Abstract

Evaluation of Heterogeneous Architectures for Artificial Neural Networks

by

Chedsada Chinrungrueng

Doctor of Philosophy in Engineering – Electrical Engineering
and Computer Science

UNIVERSITY of CALIFORNIA at BERKELEY

Professor Carlo H. Séquin, Chair

Existing *monolithic* artificial neural network architectures are not sufficient to cope with large complex problems. A better approach is to build large scale *heterogeneous* networks using both supervised and unsupervised learning modules. In these architectures an unsupervised learning algorithm, such as the *k-means algorithm*, decomposes the overall task and a supervised learning algorithm, such as one based on *gradient descent*, solves each subtask.

We have investigated heterogeneous architectures that are based on a novel k-means partitioning algorithm that integrates into its partitioning process information about the input distribution as well as the structures of the goal and network functions. We have also added two new mechanisms to our k-means algorithm. The first mechanism biases the partitioning process toward an optimal distribution of the approximation errors in the various sub-domains. This leads to a consistently lower overall approximation error. The second mechanism adjusts the learning rate dynamically to match the instantaneous characteristics of a problem; the learning rate is large at first, allowing rapid convergence, and then decreases in magnitude as the adaptation converges. This results in a

lower residual error and makes the new k-means algorithm also viable for non-stationary situations.

We evaluate the *performance* and *complexity* of these heterogeneous architectures and compare them to homogeneous radial basis function architectures and to multilayer perceptrons trained by the error back-propagation algorithm. The evaluation shows that the heterogeneous architectures give higher performance with lower system complexity when solving the Mackey-Glass time series prediction probelm and a hand-written capital letter recognition task.

Contents

List of Figures

List of Tables

Acknowledgements

I am deeply grateful to my advisor, Prof. Carlo H. Séquin, for his continuous encouragement and invaluable guidance throughout the course of my graduate study. I have learned a great deal from his teaching, knowledge, and criticism. I would like to thank Prof. Jerome A. Feldman and Prof. Stuart E. Dreyfus for serving on my dissertation committee. Thank also to Prof. Ping Ko and Prof. Christian H. Hesse for participating on my qualifying examination committee.

Next I would like to thank every one who contributed to my reserach work one way or the other. I wish to thank Reed Clay, Mani Srivastava and Vijay Medisetti for many useful discussions. It has been great pleasure to share the office with Glenn Adams, Ping-San Tzeng, and Dan Rice. Their friendship and support are truly appreciated. Special thanks should also go to Yumiko Nakanishi for her patience and encