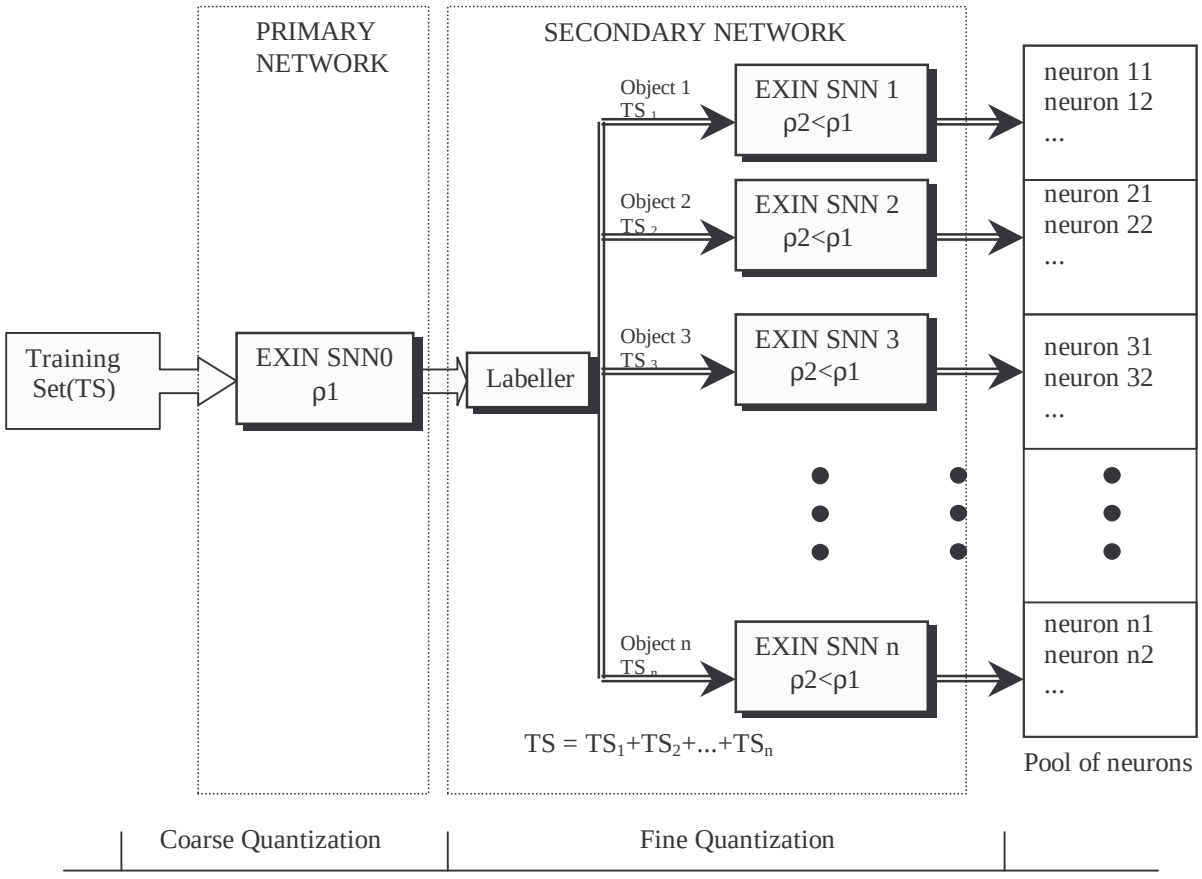


Fig. 4.3 Coarse to Fine Learning Architecture in GMR approach. The vigilance threshold in coarse quantization ρ1 is greater than the one in fine quantization ρ2.

4.2.2 Linking

After the quantization phase, the GMR network is composed of a pool of p neurons. The weight vector for neuron j (j = 1,..., p) is $w_j \in \Re^{m+n}$. Two tasks should

be fulfilled in this linking phase. One is to generate a linking matrix $V \in N^{p \times p}$, whose element $V_{\alpha\beta}$ represents the strength (integer) of the connection between neuron α and neuron β. This matrix has zero diagonal and is symmetric. It's initialized as a null matrix. The other task is to setup the domain vector $r = [r_1, \cdots, r_p]^T$, where r_i, called domain variable, is a state variable which is associated to each neuron *i*, and as it will be clear soon, represents the "radius" of the domain of the neuron in the weight space. The components of this domain vector are set to a predefined constant $\rho_{\min} = k_{\min}\rho_f$ ($k_{min} < 1$), where ρ_f is the final vigilance threshold of the quantization and k_{min} is the domain factor.

The basic requirement of the linking is that connection between neurons store the direction of the vector connecting the input vector to the weight vector of the winner (the closest neuron to the input). Hence, this linking carries a directional information driven by the input data and can be used to determine both the position and the shape of the branch in the weight space. Then the linking matrix is generated by a complete presentation of the TS to the network. For each input z_i (one iteration), the following operations are performed, mainly to determine which neuron has to be linked to the winning neuron.

1. Present one input z_i to the network.
2. Compute all the Euclidean distances between the neuron weights and the input. Sort the pool of neurons by the distance increasingly into a list of vectors: w_1, w_2, ..., w_p. The subscript number means the position of the neuron weight in the list. w_1 represents the weight vector of the winner neuron (the closest neuron). The position vector from w_1 to z_i is defined by $d_1 = z_i - w_1$.

   (a) If $d_1 > 0$ (e.g. see Figs 4.4 (a) and (b)), define the position vectors of the input and of the nonwinning neuron as $d_j = w_j - w_1$, for j = 1,...,k, and k is given by the greatest index such that:

   $$\|z_i - w_k\|_2 \le \delta \|d_1\|_2 \qquad (4.1)$$

   with δ a predefined constant greater than 1, called linking factor. The k nonwinning neurons are situated inside a hypersphere centered on the input and of radius larger than the distance from the input to the winning neuron. Test (4.1) prunes neurons that are too far from the winning neuron and so will not be retained in the candidate list for linking to the winner neuron. Go to step 3 and do the directional test to the k remaining linking candidates.

   (b) If $d_1 = 0$ (e.g. see Fig 4.4 (c)), i.e. the weight of winner neuron w_1 is identical to the input z_i, then the second winner w_2 will be connected. $V_{n1,n2}$ and $V_{n2,n1}$ are increased by 1, where n_1, n_2 are the indices of first winner and second winner, respectively. This is often the case in the region where the density of input data is low (sparse region). Go to step 4.

3. Compute, $\forall j = 1,...,k$,

$$p_j = \frac{d_1^T d_j}{\|d_1\|_2 \|d_j\|_2} \qquad (4.2)$$

and the $h = \begin{cases} \arg\max(p_j), \text{ if } \exists\, p_j > 0 \text{ for at least one value of } j \\ \arg\max(|p_j|), \text{ otherwise} \end{cases}$

The neuron whose weight is w_h is connected to the winning neuron. Equation 4.2 means that the selected unit is the neuron j whose position vector (to the winning neuron) d_j is the nearest to the straight line parallel to d_1 (linking direction), among all neurons whose position vectors are located in the half space bordered by the hyperplane containing the winning neuron weight vector and normal to the linking direction, as illustrated by the example in Fig 4.4(a). If all neurons have position vectors outside this half space as the case shown in Fig 4.4(b), the selected neuron must have the nearest position vector to the linking direction among all neurons. In brief, the criterion takes in consideration the angular distance from the linking

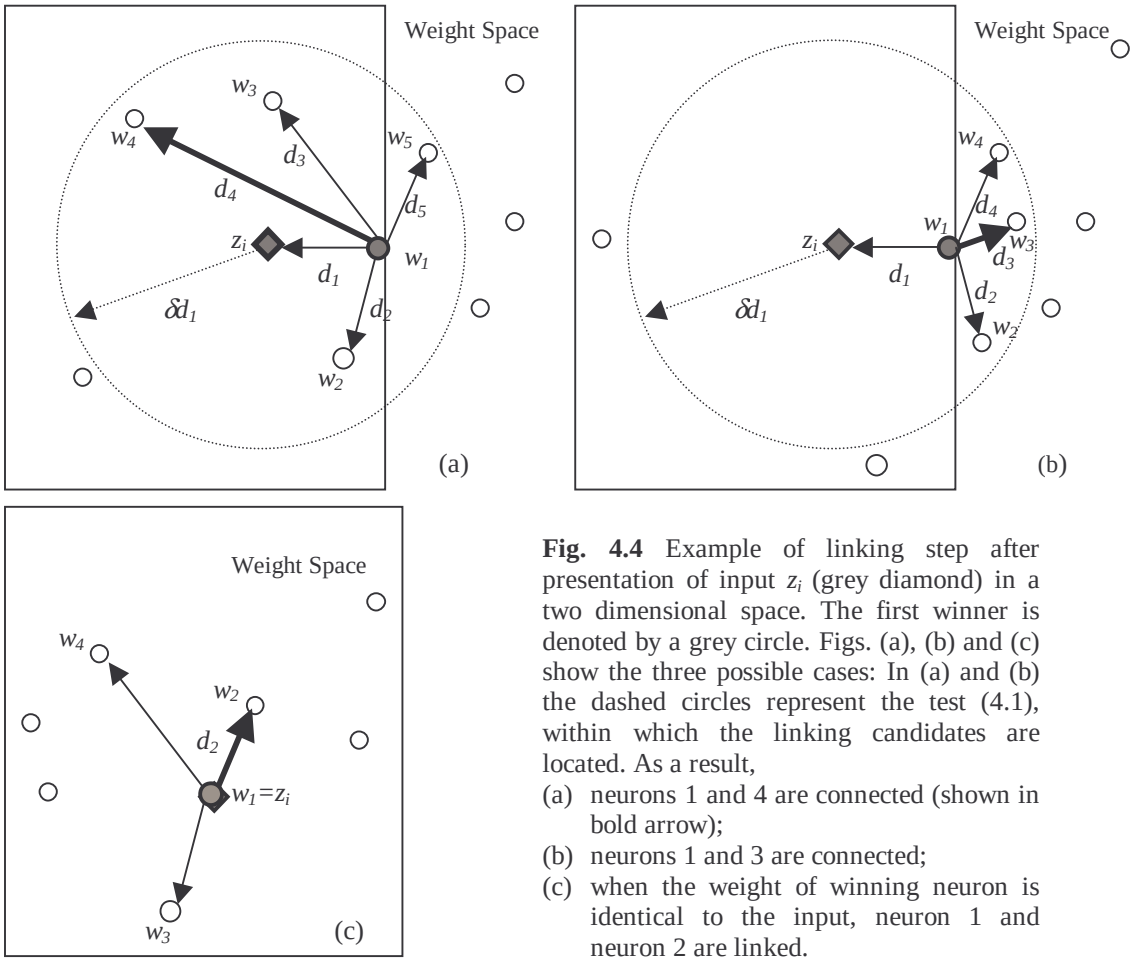


Fig. 4.4 Example of linking step after presentation of input z_i (grey diamond) in a two dimensional space. The first winner is denoted by a grey circle. Figs. (a), (b) and (c) show the three possible cases: In (a) and (b) the dashed circles represent the test (4.1), within which the linking candidates are located. As a result,

(a) neurons 1 and 4 are connected (shown in bold arrow);
(b) neurons 1 and 3 are connected;
(c) when the weight of winning neuron is identical to the input, neuron 1 and neuron 2 are linked.

direction. This choice makes the linkings carry the geometrical information which will be used in order to track branches and equilevel surfaces. Only one connection is made at each input presentation: $w_1 \leftrightarrow w_h$. If the corresponding indices of w_1 and w_h are n_1 and n_h, then the corresponding element in linking matrix $V_{n1,nh}$ and $V_{nh,,n1}$ are incremented by one.

4. If the domain state variable associated to the winning neuron r_{n1} is less than the distance from the input to the winning neuron, i.e. $\|d_1\|_2$, replace it with $\|d_1\|_2$.

After the linking steps, three informations are available for each neuron: its corresponding object neuron, the distance in the weight space from the farthest input vector in its domain and its connected neurons.

4.2.3 Object Merging

Check the whole linking matrix. If there are links between neurons belonging to different objects, further check the strength (value of the corresponding element of V) of the link. If it is above a certain threshold χ, the *merging threshold*, the two objects are merged in a unique object and the associated neurons are relabelled.

4.2.4 Recalling

After the previous training steps, GMR is able to output the required solutions. In this recalling phase, the dimension of input vector is not necessarily equal to m, instead its components can belong either to the input space or to the output space (see Fig.4.1). For sake of simplicity, call the general input x as before and consider it as m-dimensional. Define the subspace of the input as X. The following steps is to assign to the neuron i, for $i = 1,..., p$, two state variables: l_i which is an integer representing the *level* of the neuron and b_i representing the *branch* of the mapping containing the neuron. They are initialized at each input presentation to a zero vector. In the beginning, all neurons are considered as level zero neurons.

1. Feed x to GMR.
2. Compute the norm of the projections on the space $x \in \Re^m$ of the vectors from x to all final neuron weights. It is easily done by using only the elements of the weight vector whose position indices correspond to the position indices of the input elements in the augmented vector z. For instance, if the first and last elements of the vector z are taken as input, all operations use weight vectors of reduced dimension composed of only the first and last elements of the whole weight vectors. Classify neurons according to the *restricted distances* in increasing order (o_i is the position of the neuron i in the classification).
3. Define the winning (closest) neuron as w, i.e. $o_w = 1$ at the first iteration.

4. Consider the domain variable of the winner r_w. Check:

$$\left\| x - w_w \right\|_2 \le r_w \qquad \text{(level one test)} \quad (4.3)$$

that is if the input is within the domain of the winning neuron. If not, then go to step 7. If the check is satisfied, then set l_w to one (level one) and b_w to w (the mapping branch is represented by its level one neuron).

5. For each neuron j linked (by matrix V) to neuron w, check the level. If neuron j is level zero, i.e. $l_j = 0$, then $l_j = 2$ and $b_j = w$ (neuron j belongs to the branch of the winner). Resuming, the connected level zero neuron j becomes a level two neuron belonging to the winner branch. In case of *branch crossing*, i.e. one of the linking neurons, say neuron j, is level 1, $l_j = 1$, then all the neurons whose branch variables set to w should be changed to j, the number of crossed branch which is closer to the input. A variant of this step can take account of the strength of the link in the formulation of the branch test.

6. If all the neurons have been checked, go to step 7; else $o_w = o_w + 1$. Consider the new winner w, go to step 4 to do level one test.

7. Check if different branches contain neurons belonging to the same object. If so, and if the strengths between the two sets of neurons are relevant, then they are merged together.

Once the recalling (or production) steps are complete, for a given input x, every neuron has a level and a branch state and the solutions and equilevel surfaces can be output:

- The level one neurons are the ones with the input x in their domain. The weight vector of a level one neuron yields an output y as the part of the weight orthogonal to the X subspace, i.e. the remaining components of the weight vector (complement to the restricted weight vector). Two different solutions belongs to the same branch if they have the same branch label.
- All level one neurons and the branches or portion of branches containing only level one neurons constitute the equilevel surfaces. Disjoint equilevel surfaces (branches) for the same object are also possible; e.g. the equilevel curves of a saddle in the 3-D space for certain section orientations.
- The level two neurons are the ones connected to the level one neurons.
- The level zero neurons are isolated neurons which might be originated by the noise and should be clipped out.

Some interpolation for the output is also needed. For reason of space, no interpolation is done in this thesis; then the accuracy of the result only depends on the covering of the mapping. An interpolation can be achieved by using both the level one

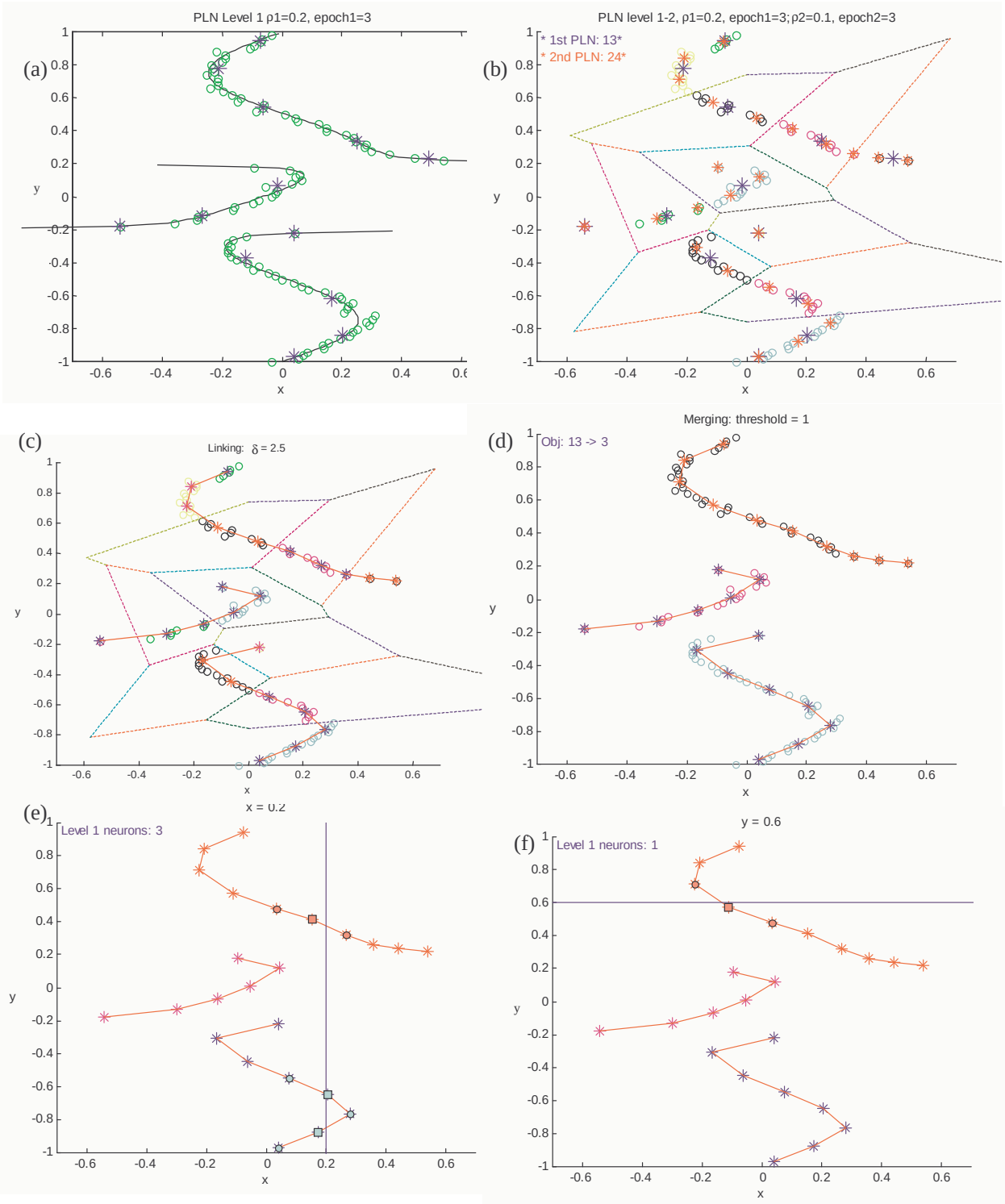


Fig. 4.5. Example of using GMR for mapping approximation.

(a) Coarse quantization: 13 object neurons (blue stars) are created. The green circles represent the TS. The solid black curves represent the inverse of the original function.

(b) Fine quantization: the TS is divided into 13 subsets (divided by the dashed lines, and each subset has different colors); 13 EXIN SNN's are used to do further quantization; thereafter the pool of 24 neurons (red stars) is generated.

(c) Linking: the final neurons are linked after the whole presentation of TS (the linkings are represented by red lines).

(d) Merging: the linked objects are merged; finally 3 separated branches are identified, and their associated subsets of TS and neurons are represented by different colors.

(e), (f) Recall: for input $x = 0.2$, GMR find 3 solutions (the value of y of level 1 neurons represented in squares) and 5 level 2 neurons (small circles). They belong to 2 branches (different colored). For input $y = 0.6$, one solution (the value of x of the red square neuron) and two level 2 neurons are found.

and level two neurons, but differently weighted. Another possibility is to replace neurons with radial basis functions and drive the interpolation by this smoothing.

A simple example is illustrated in Fig 4.5 and deals with the mapping of the inverse of the function $y = f(x) = \frac{1}{4}(\sin(2\pi x) + \frac{0.01}{x^2 - 0.04}) + \varepsilon$, where ε is a random variable with a uniform distribution centered around 0 and with a standard deviation σ = 0.15. The data set of this inverse problem is obtained by interchanging the roles of the input and output variable. The learning uses 2 level EXIN SNN's, the results of the coarse and fine quantization are shown in Figs. (a) and (b), respectively. Colors are used to discriminate different subsets of the TS. Figs. (c) and (d) reveal the reults after linking and merging. Figs. (e) and (f) show the solutions and their corresponding branches (in different colors) found after recalling.

4.3 Summary

Table 4.1 The four phases of GMR network

Phases	Purpose	Input	Output	Neural Techniques
Learn -ing	perform a coarse to fine vector quantization	whole TS in augmented *Z* space	object neurons and pool of final neurons whose labels indicate the corresponding object	unsupervised incremental network, e.g. EXIN SNN, Growing Neural Gas, Growing grid, SOM, RAN, M-RAN...
Link- ing	connect neurons which are neighbors both in distance and direction	whole TS, pool of neurons after learning phase	linking matrix which indicates the connection between neurons and domain vector which indicates the radius of the domain for each neuron in weight space	neighborhood function both in distance and direction
Merg- ing	identify the mapping branches	objects and pool of neurons in learning, linking matrix	new objects after merging, the corresponding new label for each final neuron	different objects which are linked via the connection between the neurons are to be merged if the linking strength is above the merging threshold
Recall -ing	output any possible solution, or equilevel surfaces for an input	an input vector whose components are extracted from the *z* weight	branch state and level state for neuron, solutions which are the estimations of the remaining components of level 1 neurons, level 2 neurons which are the connected neurons and level 0 neurons which are isolated	restricted distance computation and level one test, linking tracking, interpolation for output

GMR is a global mapper which can approximate every kind of mapping, by transforming the function approximation problem into a pattern recognition problem under an unsupervised framework. It's an open architecture under which different neural techniques can be used to improve its performance. Table 4.1 summarises the four main procedures in GMR network: learning, linking, merging and recalling.

Chapter 5

Using Branch and Bound to Accelerate GMR Computation

So far the entire algorithm of GMR approach has been introduced. In this chapter, the performance of this neural network in computation time will be discussed. Recall that in the above-cited algorithm, a large number of expensive distance computations are required in learning (each EXIN SNN, relabelling), linking matrix generation and recalling phase. This will be amplified greatly if either the size or the dimensionality of the training set is large or there are too many neurons (high complexity of the network).

In order to speed up the computation, the method of branch and bound (BnB) [17], is implemented in this thesis to facilitate rapid calculation of the *k*-nearest neighbors, by eliminating the necessity of calculating all distances. Before illustrating the use of BnB and its efficiency in GMR network, the general procedure of using BnB algorithm to compute *k*-nearest-neighbors (Narendra and Fukunaga, 1975) [18] will be explained in advance.

5.1 Basic Branch and Bound Search

Branch and bound is a well-known technique for solving combinatorial search problems. The basic scheme is to reduce the problem search space by dynamically pruning unsearched areas which cannot yield better results than solutions already found. In case of computing *k*-nearest-neighbors, the basic approach is first to hierarchically decompose the search space (data set) into several disjoint subspaces (subsets), and then the resulting grouping is applied to the branch and bound method which is a powerful tree search algorithm.

Let $\{X_1,...,X_N\}$ be an *n*-dimensional set of *N* data. It is required to compute the *k*-nearest neighbors of a test sample *X*, as measured by an appropriate distance function $d(.,.)$. For sake of simplicity, let us consider the case for a predefined $k = 1$. The extension of the algorithm for $k > 1$ is straightforward and will be shown later. The two stages involved in this method are :

- Decomposition of the Data Set

In the first stage, the data set is hierachically decomposed into disjoint subsets. The results of this decomposition are represented by a tree structure.

In order to do so, the data set is divided into l subsets, each subset is further divided into l subsets, and so on. Each node p is represented by the pair (m, n) (see Fig. 5.1), which means this is the mth node of level n of the tree. The tree represents a grouping of data and is characterized by the following parameters :

S_p set of data associated with node p,
N_p number of data associated with node p,
M_p sample mean of S_p,
$r_p = \max_{x_i \in S_p} d(x_i, M_p)$ (farthest distance from M_p to an $X_i \in S_p$).

Note that this tree is not necessary to be a symmetric one in which each node has

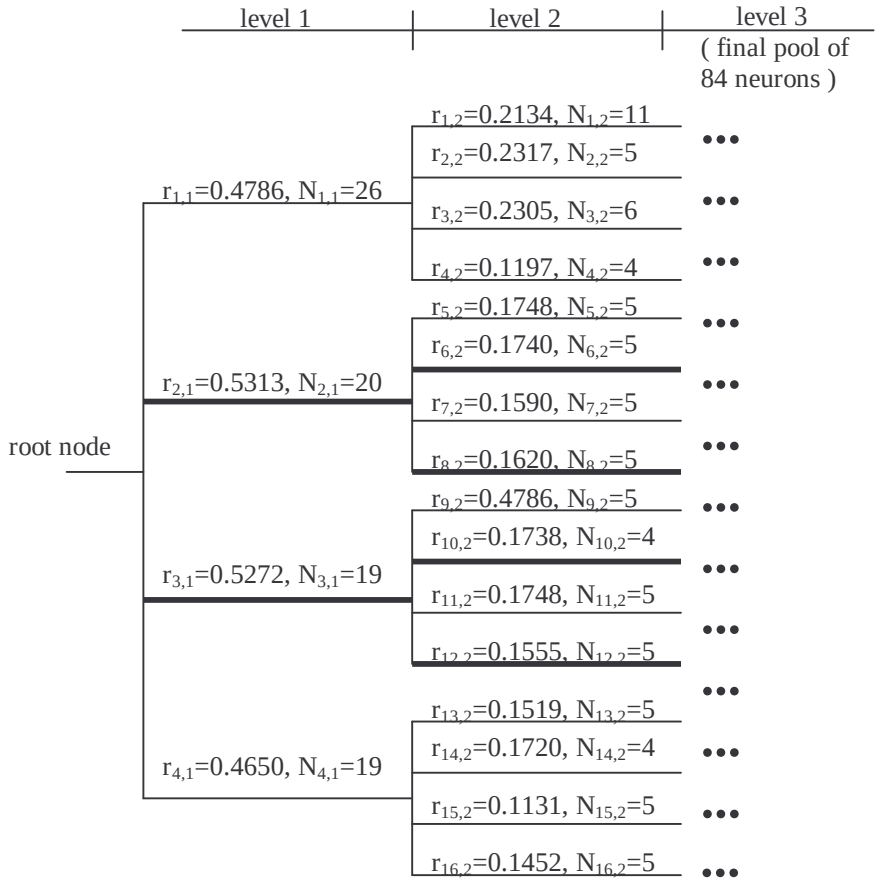


Fig. 5.1 Neuron tree structure built during learning phase of GMR. It's a 3-level tree structure obtained from the 3-level learning in a 2-D experiment (Table 5.1 (b)). Bold lines indicate the intermediate nodes expanded by the algorithm for a typical input data. By using BnB in this tree structure only 31 distance computations (instead of 84) are required to find out 4 nearest neighbors to the input data.

equal number of branches. Any clustering technique can be used for decomposing the data set. For example, using *k*-means-clustering while keeping *k* constant or decaying with levels, can create a symmetric tree; one alternative method is using EXIN SNN : here, instead of having a predefined branch factor, the number of branches of each node depends on the distribution of data set. Obviously, the resulting groups are not required to be 'meaningful' clusters. Computational economy is the chief consideration that affects the choice of the clustering procedure.

- Tree Search by Branch and Bound

After the data set has been decomposed, and the quantities M_p, r_p, N_p; and S_p evaluated, each node p can be tested as to decide whether or not the nearest neighbor to X may be in S_p, by the application of the following rules.

Rule 1: No $X_i \in S_p$ can be the nearest neighbor to X, if

$$B + r_p < d(X, M_p)$$

B is the distance to the current nearest neighbor of X among the design samples considered up to present. Initially, B is set to ∞.

Rule 2: X_i cannot be the nearest neighbor to X, if

$$B + d(X_i, M_p) < d(X, M_p),$$

where $X_i \in S_p$.

We can now apply the branch and bound method, in order to search the tree in Fig 5.1, testing the nodes by Rules 1 and 2. The tree-search algorithm is as follows.

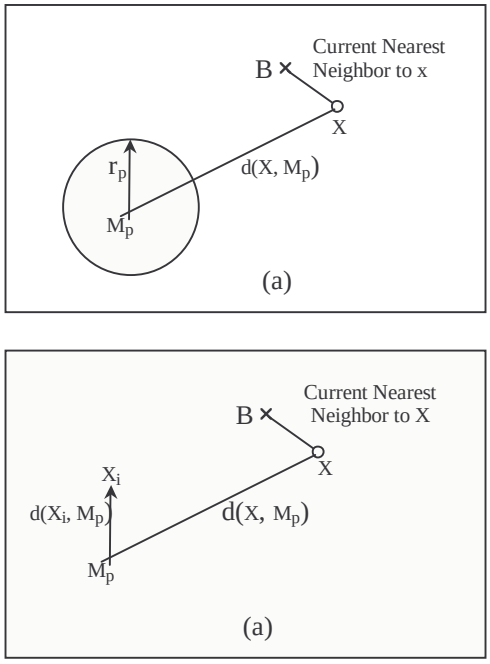


Fig. 5.2 (a) Rule 1 ; (b) Rule 2

1. (Initialization): Set $B = \infty$, current level L= 1, and Current_Node as root node.
2. (Expansion of Current_Node): Place all nodes that are immediately direct successors of the Current_Node into the Active_List at Current_Level. Compute and store $d(X,M_p)$ for these nodes. Update $B = \min\left[B, d(X, M_p) + r_p\right]$.
3. (Test for Rule 1): For each node p in the Active_List at the Current_Level, if $B + r_p < d(X, M_p)$, remove p from the Active_List at Current_Level.
4. (Backtracking): If there are no nodes left in the Active_List at Current_Level, backtrack to the previous level, i.e., set $L = L - 1$. If $L = 0$, then terminate the algorithm. If $L \neq 0$, go to Step 3. If there are one or more nodes in Active_List at Current_Level, go to Step 5.

5. (Choose the Nearest Node for Expansion): Choose the nearest node *p* (yielding the smallest $d(X, M_p)$), among the nodes in Active_List at Current_Level, and call it Current_Node. Remove *p* from the Active_List at Current_Level. Set Current_Level $L = L + 1$; if Current_Level is the final level, go to Step 6. Otherwise, go to Step 2.
6. (Test for Rule 2): For each X_i in Current_Node *p*, do the following : if $d(X, M_p) > d(X_i, M_p) + B$, X_i cannot be the nearest neighbor to X ; hence, do not compute $d(X, X_i)$; otherwise, compute $d(X, X_i)$. If $d(X, M_p) < B$, set Current_NN = *i*, and $B = d(X, X_i)$. After all the X_i's in the current node are tested, go to step 2.

At the termination of the algorithm, the nearest neighbor is given by Current_Nearest_Neighbor and its distance to *X*, by *B*.

- Extension to *k*-nearest neighbors

Extension to *k*-nearest neighbors is straightforward, taking *B* the distance to the current *k*th nearest neighbor. When a distance is actually calculated in Step 5, it is compared against the distances from *X* to its current *k*-nearest neighbors, and the current *k*-nearest neighbor table is updated as necessary, discarding the sample from the table that is now the *k*+1 th nearest neighbor to *X*.

The power of BnB in saving computation time is obvious: "Typically, as average of only 61 distance computations were made to find the nearest neighbor of a test sample among 1000 design samples" [18].

5.2 Using Branch and Bound in GMR

BnB method is powerful, but how to implement it in the GMR network? Will it require an extra computation load to generate the weight tree for searching? The answer lies both in the experimental results and in its analysis. The use of BnB in the linking and the learning phase will be examined in the following sections.

5.2.1 Implementation of Branch and Bound for linking

The most time consuming phase in GMR is linking. Remember that in linking phase, all the inputs are presented to the pool of *p* neurons, before the directional similarity test is done ; candidates should be chosen by computing all the distances from the input data to each neuron, and the list of distances is sorted so as to find the winner neuron and the *k*-nearest neighbors.

Hence the goal of using BnB here becomes the pruning of the unnecessary computations of distance $d(z, w_i)$, where z is every input in the TS, w_i is ith weight of the pool of final neurons.

5.2.1.1 Creation of the Neuron Tree during Learning Phase

Here we don't need to do the extra clustering in order to create the tree. It's more straightforward to generate the tree by making use of the existing hierarchical information (objects and neuron sets) during the coarse to fine vector quantization procedure.

The neuron tree can be automatically created during learning. However, two adjustments in the parameters are involved here: (1) the sample mean of the subset M_p becomes the corresponding weight of the object neuron created in its upper level quantization, $W_{obj(p)}$; (2) r_p, the farthest distance from M_p to an $X_i \in S_p$, turns to the farthest distance from the $W_{obj(p)}$ to an input data p for which it wins during the labelling procedure.

A tree created in this way is represented in Fig 5.1. The number of levels in the tree is the same as the levels of EXIN SNN's used for learning.

5.2.1.2 Using Branch and Bound Search in Linking Phase

In the following is the modification of the linking procedure in order to use BnB.

For each presentation of an input z to the pool of neurons, instead of computing all the distances and then sorting, only the k-nearest neighbors of an input are computed by using a sub function named *nearestNb* which uses the BnB search algorithm. The algorithm in *nearestNb* is similar to the general k-nearest neighbors algorithm which has been introduced previously. Furthermore it can get the nearest neighbors that are within a sphere centered on the input data : it adds a filter (simply with if-then rules) to eliminate the neurons whose distances are larger than one radius threshold. Although the computation which can be saved by using *nearestNb* for each input presentation is limited, the accumulative saving becomes huge when the size of TS is large.

Recall that in the basic GMR algorithm, before doing the direction test, the distance test is done in order to select the candidates. In BnB version, two ways to select the candidates are listed:

- One way requires two neuron filtering steps:
  1. Use *nearestNb* to calculate k-nearest neighbors of every input, and the k nearest distance to the input data should be no larger than one radius threshold $\delta\rho_f$, where δ is the linking factor, a predefined constant greater than 1. ρ_f is the vigilance threshold for the final level in the EXIN SNN learning.

2. Further remove the neurons whose distances to the input data are larger than δd_1, where d_1 is the nearest distance to to input data.

For convenience of notation, this way to use BnB is called *δ-BnB* method.

- An alternative way to use BnB requires only one step of neuron filtering, that is it simply computes the *k*-nearest neighbors to the input data, where *k* is a predefined constant. This variant of using BnB is referred to as *k-BnB* method in this thesis.

Using the *δ-BnB* method can not guarantee that every presentation of an input data can create a linking between two neurons (either creating a new link or strengthening an existing link by 1), just like in the general linking method cited previously. For example when the first winner neuron and the input data are too close (but not identical), if the δ is not large enough, linking holes (a "hole" here means that there is no link between two neurons which should be linked) likely appear, as shown in Fig. 5.3(a). It's often the case in the sparse data region. On the other hand, it can avoid inappropriate linking between distant neurons.

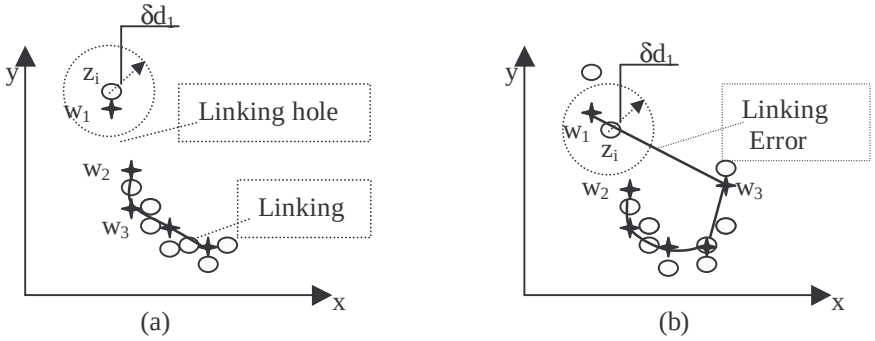


Fig. 5.3 Example of linking mistake that appears when using δ-BnB or *k*-BnB in candidate steps. The solid circles represent data points of training set. The star marks represent neurons. Solid lines denote linking. The dashed circle shows a hypersphere within which linking candidates are selected.

(a) As δd_1 is too small, no linking candidates are found, this presentation of z_i fails to generate a linking. If *k*-BnB is used in candidate steps, *k* (>1) nearest neighbors are chosen as linking candidates ; one linking from w_1 to another neuron (w_2 or w_3) is guaranteed to be added after the directional test.

(b) An inappropriate linking is added between w_1 and w_3 when using *k*-BnB in candidates steps.

Here *LinkRate* is introduced in order to evaluate the success rate that an input presentation can contribute a linking creation. As we can see shortly, it can somehow reflex the efficiency of the linking procedure.

$$LinkRate = \sum_{i,j=1}^{p} V_{ij} \Big/ (2N) \qquad (5.1)$$

where *V* is the linking matrix, *p* is the number of neurons, *N* is the size of the TS. Apparently the *LinkRate* in δ-BnB method could be less than one.

As for the *k*-BnB method, each presentation of an input data certainly causes a linking in normal cases when *k* and *N* are greater than 1. Thus the *LinkRate* in using *k-BnB* method is normally 1. However, as shown in Fig 5.3 (b), this method also suffers from the undesired linking between distant neurons.

Further discussion of these two methods will be given in the following experiment section.

5.2.1.3 Experimental Results

Experiments in this section are given in order to see how efficient in saving computation time is the use of BnB in GMR, and to examine the factors which influence this efficiency, such as the levels in learning, the size and dimensionality of the TS, the number of neurons. It has also given a comparison between different methods of application of BnB in linking.

5.2.1.3.1 Measurements

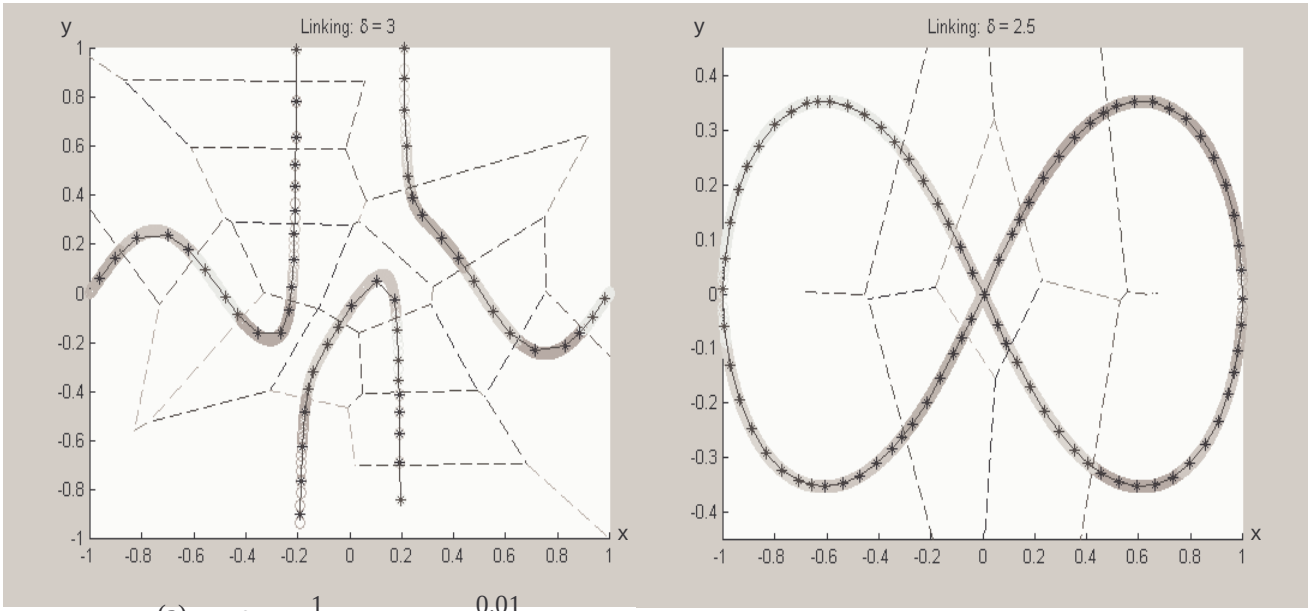
Here the computation time is measured by the variable *flops*, i.e. the cumulative number of floating-point operations according to the run time data given by Matlab 5.x. Unlike the *elapsed time* which is machine-related, the *flops* is independent from the computer system. Therefore, although the program for this thesis is only implemented in Matlab 5.2, the *flops* is still meaningful when the algorithm is implemented in the other development systems.

The first experiment requires the approximation of the 2-D function $y = f(x) = \frac{1}{4}(\sin(2\pi x) + \frac{0.01}{x^2 - 0.04})$. The TS consists of 1974 data points (x, y), whose components are within the range of [-1, 1]. The second 2-D example's TS is composed of 4000 points (x, y), being $(x^2+y^2)^2 = x^2-y^2$. The third TS is composed of 3000 data points drawn from the surface of 8 spheres in 3-dimensions. For each example, 3 different methods under 2 categories will be tested and compared. One category uses 2 levels EXIN SNN in learning phase, the other uses 3 levels. Each category has 3 variants:

1) not use Branch and Bound searching;
2) use δ-*BnB* method in linking phase;
3) use *k-BnB* method in linking phase.

The data sets and final neurons and linkings for the first 2-D example are plotted in Fig 5.4. The results of the 3-D example are plotted in Fig 6.11 in the simulation section of Chapter 6. The results of the experiments are shown in Table 5.1. Some important parameters which are marked with '*' in the tables are defined in the following list:

1.	*Flops difference*	Difference in flops between not using branch and bound and using branch and bound.
2.	*LinkErrorRate*	A measure of the linking quality. *LinkErrorRate= LinkError / IdealLinksNumber*. where *LinkError* is defined as the difference between the experiment result and the ideal estimation, in number of neurons which are linked to a certain neuron.
(a)	for 2-D example 1	The *ideal links number* (not strength) for each neuron should be 2 for the intermediate neurons and 1 for the terminal neurons. (i.e. each neuron has two or one neurons being linked to it.)
(b)	for 2-D example 2	The *ideal links number* for each neuron should be 2 for all the neurons, except 4 for the centroid neuron.
(c)	for the 3-D example	Ideally, each neuron has 3 or 4 other neurons being linked to it (3 is assumed here). Hence, *IdealLinksNumber* in exps. 1.0, 1.1, 1.2 is 3×625 (625 is the number of final neurons), in exps. 2.0, 2.1, 2.2 is 3×626.
3.	*Link Rate*	Defined as the success rate that an input presentation can cause creation or strenthening of a link. See eq. (5.1)



(a) $y = f(x) = \frac{1}{4}(\sin(2\pi x) + \frac{0.01}{x^2 - 0.04})$

(b) $(x^2+y^2)^2 = x^2-y^2$

Fig. 5.4 Data sets and resulting neurons and linkings of experiments in Table 5.1 (a) and (b).
Figs. (a) and (b) show the final neuron weights (stars), links (black segments), and Voronoi tessellation of the input space before final quantization. Data points belonging to different objects are represented by circles with different grey scale within a dashed-line region.

Table 5.1 Using Branch and bound in GMR linking

(a) 2-D example: TS is composed of 1974 points (x, y), $y = f(x) = \frac{1}{4}(\sin(2\pi x) + \frac{0.01}{x^2 - 0.04})$, x, y∈[-1, 1]

Levels of EXIN SNN / Computation Time (flops: $\times 10^5$)		2-level [ρ1=0.2,epoch1=2 ; ρ2=0.07, epoch2=2 ; ρ_{min}=0.6ρ2, δ=2] [level1 neuron :19, level2 neuron : 56], [Objects after merging: 3]			3-level [ρ1=0.5,epoch1=2; ρ1=0.2,epoch2=2; ρ3=0.07,epoch3=2] ;[ρ_{min}=0.6ρ3,δ=2] [level1 neuron :5 , level2 neuron :22 , level3 neuron :55], [Objects after merging: 3]		
branch and bound / linking candidates (distance test)		1.0 No δ=3	1.1Yes δ-BnB δ=3	1.2 Yes k-BnB k=2	2.0 No δ=3	2.1 Yes δ-BnB δ=3	2.2 Yes k-BnB k=2
learning flops		7.72	7.74	7.74	7.55	7.55	7.55
linking flops		*8.01*	*4.47*	*4.59*	*7.86*	*5.44*	*5.77*
merging flops		2.58e-3	2.63e-3	2.63e-3	2.96e-3	2.97e-3	2.97e-3
Total Flops		15.73	12.21	12.34	15.42	12.99	13.32
Flops diff *[1]	num	---------	3.52	3.40	---------	2.42	2.09
	rate	---------	**22,37%**	**21,58%**	---------	**15,71%**	**13,57%**
Link Error Rate*[2]		0	0	0	0	0	0
Link Rate*[3]		0.4995	0.5000	1	0.4904	0.5638	1

Table 5.1

(b) 2-D example: TS is composed of 4000 points (x, y), where $(x^2+y^2)^2 = x^2-y^2$.

Levels of EXIN SNN / Computation Time (flops: $\times 10^5$)		2-level [ρ1=0.2, epoch1=2 ; ρ2=0.035, epoch2=2 ; ρ_{min}=0.5ρ2, δ=2] [level1 neuron :15, level2 neuron : 83], [Objects after merging: 1]			3-level [ρ1=0.5,epoch1=2; ρ1=0.2,epoch2=2; ρ3=0.035, epoch3=2] ; [ρ_{min}=0.5ρ3, δ=2] [level1 neuron :4 , level2 neuron :16 , level3 neuron :84], [Objects after merging: 1]		
branch and bound / linking candidates (distance test)		1.0 No δ=2.5	1.1Yes δ-BnB δ=2.5	1.2 Yes k-BnB k=2	2.0 No δ=2.5	2.1 Yes δ-BnB δ=2.5	2.2 Yes k-BnB k=2
learning flops		14.27	14.28	14.28	14.88	14.88	14.88
linking flops		*23.62*	*8.48*	*9.63*	*23.97*	*10.81*	*12.77*
merging flops		3.26e-3	3.26e-3	3.26e-3	3.35e-3	3.38e-3	3.38e-3
Total Flops		37.89	22.77	23.92	38.85	25.69	27.66
Flops diff *[1]	num	---------	15.12	13.98	---------	13.15	11.19
	rate	**---------**	**39,90%**	**36,88%**	**---------**	**33,86%**	**28,80%**
Link Error Rate*[2]		0	0	0	0	0	0
Link Rate*[3]		0.3346	0.4373	1	0.4308	0.4455	1

Table 5.1
(c) 3-D example: TS is composed of : 3000 points on the 8 spheres.

Levels of EXIN SNN / Comp-Time (flops: $\times 10^5$)		2-level [ρ1=0.2,epoch1=1, ρ2=0.07,epoch2=2] [ρ_{min}=0.6ρ2] [level1 neuron :62, level2 neuron :625] [Objects after merging:8]			3-level [ρ1=0.5,epoch1=2,ρ2=0.2,epoch2=1, ρ3=0.07,epoch3=2], [ρ_{min}=0.6ρ3] [level1 neuron :8 , level2 neuron :57, level3 neuron :626], [objects ater merging:8]		
branch and bound / linking candidates (distance test)		1.0 No δ=3.5	1.1Yes δ-BnB δ=3.5	1.2 Yes k-BnB k=4	2.0 No δ=3.5	2.1 Yes δ-BnB δ=3.5	2.2 Yes k-BnB k=4
total learning flops		36.44	36.44	36.44	34.40	34.41	34.41
linking flops		*198.66*	*47.15*	*37.95*	*198.93*	*38.86*	*32.84*
merging flops		0.01804	0.01747	0.01748	0.01749	0.01712	0.01718
Total Flops		235.12	83.61	74.41	233.35	73.29	67.27
Flops diff *[1]	num	---	151.51	160.71	---	160.06	166.08
	rate	---	**64,44%**	**68,35%**	---	**68.59%**	**71.17%**
Link Error Rate*[2]		0.1141	0.1083	0.0501	0.1076	0.1134	0.0474
Link Rate*[3]		0.7114	0.7121	1	0.7152	0.7165	1

5.2.1.3.2 Discussion

Based on the above experimental results, several questions could be answered analytically without serious mathematical proof.

- *Will the tree creation add extra computation load?*

The coarse to fine learning under the GMR strategy already provides a hierarchical structure of neurons; the extra effort in building a tree can be neglected. Even if the tree has to be built via another clustering process, this preprocessing is still inexpensive in terms of computation time in comparison with the time, which could be saved in the searching, as will be discussed in the next section.

- *How efficient the BnB technique could be in speeding up the GMR?*

Very powerful, and the efficiency depends on the size and dimension of TS and also on the number of neurons.

For example in the second 2-D example, the total flops are reduced by around 40% compared with not using BnB. If only considering the linking phase, the percentage of flops which can be saved by using BnB is even more apparent: in the first 2-D example the best result is 44% flops being saved in linking, in the second 2-D example (the size of TS is double to the first one) 64%, in the 3-D example 83%.

- *How the levels of searching affect the efficiency of BnB?*

The level of searching tree here is the same as the level of EXIN SNN used in learning. Thus the level selection will not only affect the computation time in linking phase, but also in learning phase. Based on the result in Table 5.1, it seems that for the small 2-D data set, a 2-level learning, which also leads to a 2-level tree searching is optimal. While for a larger data set, or a higher dimensional data set, 3 levels might be needed. It is also suggested, that [17] the optimal setting for the BnB searching involves a compromise between the number of final neurons belonging an object neuron (i.e. the size of the final subset of the neurons), and the number of intermediate neurons.

- *How the δ-BnB method differs from the k-BnB method in the linking phase?*

As mentioned previously, this experiment result shows that the link rate of using *k*-BnB is always 1, while δ-BnB is less than 1. However, higher success linking rate can also imply longer computation time, if the actual linking is not so dense. As can be seen in the 2-D examples, the flops used by *k*-BnB are always higher than δ-BnB, while the linking error rate is the same as the δ-BnB.

In the 3-D example, according to the linking error rate, *k*-BnB can yield better linking result than δ-BnB and less computation time requirement as well. This suggests that in higher dimensional space with a certain linking distribution (e.g. in this 8-sphere 3-D example), the *k*-BnB might outperform δ-BnB both in computation time and linking quality.

Note that the *linking error rate* is a measure of linking quality. Here, for sake of simplicity, only the linking hole is considered as an error; other inappropriate linking is ignored (actually the occurence of other kinds of undesired linking is rather rare in these experiments for a suitable linking factor δ).

5.2.2 Implementation of Branch and Bound in Learning

As shown above, the branch and bound method can facilate rapid computation of the nearest neighbors, which is required not only in linking, but also in the learning phase, e.g. each EXIN SNN and each relabelling procedure. Thus the BnB should be able to applied in learning also. Since EXIN SNN is incremental, only the relabelling procedure will be considered here.

5.2.2.1 Using Branch and Bound in Relabelling

The relabelling might require more than half of computation time of the total learning time when the data set is not small; for example, in the above 3-D experiment, for 2 level EXIN SNN, 53% of the total learning time is used in relabelling, and for the 3 level, 59%.

Unlike the linking phase, a hierarchical neuron structure is not available before relabelling, thus an extra tree creation procedure is needed before using BnB. Again,

the function *nearestNb* will be employed to locate the nearest neuron to the presented input vector among the pool of neurons in the current level learning. Here, two methods are tried for creating the neuron tree structure:

1) Hierachically decompose the neuron set by *k*-means-clustering, keeping *k* a predefined constant for each level. Store the neccessary information (as introduced in section 5.1) in a tree structure. The tree created in this way is symmetric. The k-means algorithm chosen for this experiment is in the batch version (Lloyd, 1982) [4]:

   > It begins by assigning the points (neurons) at random to *k* sets and then computing the mean vectors of the points in each set. Next, each point is re-assigned to a new set according to which is the nearest mean vector. The means of the sets are then recomputed. This procedure is repeated until there is no further change in the grouping of the data points.

2) Hierachically divide the neuron set by EXIN SNN vector quantization in coarse-to-fine manner as described in Section 4.2.1. The training set corresponds to the set of neuron weights. The vigilance threshold ρ is predefined for each level. Construct the tree using the same procedure as described in Section 5.2.1.1 by recording the neccessary parameters.

5.2.2.2 Experimental Results

Table 5.2 Using k-means / EXIN SNN to construct neuron tree structure for applying BnB in labelling.

Levels of EXIN SNN / Comp-Time (flops: $\times 10^5$)		[ρ1=0.2,epoch1=1, ρ2=0.07,epoch2=2] [ρ_{min}=0.6ρ2] [level1 neuron :62, level2 neuron :625] [Objects after merging:8]			Levels of EXIN SNN / Comp-Time (flops: $\times 10^5$)		[ρ1=0.5,epoch1=2,ρ2=0.2,epoch2= 1, ρ3=0.07,epoch3=2], [ρ_{min}=0.6ρ3] [level1 neuron :8 , level2 neuron :57, level3 neuron :626], [objects ater merging:8]		
BnB label 1 tree creation		1.0 No	1.1Yes k-means k=7, level=2	1.2 Yes exinSnn [ρ1=0.8, ρ2=0.5], level=3	BnB label 2 tree creation		2.0 No	2.1 Yes k-means k=6, level=2	2.2 Yes exinSnn [ρ1=0.8, ρ2=0.5] level=3
PLN 1		10.23	10.23	10.23	PLN 1		4.65	4.65	4.65
La-bel 1	crtTree	---------	0.35	0.20	Label 1		2.57	2.57	2.57
	Label		9.40	8.88	PLN 2		2.07	2.07	2.07
	Total	19.51	9.78	9.08	**La-bel 2**	crtTre	---------	0.29	0.18
	Diff %	**---------**	**49,88%**	**53,44%**		Label	---------	8.99	8.19
PLN 2		6.71	6.71	6.71		Total	17.94	9.28	8.37
Total		36.44	26.72	26.02		**Diff**	**---------**	**48,27%**	**53,34%**
Total Diff %		---------	**26,70%**	**28,60%**	PLN 3		7.17	7.17	7.17
					Total:		34.41	23.67	22.86
					Total Diff %		**---------**	**31,19%**	**33,84%**

The number of labelling flops for each method shown in Table 5.2. is the best result among more than 10 trials with different parameter settings. As we can see, EXIN SNN method requires less computation time to construct the tree than *k*-means method. The BnB searching in the tree structure created by EXIN SNN method is also more efficient than in the tree constructed by *k*-means method. Furthermore, the time consumed by *k*-means clustering to construct a tree is not fixed in every trial, sometimes the clustering fails due to the existence of 'mean vector' which can never win, because of the random initialization of *k* means. Frequently its result is not so good even with the same parameter (*k* and level) settings. In contrast, EXIN SNN method is faster and its performance is rather stable by avoiding the random initialization procedure.

Note that BnB is not used in the first relabelling in the 3-level learning experiment, since the number of neurons in level one is too small : using BnB is not so economical in this case.

5.3 Conclusions

The summary of this chapter is in the following:

- Branch and bound is a powerful searching technique. It can facilitate rapid calculation of the *k*-nearest neighbors, by eliminating the necessity of many distance computations.
- By making use of the hierachical neuron information after GMR learning, a tree structure can be created without extra computation load. Thereafter, branch and bound search can be applied in linking phase to save the computation time in searching the neuron candidates for linking.
- Either δ-BnB method or *k*-BnB method can be chosen in linking candidate steps for different cases.
- After decomposing the neuron set by the above *k*-means clustering or EXIN SNN, branch and bound can be used to accelerate the relabelling procedure. In the above experiments, the EXIN SNN method exhibits better performance than *k*-means method in facilating fast BnB searching.
- The efficiency of branch and bound is proportional to the size and dimension of the data set, and is related to the number of the nodes (neurons) in the searching tree structure.

Chapter 6

Experiments

In order to apply the algorithms described earlier to real problems, we need to address a variety of practical issues. In this chapter, the parameter setting of the GMR network will be discussed accompanied with a series of experiments. The properties of GMR, such as its ability to map data set which have sparse regions and are noisy, are to be studied. Other examples of mapping approximation by GMR will be also be given.

6.1 Learning

In the learning phase, the important settings include stopping criterion, vigilance threshold, the number of levels and neuron cleaning. Note that all the experiments and analysis in this section are based on EXIN SNN training; if other mapping algorithms are used, the parameter setting might be different.

6.1.1 Stopping Criteria

Two stopping criteria for an individual EXIN SNN training under a certain vigilance threshold ρ are considered both from the computational efficiency and mapping accuracy point of view.

- One possibility is to stop after a fixed number of epochs. This method is chosen in all the other experiments for this thesis. Although a rough idea can be obtained from some preliminary test, the main problem arises when several networks are being trained (e.g. in the second level vector quantization, several EXIN SNN's are trained individually with different number of inputs and neurons); hence the appropriate number of epochs may be different for different networks.
- The other way is to stop when the relative change in weight adjustment falls below a specific value. Here this criterion is referred as ∇-stop criterion in this thesis, and the weight change, called the ∇_{max}, for one epoch is calculated according to the following formula:

$$\nabla_i = \sum \left\| w_i^{(new)} - w_i^{(old)} \right\|_2 \qquad (6.1)$$

$$\nabla_{\max} = \max_{i=1}^{p} \nabla_i \,, \qquad (6.2)$$

where ∇_i represents the total increment of the Euclidean distance in weight for neuron i in an epoch and ∇_{max} represents the maximun ∇_i among all the p neurons in one epoch.

Although the ∇-stop can make us more confident on the stability of the quantization result than the previous criterion, it may decrease the computational efficiency due to the distance computations for each iteration, especially when the data set is huge or the dimensionality is high.

In order to illustrate this idea, consider the problem of fitting the function:

$$y = f(x) = \frac{1}{4}(\sin(2\pi x) + \frac{0.01}{x^2 - 0.04}) \qquad (6.3)$$

where the TS is given by 1974 points (x,y) uniformly distributed in the interval of [-1,1], by using a single EXIN SNN training, in which the vigilance threshold is 0.3 and the maximum number of epochs is 30. In this example, as shown in Table 6.1, the computational time is measured by flops; the matching result (or clustering result) can be evaluated by ∇_{max} and the value of the sum of squares clustering function defined as:

$$SSD = \sum_{i=1}^{p} \sum_{n \in S_j} \left\| z^n - w_j \right\|^2 \qquad (6.4)$$

where p is the number of neurons in the network. The data set $\{z^n\}$is partitioned into p subset S_j (j = 1, ..., p) by relabelling according to the closest neuron, whose weight w_j is the jth weight among the p neuron weights. For convenience of notation, this value is referred to as *SSD*, i.e. the sum of squares distance from the input data to the corresponding winner weight.

Table 6.1 Comparison of two stopping criteria in EXIN SNN: fixed-epoch vs. ∇-stop.

epochs	# neurons	∇_{max}	*SSD*	fixed-epoch flops ($\times 10^5$)	∇-stop flops ($\times 10^5$)	diff flops ($\times 10^5$)
1	9	0.8618	51.3976	0.85	1.00	0.16
2	11	0.5607	39.7862	2.38	2.70	0.31
3	13	0.2353	36.0730	4.20	4.67	0.47
4	13	0.0850	35.0552	6.15	6.79	0.63
5	13	0.0528	34.3856	8.11	8.90	0.79
10	13	0.0185	32.8132	17.89	19.46	1.58
23	14	0.0085	31.4117	43.37	47.00	3.63
30	14	0.0076	31.0809	58.01	62.75	4.74

In the final column of Table 6.1, *Diff Flops* reports the difference between the number of flops required by the fixed-epoch criterion and the ∇-stop criterion. This table shows the mapping result after 5 epochs of EXIN SNN training.

It's apparent, that more epochs of training yield higher accuracy because the *SSD* decreases monotonously with increasing number of epochs. However, the computation time grows as well. In order to achieve the same accuracy, the fixed-epoch criterion requires less flops than the ∇-stop method. On the other hand, after three epochs, the number of neurons in the network remains unchanged untill the 23rd epoch, when another neuron is created. Furthermore the value of ∇_{max} falls quickly from 0.86 to 0.085 during the first three epochs; after that, the decreasing of ∇_{max} slows down quickly as well. The same decreasing trend also happens to *SSD*. This suggests, the accuracy of the mapping changes little after a certain number of epochs. Therefore, a trade-off between the accuracy and the computational efficiency should be reached.

Based on the above empirical reason, in the later experiments the fixed epoch criterion is used to save the computation time, and only 1-5 epochs training is done according to the accuracy requirement. Practically, for the coarse quantization, 1-3 epochs are enough, while in the fine quantization, more epochs might be needed for better mapping accuracy.

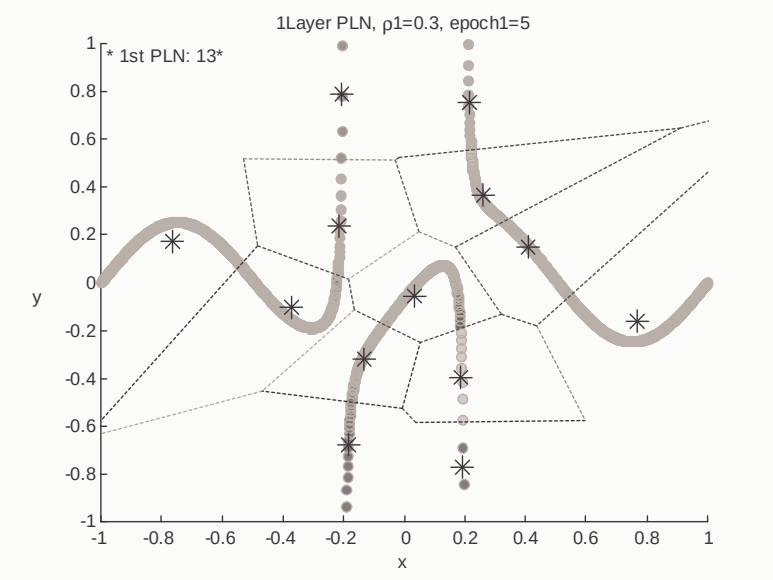


Fig. 6.1 Mapping result of a single level EXIN SNN after 5 epochs training. TS is composed of 1974 data points (x,y): $y = f(x) = \frac{1}{4}(\sin(2\pi x) + \frac{0.01}{x^2 - 0.04})$, uniformly distributed in the interval [-1, 1]. Circles represent input data; stars represent neurons, dashed lines represent the edges of the Voronoi tessellation.

6.1.2 Vigilance Threshold In Single Level Learning

Considering only one level learning, we might compare the influence of vigilance threshold setting under two approaches: fixed ρ and decaying ρ for epoch increasing. Define the vigilance threshold in the first epoch as ρ_1 and the final epoch as ρ_f, then $\rho_1 = \rho_f$ for the fixed-ρ method, and $\rho_1 > \rho_f$ for the decaying-ρ method. In the next experiment, the decaying ρ has the following form :

$$\rho_{t+1} = \rho_t . (\frac{\rho_f}{\rho_t})^{t/f} \qquad (6.5)$$

where t is the epoch number during training and f is the final epoch number.

In order to see how the vigilance threshold affects the mapping acurracy and computation efficiency, the same example as before (data set is shown in Fig 6.1) is considered. Except the ∇_{max}, the other measures to the computation time and mapping accuracy are the same as the above ones. Table 6.2 gives the experiment results for different ρ settings. The maximum number of epochs is 5 and the results of the mapping are recorded after each epoch. The *SSD* and the computation time under different ρ settings with regard to the epoch number are plotted in Fig 6.2 (a) and (b), respectively.

Table 6.2 Mapping results with different vigilance threshold ρ settings in a single EXIN SNN: fixed ρ versus decaying ρ

	fixed ρ (0.25)					*decaying ρ (0.5~0.25)*				
epoch	1	2	3	4	5	1	2	3	4	5
# neurons	12	16	18	18	18	5	9	15	16	17
SSD	37.15	19.94	17.56	16.97	16.62	137.44	63.41	33.01	28.33	23.49
flops ($\times 10^5$)	1.00	3.13	5.64	8.29	10.93	0.57	1.78	3.55	5.84	8.34

	fixed ρ (0.375)					*decaying ρ (0.375~0.25)*				
epoch	1	2	3	4	5	1	2	3	4	5
# neurons	8	10	12	12	12	8	10	14	16	16
SSD	56.11	39.41	35.03	33.98	33.35	56.11	39.41	28.59	24.08	22.47
flops ($\times 10^5$)	0.78	2.18	3.86	5.67	7.49	0.78	2.18	4.05	6.20	8.57

The shadow square areas in Table 6.2 suggest that the fixed ρ method outperforms the decaying method both in the computaional efficiency and mapping accuracy for a single EXIN SNN training. The mapping accuracy of each epoch depends mainly on the vigilance threshold of that epoch. The similar analysis about the trade-off between the computational efficiency and mapping accuracy can also be applied to this experiment. Hence, only the fixed ρ method is used in the remaining experiments of this chapter.

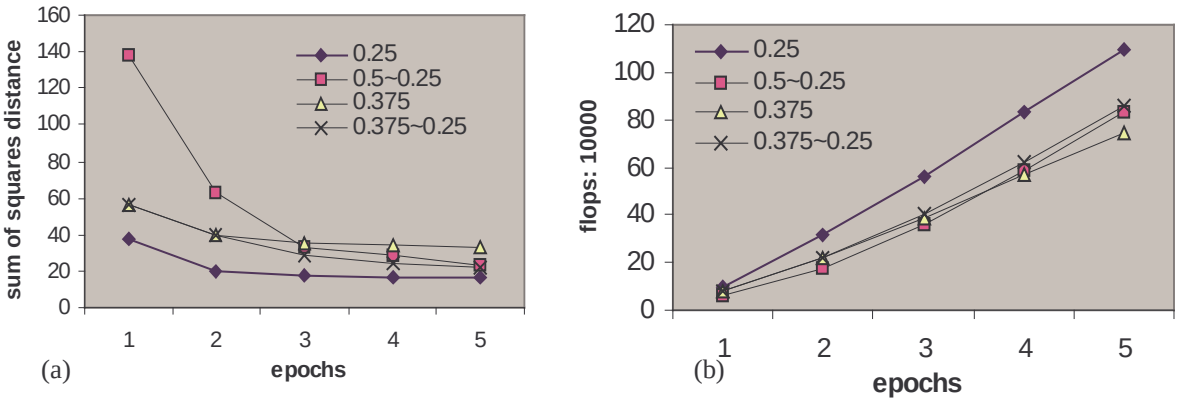


Fig. 6.2 Mapping result of different vigilance threshold settings (fixed or decaying) after each epoch in a single EXIN SNN training. Different types of marks on the line represent different vigilance threshold ρ setting. The mapping accuracy in sum of squares distance (value of function (6.4)) is shown in Fig (a), the computational time in flops ($\times 10^5$) is shown in Fig (b).

6.1.3 Multilevel Learning

Table 6.3 Results of multilevel EXIN SNN's learning (* *acc flops* : accumulated flops ($\times 10^5$)).

(a) Results for different levels of EXIN SNN's learning in a 2-D example

	1 level	*2 level*		*3 level*			*4 level*			
level	1	1	2	1	2	3	1	2	3	4
ρ	.07	0.2	.07	0.5	0.2	0.07	0.5	0.25	0.15	0.07
epoch	2	2	2	2	2	2	2	2	2	2
# neurons	55	20	66	7	24	67	7	21	34	66
SSD	2.43	14.18	1.50	84.99	13.21	1.52	84.99	17.28	6.49	1.61
flops ($\times 10^5$)	10.67	3.95	3.86	1.53	2.01	4.39	0.53	1.95	3.72	5.60
acc* flops	10.67	3.95	7.81	1.53	3.54	7.93	0.53	2.48	6.20	11.80

Table 6.3
(b) Results for different levels EXIN SNN's learning in a 3-D example

	1 level		*2 level*							
	exp1	exp2	exp3		exp4		exp5		exp6	
level	1	1	1	2	1	2	1	2	1	2
ρ	0.07	0.07	.3	.07	.2	.07	0.15	.07	.15	.07
epoch	1	2	2	2	2	2	1	1	1	2
# neurons	628	641	21	668	74	709	124	720	124	733
SSD	4.55	4.27	119.76	4.20	36.81	3.91	24.04	3.99	24.03	3.72
flops ($\times 10^5$)	108.98	333.35	11.08	32.27	36.38	32.32	21.85	45.38	21.85	47.92
* acc flops	108.98	333.35	11.08	43.35	36.38	68.70	21.85	67.23	21.85	69.77

	3 level						*4 level*			
	exp7			exp8			exp9			
level	1	2	3	1	2	3	1	2	3	4
ρ	.5	02	.07	.5	0.15	.07	0.5	0.25	0.15	0.07
epoch	2	2	2	2	1	2	1	1	1	1
# neurons	8	74	709	8	124	733	8	28	151	737
SSD	194.98	36.81	3.91	194.98	24.03	3.72	194.98	112.82	21.73	3.94
flops ($\times 10^5$)	5.18	8.54	32.35	5.18	6.49	47.95	1.97	4.10	12.21	54.82
* acc flops	5.18	13.73	46.08	5.18	11.67	59.62	1.97	6.07	18.28	73.11

In this experiment section, we hope to get some empirical ideas about how to set up the number of levels and the vigilance threshold for each level, so as to achieve a 'good' result which has the compromise between the mapping accuracy and the computational efficiency.

Here, two data sets are considered. The first one is still the same 2-D data set as in the previous section. The second data set is composed of 3000 3-D points scattered on 8 spheres. For comparison, the final vigilance threshold ρ_f is fixed to 0.07 (both in 2-D and 3-D example). The number of levels ranges from 1 to 4. The vigilance threshold in the intermediate level can be chosen freely. Both the results of the intermediate levels and final level are shown in Table 6.3. These results are the 'good' ones selected

by trial and error with different parameter settings, the 'best' final results for each number of levels are in the shadow areas and plotted in Fig 6.3. The epochs listed below are limited to less than 3 for sake of simplicity. In practice more epochs can be used, provided the computational resources are enough.

As shown in Table 6.3, for these examples two or three levels of learning outperform single level and four levels of learning both in the mapping accuracy and the computational efficiency. However, the computational time is sensitive to the vigilance threshold and epoch setting. Although it's difficult to get an optimal setting at once, we can consider an acceptable range of settings: it seems that the vigilance threshold ρ_t should not be less than half of ρ_{t+1}.

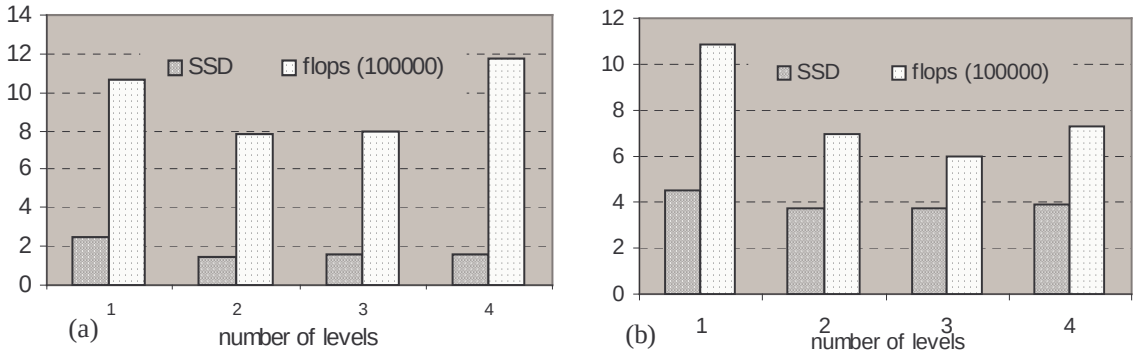


Fig. 6.3 Comparison of different mapping accuracies measured by *SSD* and computation time measured by flops ($\times 10^5$) using different number of levels in EXIN SNN's learning. (a) Results in a 2-D example (b) Results in a 3-D example.

6.1.4 Neuron Cleaning

As mentioned in Chapter 3, EXIN SNN can use a cleaning algorithm in order to maintain the regularity of the mapping. If the Euclidean distance between two neurons falls below the vigilance threshold, the neurons are merged according to equation (3.2). This kind of neuron cleaning can be done after each single EXIN SNN training. The levels to be cleaned can also be chosen. Furthermore, as a consequence of decreasing the number of neurons, the computation time for its later processing might reduce as well.

However, the regularity effect by neuron cleaning can be achieved by increasing the vigilance threshold, i.e. decreasing the mapping resolution. This will be shown in section 6.4, where a noisy training set is dealt with.

6.2 Linking and Merging

6.2.1 Linking Factor and Merging Threshold

Linking results have great effect on the result of merging. Two parameters are involved here:

- The first is the linking factor δ. The larger the δ, the more neurons might be selected to do direction test. Hence δ should not be too large, otherwise undesired linkings (e.g. linking between two neurons far away from each other and belonging to two different branches) are easier to be added. On the other hand, δ should not be too small, otherwise there will be no enough linkings as it should, since too many neuron candidates for linking are already excluded before doing the direction test.
- The merging threshold χ, is used to decide how strong the inter-object linking can force the two objects to be merged. In practice, χ is generally set to one, which implies the existence of a linking between two objects is the only prerequisite for merging. This is the case for all the examples shown in this thesis. Although higher χ can be used, setting χ to 1 can normally get good merging result for data set without much noise.

Table 6.4 Linking and merging results (χ=1) of different δ settings.

linking factor δ	number of neurons that are connected with *n* neurons							link rate	# branches after merging
	n=0	n=1	n=2	n=3	n>=4	n < ideal linking number	n > ideal linking number		
ideal	0	6	77	0	0	0	0	--------	3
1,2	7	18	57	0	0	19	0	0,1003	5
1,5	5	16	61	0	0	15	0	0,2062	5
2	2	16	64	0	0	12	0	0,2852	3
2,2	1	14	67	0	0	9	0	0,3749	3
2.4	0	7	68	7	0	1	7	0,4134	3
2,5	0	5	66	11	0	0	12	0,4306	3
2,8	0	5	55	21	1	0	23	0,4762	3
3	0	7	47	25	3	1	28	0,5051	4
4	0	7	31	25	19	1	44	0,6039	4
6	0	5	18	18	41	0	59	0,7264	3

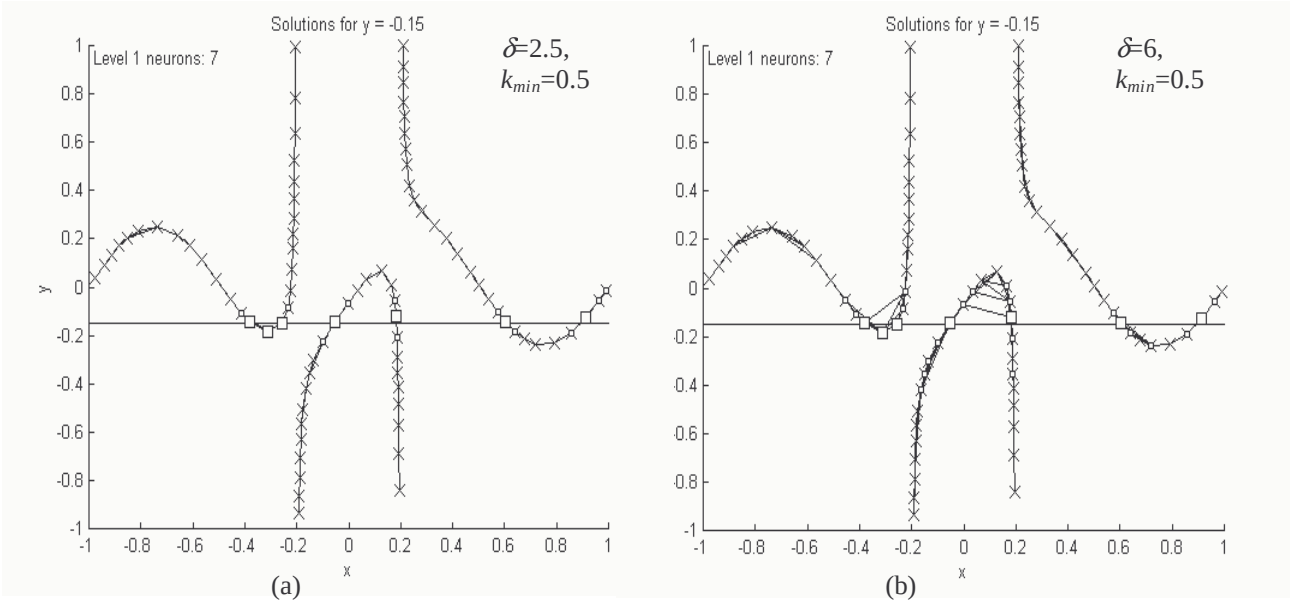
Table 6.4 shows a series of linking distributions with different δ setting, for the same 2-D mapping problem as in the previous section. The final vigilance threshold is set to 0.05, the number of final neurons is 82.

Linking should carry the topological information of the data set. For the current problem the prior knowledge of the data set is available, and an ideal linking result can be estimated (see Table 5.1, *ideal linking number*). We will compare the actual linking results with the ideal ones.

Consequently, 2.5 is considered to be an optimal δ setting for this mapping. Practically the normal range of δ is 2 ~ 4. For higher accuracy mapping, δ can be much more flexible, while for mapping of lower accuracy the result is more sensitve to the δ value.

Although some redundant links may exist in this network, it's not a serious problem. As can be seen in Fig 6.4, even when δ is set to 6, it can still provide the same level one neurons as setting δ = 2.5. Furthermore, the redundant linkings can be pruned by an additive linking pruning procedure.

Another issue related to the δ setting is the computational time. Increasing *linking rate* (for definition see Table 5.1) reveals that the larger δ setting allows more candidates to participate in the direction test; thus more computation time is required, especially for large training sets.



(a) (b)

Fig. 6.4 Comparison of the linking and recalling results under different δ settings.

TS is 1974 points (*x*,*y*) on interval [-1, 1], where $y = f(x) = \frac{1}{4}(\sin(2\pi x) + \frac{0.01}{x^2 - 0.04})$. After the learning phase, the GMR network gets 82 final neurons and 22 objects neurons. Both in Fig .(a) and Fig.(b), domain factor k_{min}=0.5, merging threshold χ=1. The final neurons are represented by crosses, segments between neurons denote the links. Although Fig.(b) has more redundant links than Fig.(a), the merging results are the same (3 branches are found). The level one neurons (squares) obtained in Fig.(a) (δ=2.5) are the same as the ones in Fig.(b) (δ=6). However, Fig.(b) has more level two neurons (small circles) than Fig.(a), and the links among level 1 and 2 neurons are also different.

6.2.2 Domain Factor

Another important parameter which affects the recalling result is the domain factor k_{min}. The domain state variable for each neuron is initialized to $\rho_{min} = k_{min}\rho_f$ ($k_{min} < 1$). Too small k_{min} might cause the solution missing if the input is outside the training set due to the lack of overlaps between the domains. On the contrary, too large k_{min} might

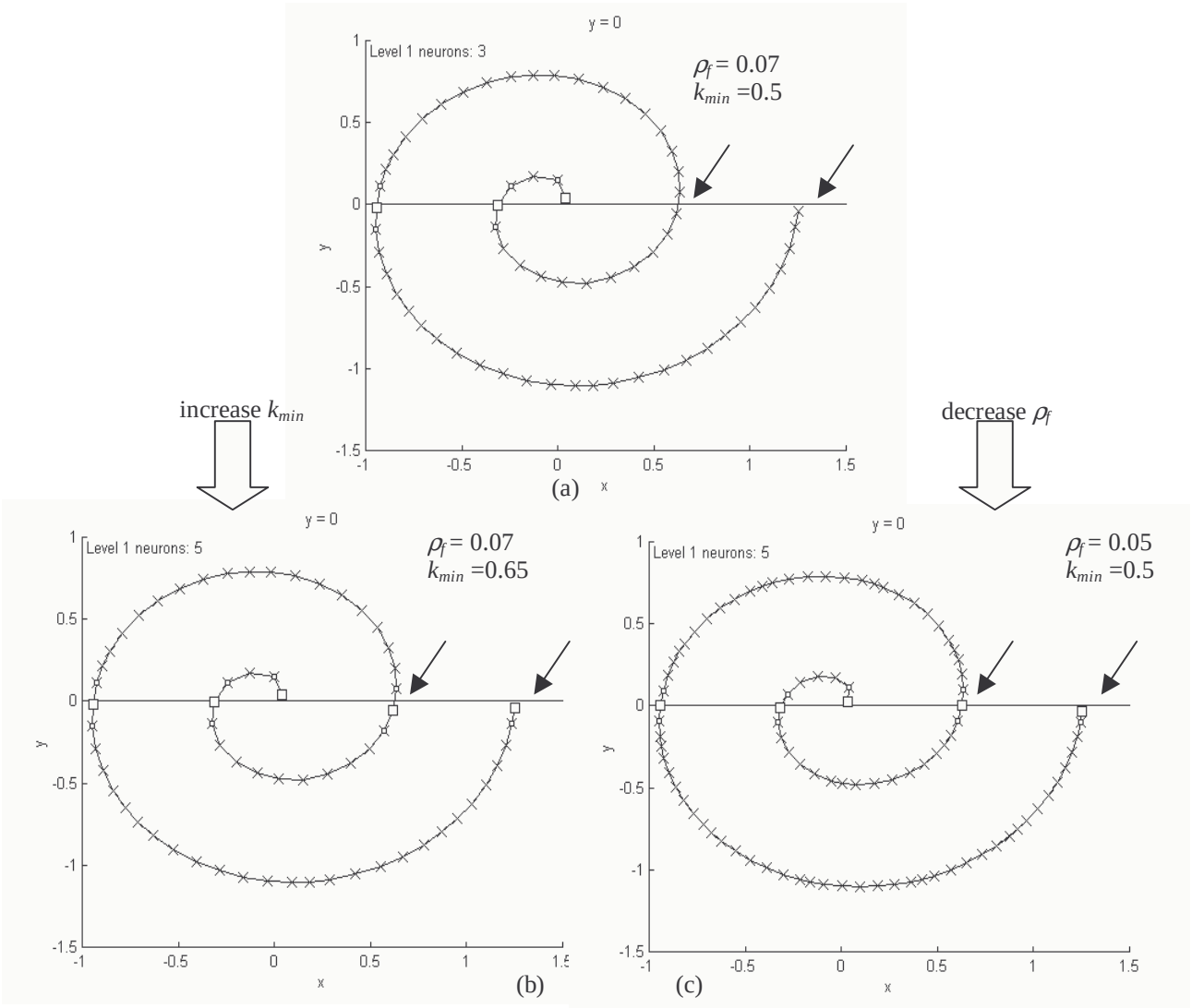


Fig. 6.5 Comparison of recalling results with different settings of mapping accuracy (ρ_f) and domain factor k_{min}. TS is composed of 1250 points on a spiral ρ=θ, $x \in$ [-1, 1.5], $y \in$ [-1.5, 1]. Figs.(a), (b) and (c) share the same linking factor δ=2.5 and the same merging threshold χ=1, and the input for recalling y=0. Crosses denote final neurons, segments denote links, and the level one neurons are marked as squares, level two neurons as circles. Both in Fig.(a) and Fig.(b), ρ_f = 0.07, 63 final neurons are obtained. In Fig.(c) ρ_f=0.05, the final neurons are 93. Fig.(a) shows that two solutions are missed if k_{min}=0.5. As indicated in Fig.(b), by increasing k_{min} to 0.65, the GMR can correctly identify the 5 solutions. Alternatively, keeping k_{min}=0.5, increasing mapping accuracy (decreasing ρ_f) in Fig.(c) can also lead to better results.

cause too many neurons to be activated (to become level one neurons). A compromise should be achieved in order to obtain the generalization ability while still keeping the accuracy of the solution. However, redundant solutions are preferred, as the interpolation after recalling can be used to refine the outputs. Empirically the domain factor ranges from 0.4 to 0.8 depending on the mapping accuracy.

We consider an example which deals with the mapping of the spiral of Archimedes $\rho = a\theta$ (a = 1). As shown in Fig 6.5, a mapping with lower accuracy might need a higher k_{min} in order to always have a solution in each corresponding branch.

6.3 Sparse Region Approximation

The experiments in this section deal with mapping which has sparse regions where the density of the data points is rather low compared with the other parts. The TS consists of 2000 equispaced data points (x, y) with x∈[-1, 1], while value in y coordinate ranges from -62.75 to 62.72.

As shown in Fig 6.6 (a), after the vector quantization phase, the GMR network successfully gets a mapping of the whole data set, and the data points in sparse region are also represented by neurons with low density; furthermore, within the interval of [-1, 1] on *y*-coordinate, the mapping and linking are kept as well as in the previous example. On the other hand, because the links in sparse area are mainly created according to the second-winner criterion, there exist some undesired links. Actually, without prior knowledge about the linking direction, it's not straightforward to decide which to be linked in such a sparse region, even using visulization.

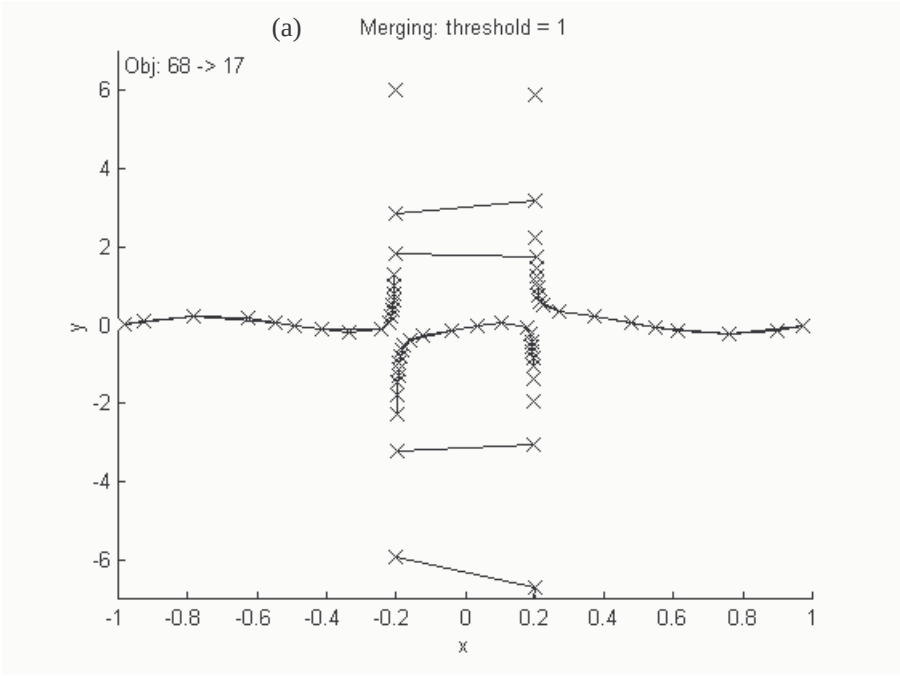
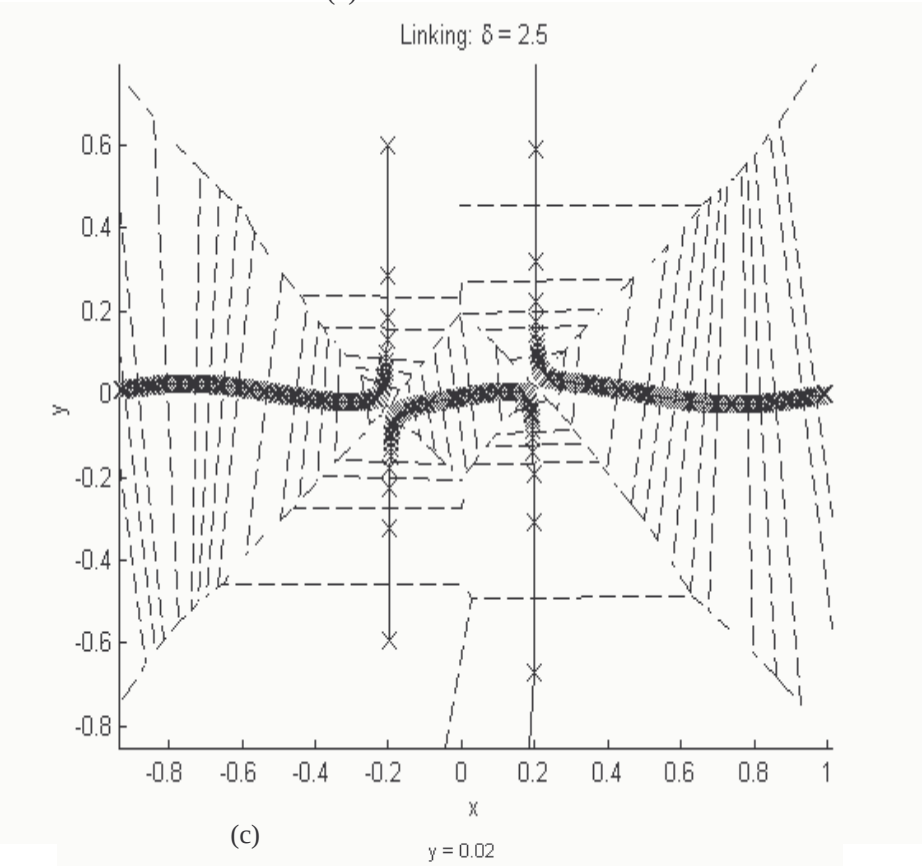


Fig. 6.6 (a) example of GMR mapping on a data set with sparse region without preprocessing the data set.

(b)



(c)

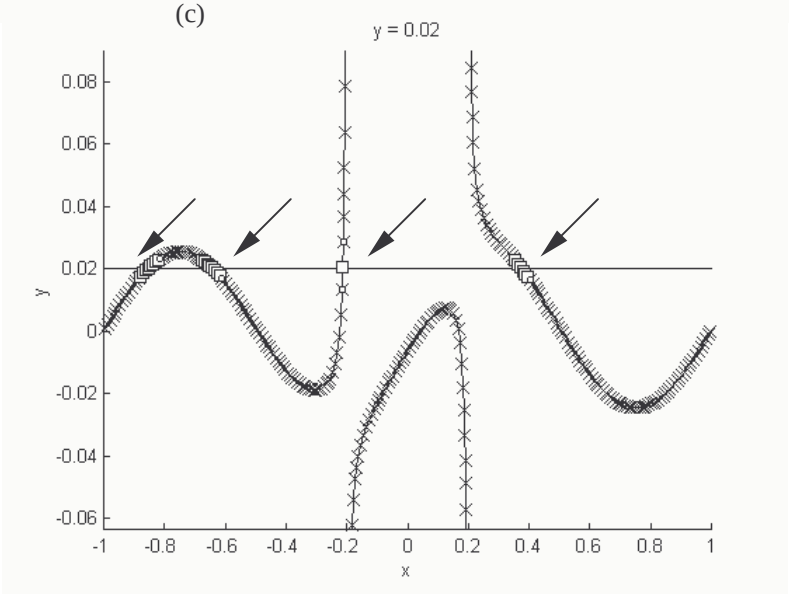


Fig. 6.6 Approximation of a function with sparse region.

TS is composed of 2000 data points (*x*, *y*), where $y = f(x) = \frac{1}{4}(\sin(2\pi x) + \frac{0.01}{x^2 - 0.04})$, $x \in [-1, 1]$.

GMR gets the results in Figs. (b), (c) and (d) by dividing all the *y* values by 10 and increasing the mapping accuracy by smaller vigilance threshold setting $\rho_f = 0.005$. In the mapping of Fig.(a), *y* keeps the original value of the function, the vigilance threshold ρ_f is 0.07 as usual. Crosses denote final neurons, segments denote links, dashed lines (in Fig. (b)) represent Voronoi tessellation after the coarse vector quantization.

Fig 6.6 (b) and (c) indicate that simply by further normalizing the data set, and improving the mapping accuracy, a much better result can be achieved. In this experiment, the data set is preprocessed by dividing all the values in y-coordinate by 10; the vigilance threshold of the level one and level two (in EXIN SNN's) are set to 0.02 and 0.005, respectively. Other parameters are the same as the previous ones: linking factor δ = 2.5, domain factor k_{min} = 0.5, merging threshold χ = 1. Fig. (b) illustrates an accurate linking distribution in the sparse area, Fig. (c) shows that GMR correctly locates 7 solutions belonging to 2 different branches.

6.4 Mapping of Noisy Training Set

The following experiments aim at testing the ability of GMR against noise without using additional regularization techniques. The data set is composed of 4000 equispaced points representing the lemniscate of Bernoulli, i.e.

$$(x^2+y^2)^2 = a^2(x^2-y^2) \qquad (6.4)$$

for a = 1. In order to test the regularity of the mapping, the original data set is permuted and then divided into two subsets. Each subset contains 2000 data points. The noisy training set is generated from the first subset by adding a noise componet ε to each value of y, where ε is a random variable with a uniform distribution centered around 0 and with a standard deviation σ (noise level). The second subset will be used for test. In these experiments, the mapping of two noisy sets with σ equal to 0.1 and 0.15 are carried out under different vigilance threshold. The results of using additional neuron cleaning process and not using this cleaning are also compared. All the learning is done in two levels; ρ_1 is set to 0.2; ρ_2 varies from 0.035 to 0.08. Here the cleaning process is done in each single EXIN SNN in the second level. Just like the above experiments, *SSD* is used to measure the mapping result. In Table 6.5, column 5 reports the *SSD* for the noisy training set; column 6 reports the *SSD* to the test set. The number of neurons in level one and level two are recorded in column 4.

Table 6.5 Mapping from data set with different noise level.

noise level σ	vigilance threshold ρ_2	cleaning or not?	# neurons	*SSD* to TS (subset1 + noise)	*SSD* to test set (subset2)
0,1	0,035	N	116	0,4458	0,4130
	0,035	Y	106	0,5078	0,4836
	0,04	N	98	0,5847	0,5478
	0,04	Y	91	0,6534	0,6237
0,15	0,06	N	91	1,2903	0,9487
	0,06	Y	67	1,7520	1,2435
	0,08	N	61	2,2090	1,4917
	0,08	Y	44	2,9834	2,5250

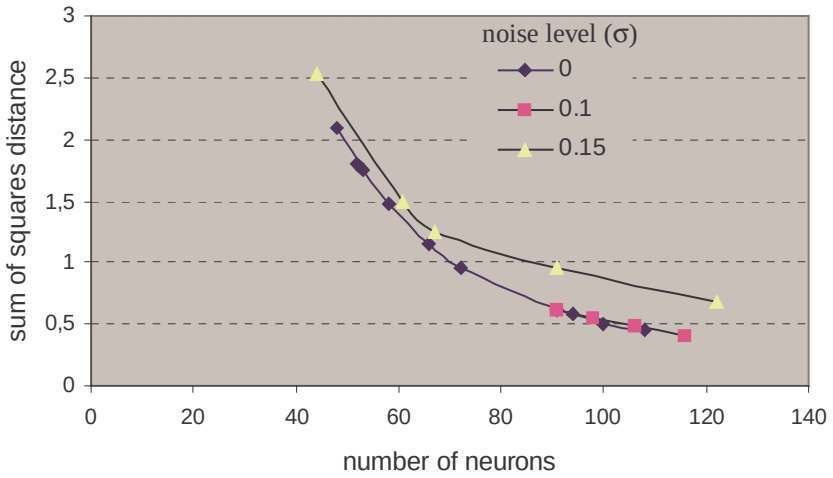


Fig. 6.7 Comparison of mapping (*SSD*) over data set with different noise level (σ=0, 0.1, 0.15).

Fig 6.7 is the plot of the results in Table 6.5 compared with the mapping result under noise free TS. When σ = 0.1, the mapping result is very close to the result learned from noise free TS. When σ = 0.15, the mapping result becomes very similar to the noise free test when the number of neurons is small enough (< 70). It reveals that GMR is robust to noise at the cost of lower mapping accuracy. This is the same as many other neural networks. The generalization with simple structures is better than the complex ones. Fig 6.8 illustrates the results of mapping on the noisy training set (σ = 0.15) by GMR, where ρ_2 = 0.08, no cleaning is used and the linking factor δ is 3. The grey circles in Fig. (a) denote the training set, the crosses denote the final

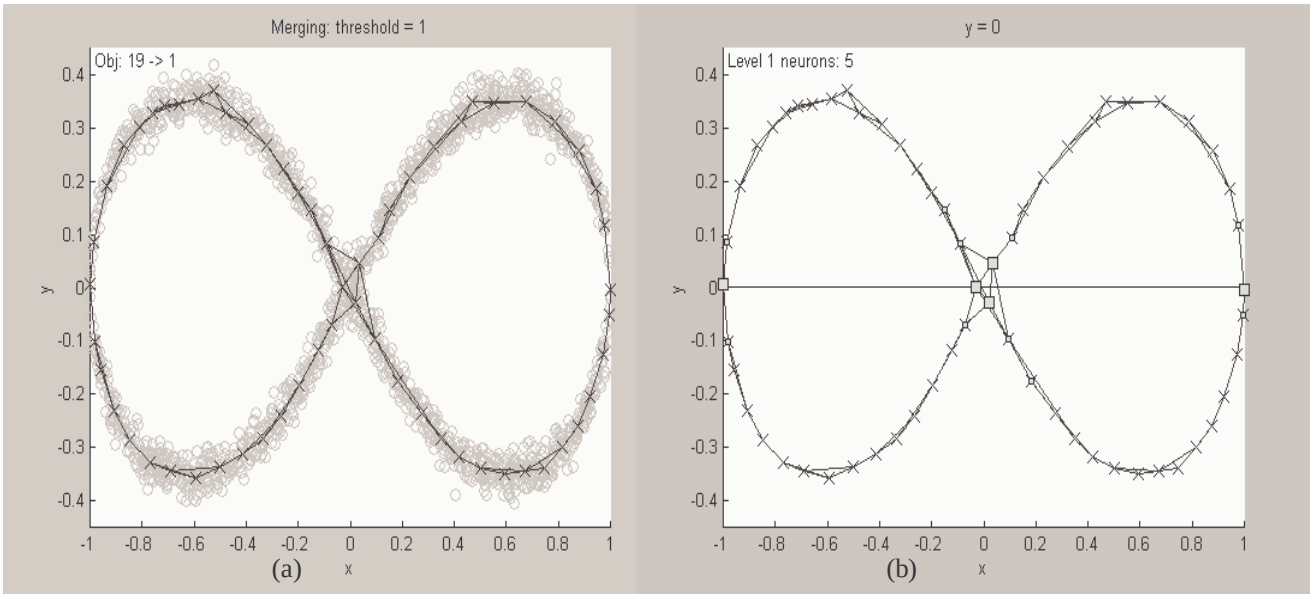


Fig. 6.8 GMR gives the mapping from a noisy TS (σ=0.15).
(a) TS vs. GMR final neurons and links.
(b) It shows that GMR can also provide correct solutions for a certain *y* value.

neurons, the links are represented by segments. Fig. (b) shows that GMR indentifies 5 level one neurons (squares) and a few level 2 neurons (small circles); the more accurate result can be obtained by further interpolation.

Experimental results also show that both neuron cleaning and lower vigilance threshold can lead to less final neuron and higher regularity, as can be seen from Table 6.5.

Moreover, as claimed previously, GMR is an open achitecture; it can provide a pool of neurons for modelling within which different neural architechtures can be implemented. Hence its generalization ability may be improved by using other techniques. For example, robust Total Least Square (TLS) techniques [1] might be used in the learning weight adaptation phase for rejecting noise.

6.5 Other Simulations for Mapping Approximation

In general, the object merging threshold χ is set to 1, all data sets are normalized in order to have range [-1,1] for every coordinate.

The TS of the first example is composed of 5000 equispaced points extracted from:

$$y = \frac{1}{100}\left(\frac{(4x)^4 e^{-4x}}{(4x-3)^2(4x+1)(4x+2)}\right) \qquad (6.5)$$

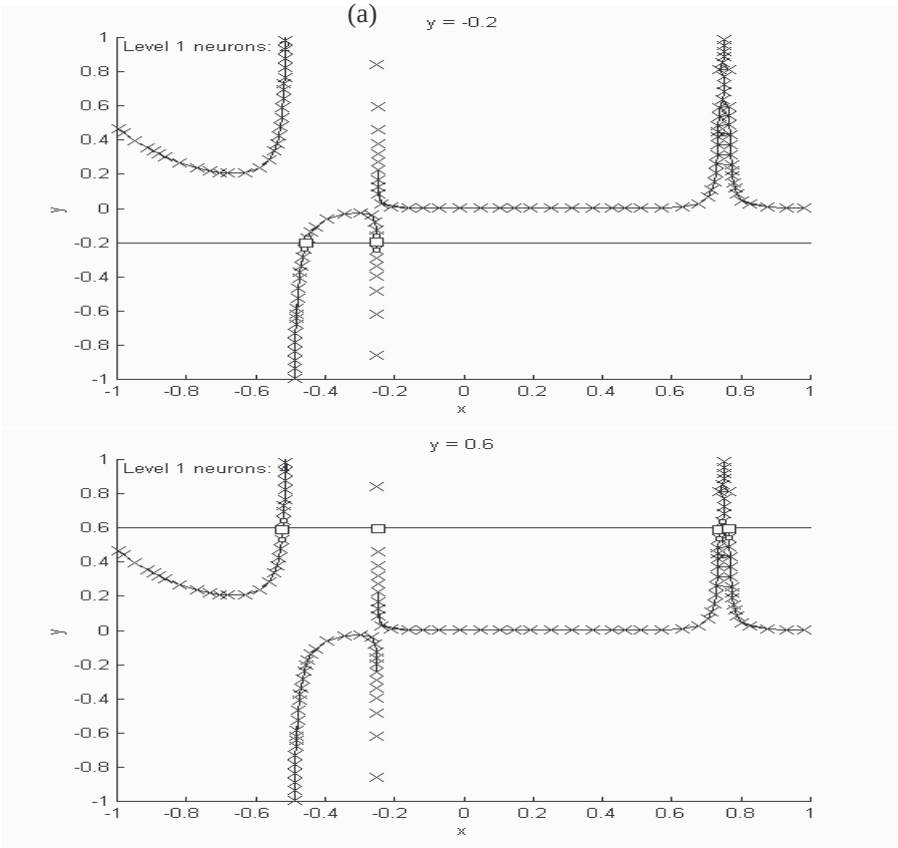


Fig. 6.9 Example:
(a) GMR finds 2 solutions belonging to a unique branch.
(b) GMR finds 4 solutions belonging to 2 distinct branches.

Fig 6.9 shows the results (square neurons) for certain values of *y*, superimposed to the linking representation. As revealed in Fig.(b) GMR correctly finds 4 solutions but considers the two solutions on the right as belonging to the same branch due to the low resolution.

The following example deal with the mapping of the curve of Lissajous

$$x = \cos(at),\ y = \sin(bt) \qquad (6.6)$$
$$(a = 3,\ b = 5)$$

The training set consists of 5000 equispaced points on the interval of [-1, 1] without noise. The parameter for training phase are the following:

- 3 level learning: $\rho_1 = 0.5$, $\rho_2 = 0.15$, $\rho_3 = 0.025$; 2 epochs for each EXIN SNN.
- Creation of the linking matrix and setup of the domains: $\rho_{min} = 0.65\rho_3$, $\delta = 2.3$.
- Object merging: $\chi = 1$.

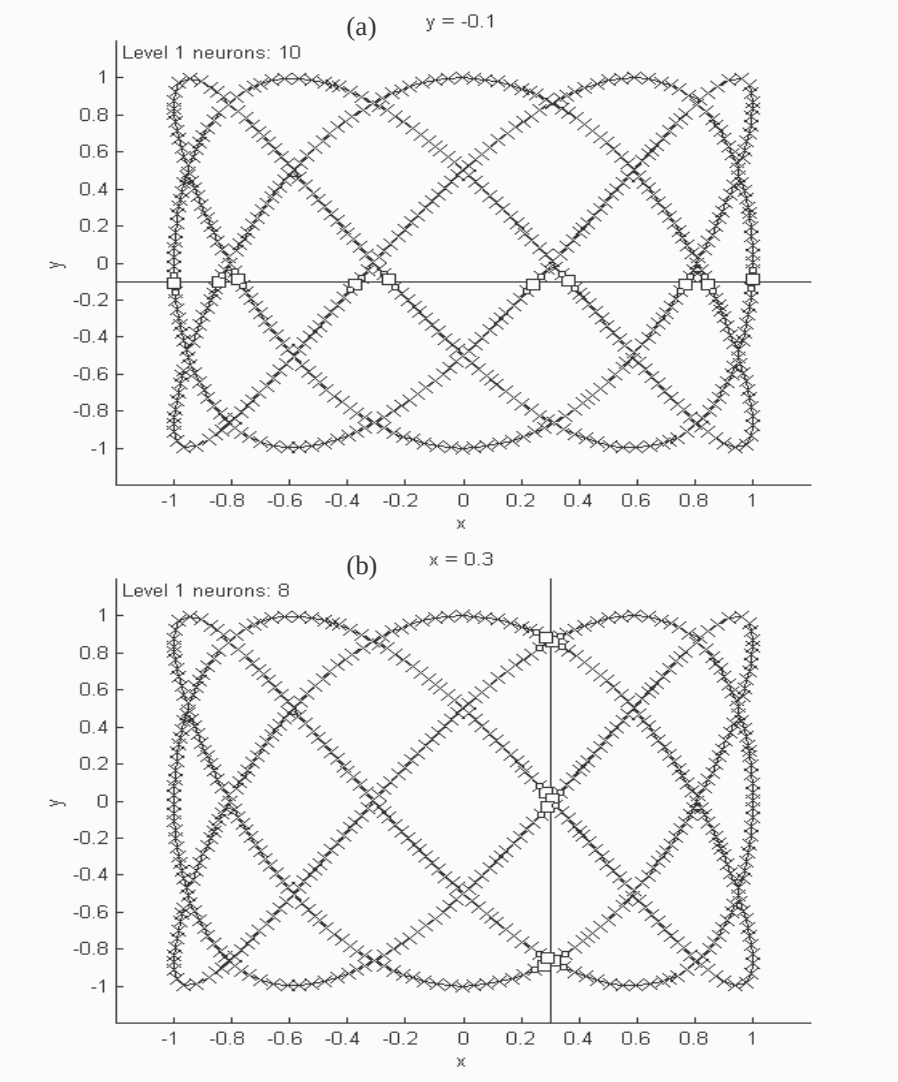


Fig. 6.10
Example of GMR approximation.

(a) GMR finds out the exact solutions for y = 0.2.
(b) GMR identifies 8 level one neurons for x=0.3.

The number of neurons generated from 1 to 3 levels is 9 – 75 – 550, respectively. In the end, 75 objects are merged into one unique branch. Fig 6.10 (a) shows that the GMR is capable of correctly locating the solutions. From this information, by using an interpolation technique, accurate solutions can be found.

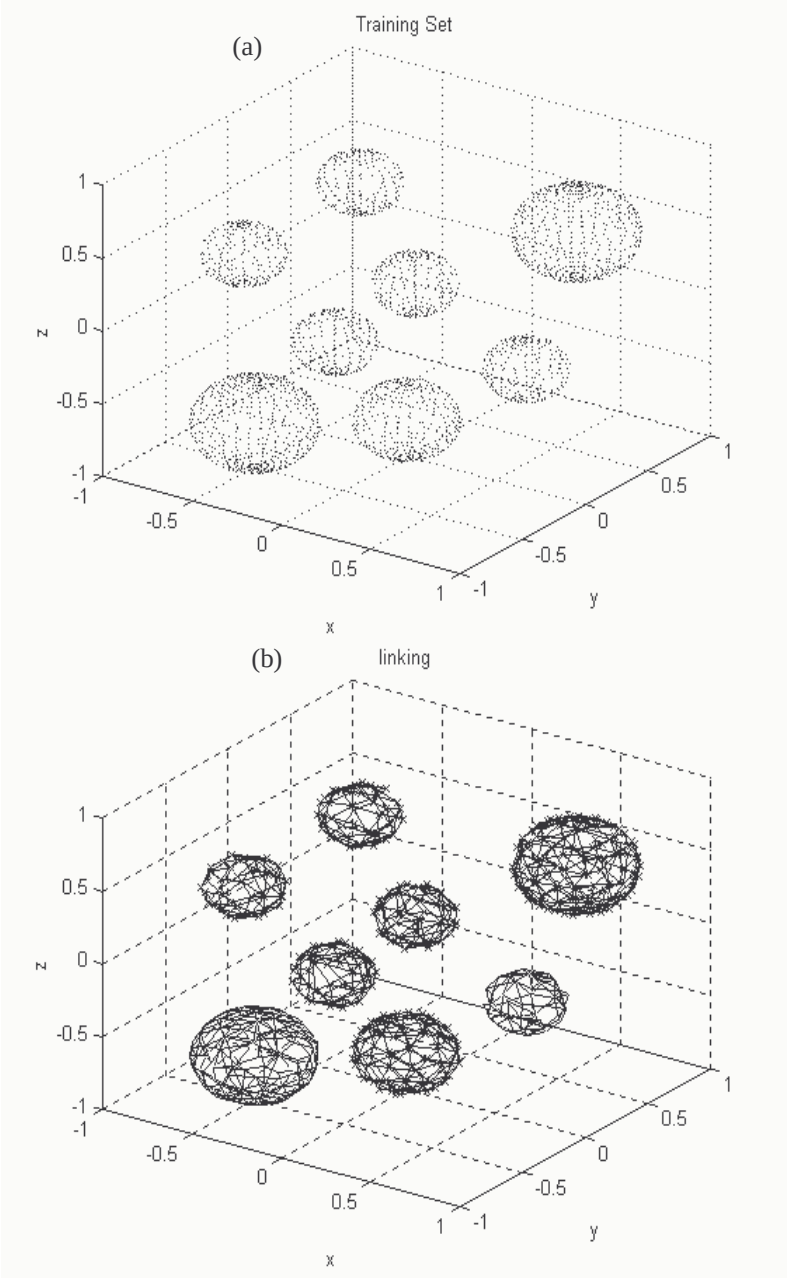
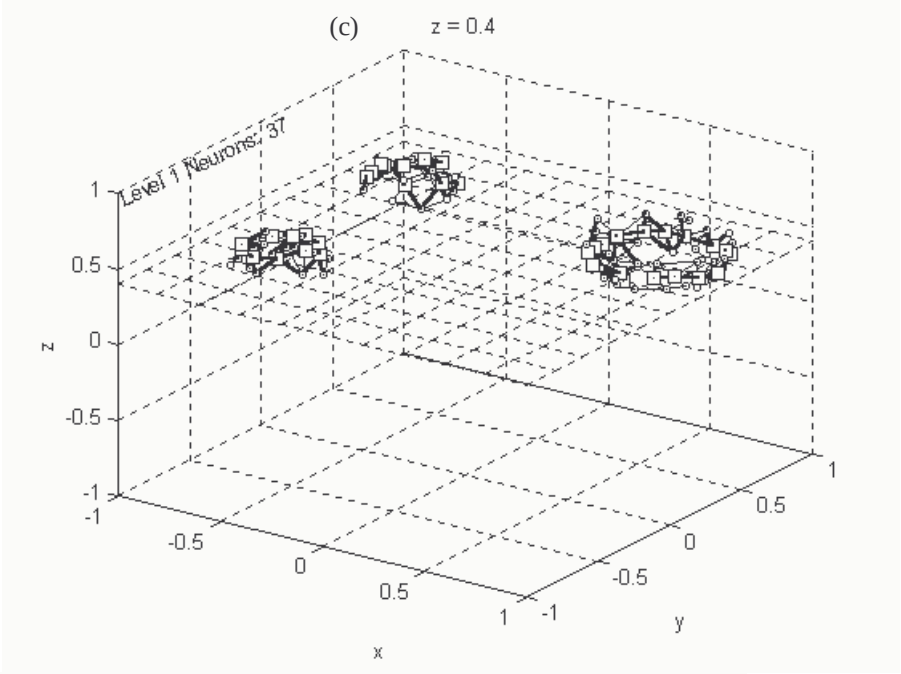
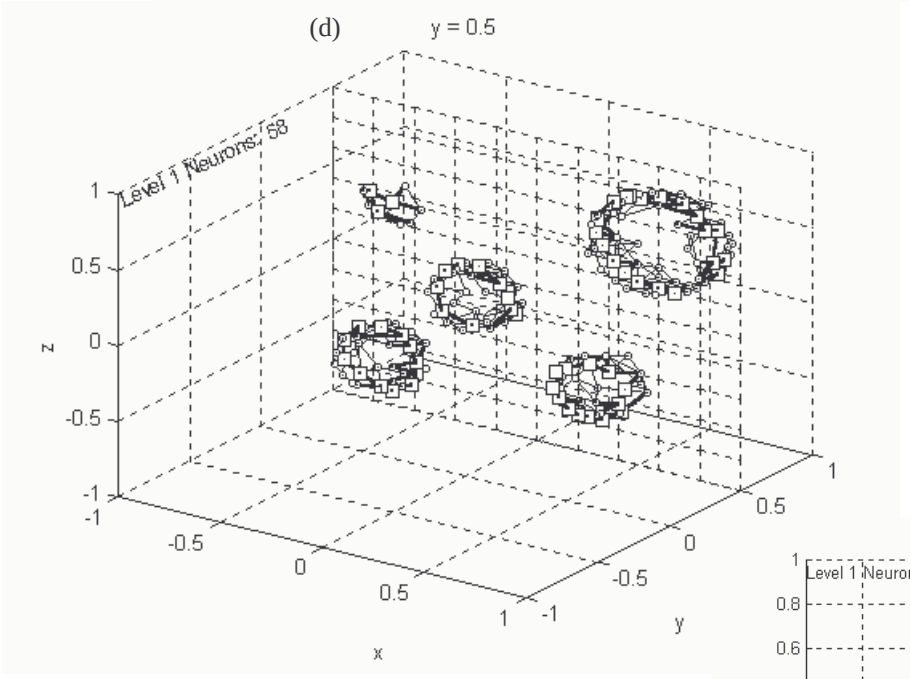


Fig. 6.11 GMR mapping of 8 sphere in a 3-D scene.
(a) training set (points),
(b) the neuron weights (crosses) and links (segments)





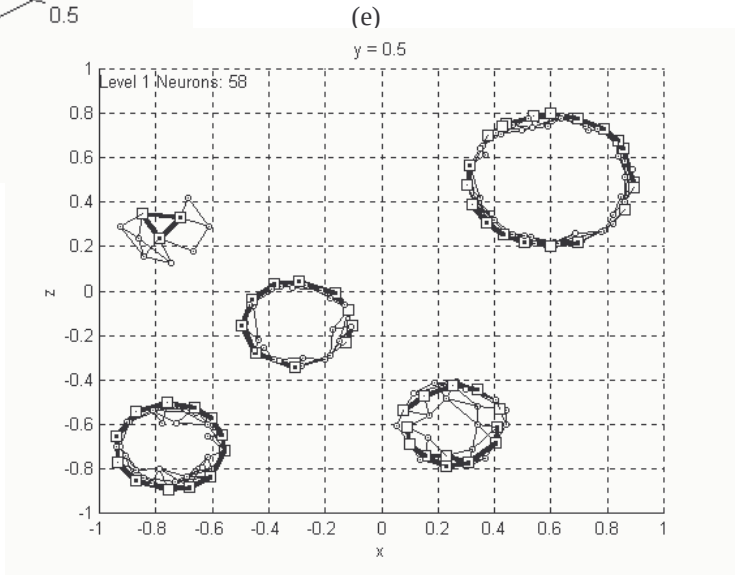


Fig. 6.11 GMR mapping of 8 sphere in a 3-D scene.
Fig.(c), (d) and (e): for a certain value of *z* and *y* as input respectively, show the activated (level one) neurons (squares), and level 2 neurons (circles), the links among level one neurons (bold segments) yield the contours of the object. The links connected to the level two neurons are also shown as segments. Fig.(e) is a projection of Fig.(d) in 2 dimentions for clearer view of the results for input $y = 0.5$.

The last example deals with the same mapping problem as in section 6.1.3, on a three dimensional scene composed of 8 spheres in different positions. This TS is made up of 3000 points randomly extracted from the spheres. Fig 6.11 (a) shows the distribution of the training data. After the merging phase 8 objects are correctly identified. This emerges in Fig.6.11 (b) which shows the neuron weights and the linking representation. Figs.(c), (d) and (e) show the level one neurons together with the associated links between the level one and level two neurons for different values of *z* and *y*, respectively. Fig.(e) is a projection of Fig.(d) from 3 dimensional space into 2 dimentional space for a clearer view of the contours (links among level one neurons).

6.6 Summary

As a conclusion, it follows a list of the main results obtained in our experiments.

- The parameter setting for GMR training has been studied for the case of application of EXIN SNN.
  - Two criteria for stopping the EXIN SNN training are compared. From the computational efficiency point of view, the fixed epoch criterion is comparable with the ∇-stop criterion (stop when the weights increment is small enough).
  - A fixed but smaller vigilance threshold ρ is capable to achieve the desired mapping accuracy earlier than using a decaying ρ.
  - Multilevel learning is superior to single level learning both in the computational efficiency and mapping accuracy.
  - Practically, there exist choices for a range of values of linking factor, merging threshold and domain factor.

- GMR is capable of doing mapping on a data set with sparse regions, while respecting the discontinuities, provided the mapping resolution is high enough.

- Without considering other interpolation techniques, GMR is robust to the noise for a lower mapping resolution. For the training phase, regularization can be achieved by a higher vigilance threshold, neuron cleaning and linking cleaning. Furthermore, other smoothing techniques can also be used after the recalling phase in order to refine the results.

- The above simulations suggest that GMR has generalised mapping approxiamtion capability, particularly performs well in solving the inverse and discontinuous problem. It has exhibited its powerful abilities in high level modelling. An application of GMR has been devised by G.Cirrincione to solve the scene reconstruction problem in computer vision [1]. By combining other neural techinques in this open architecture, more applications can be expected.

Chapter 7

Conclusions

We have studied and implemented the GMR algorithm. The acceleration of its computation is then accomplished by applying the branch and bound search technique. The features of GMR are further discussed based on a series of experiments and simulations. A summary of the thesis now follows:

General Features of GMR

- GMR is an incremental self-organizing competitive neural network with adaptive linking among neurons. By transformation the function approximation problem into a pattern recognition problem, GMR is able to:
  - approximate every kind of mapping (function or relation) in both senses, i.e. $M(x,y) : x \in \Re^m \leftrightarrow y \in \Re^n$.
  - yield all the solutions and the corresponding branches of the mapping and, if the case, the equilevel surfaces, for a given input. Its input could be every possible collection of components of x and y and its output is the estimation of the remaining components.

- It is an open architecture, under which many different neural techniques can be chosen to improve its performance.

Computation Efficiency and Mapping Accuracy

- The training of GMR can be easily speeded up by applying some tree search techniques. Branch and bound search technique has been proved to be very powerful when it's used to facilitate k-nearest neighbours computation.
  - In case of GMR which uses coarse-to-fine strategy in vector quantization, the hierarchical information of neurons can be easily employed to create a tree structure without extra computation load. Even if we have to use extra clustering procedures to create a tree, e.g. before we apply branch and bound

in relabelling, the extra computation effort is still worthy compared with the time we can save later.

- The efficiency of the branch and bound is proportional to the size and dimension of the data set and is related to the number of the nodes (neurons) in the searching tree structure. In our experiments, the best result is the following: more than 70% of the flops of training can be saved when dealing with a 3-D data set which consists of 3000 data points.
- Experimental results also show that a tree created by using multilevel EXIN SNN's can achieve higher computational efficiency for branch and bound searching, compared with the tree created by using *k*-means clustering. Also EXIN SNN outperforms *k*-means in the speed in doing clustering.

- Many parameter settings involve a compromise between computational efficiency and mapping accuracy. In case of EXIN SNN learning, several empirical settings can be chosen, e.g. fixed epoch stopping criterion, fixed vigilance threshold in each level. The computation time is sensitive to the number of levels, vigilance threshold of each level and epoch setting. The linking and domain factor will also influence the accuracy of the outputs. An empirical range for the parameter setting is available.

- Alternative methods of parameter setting could further improve the GMR performance. A technique which adjusts those experimental values based on the changes in the approximation error over a series of quantization and approximation runs might lead to better choices for those parameters.

Future Work

GMR is indeed a strategy which exploits neural features. It has some fundamental ingredients which are the basic prototypes of the neural networks (neural modules) and rules according to which these modules interchange their information [2]. It will be left for the future work to study the variants of its ingredients and try new application domains:

- Certainly, the GMR performance must be improved by using interpolation techniques. Either linear or nonlinear techniques can be applied to improve its smoothness.

- The weight adaptation techniques in the learning phase can be further analyzed. For example, robust Total Least Square (TLS) techniques might be used here for rejecting noise [1].

- Other linking techniques can be tried both for linking creation and pruning.

- Further analysis of the limits in the determination of the branches of the solutions, analysis of the curse of dimensionality, and theorems of convergence, could also be the subject for future research.

- Also interesting is to make comparison with the other existing neural approaches (e.g. mixture-of-experts).

In the end, GMR provides a powerful framework for generalised mapping approximation. It can be used to solve inverse and discontinuous problems. More applications of GMR could be expected in the future.

References

[1] G. Cirrincione. *A Neural Approach to the structure from Motion Problem*. PhD thesis, LIS INPG Grenoble, December 1998.

[2] G. Cirrincione, M. Cirrincione, Sabine Van Huffel. *Mapping Approximation by the GMR Neural Network*, 1999.

[3] G. Cirrincione. *Neural Networks for Inverse Discontinuous Problems.* Presentation slides, ESAT, Kuleuven, 2000.

[4] Christopher M. Bishop. Neural Networks for Pattern Recognition. *Oxford University* Press, 1995.

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

[6] R. Hecht-Nielsen. Neurocomputing. *Addison Wesley*, 1990.

[7] D.A. Sprecher. A universal mapping for komogorov's superposition theorem. *Neural Networks*, 6(8): 1089-1094, 1993.

[8] E.D. Sontag. Feedback stabilization using two-hidden-layer neets. *IEEE Trans. on Neural Networks*, 3:981-990, 1992.

[9] J. Platt. A resource allocating network for function interpolation. *Neural Computation*, 3:213-225, 1991.

[10] L. Yingwei, N. Sundararajan, and P. Saratchandran. A sequential learning scheme for function approximation using minimal radial basis function neural networks. *Neural Computation*, 9(2):461-478, February 1997.

[11] R.A. Jacobs, M.I. Jordan, S.J. Nowlan, and G.E. Hinton. Adaptive mixtures of local experts. *Neural Computaion*, 3:79-87, 1991.

[12] M.I. Jordan and R.I. Jacobs. Hierarchical mixtures-of-experts and the EM algorithm. *Neural Computation*, 6:181-214, 1994.

[13] Peng, F., Jacobs, R. A., and Tanner, M. A. Bayesian inference in mixtures-of-experts and hierarchical mixtures-of-experts models with and application to speech recognition. *J. Ameri Statist. Assoc.* 91: 953-960, 1996.

[14] B. Fritzke. A Growing Neural Gas Network learns topologies. Advances in Neural Information Processing Systems 7, *MIT Press*, Cambridge MA, 1995.

[15] B. Fritzke. Growing Grid – a self-organization network with constant neighborhood range and adaptation strength. *Neural Processing Letters*, Vol. 2, No. 5, pp. 9-13,1995.

[16] Jacek M. Zurada, Introduction to Artificial Neural Systems, *West Publishing Company*, 1994.

[17] Patrick Henry Winston, Artificial Intelligence, *Addison Wesley*, 1993.

[18] Keinosuke Fukunaga and Patrenahalli M. Narendra. A Branch and Bound Algorithm for Computing *k*-Nearest Neighbors, *IEEE Transactions on Coumputers* : 750 – 753, July 1975.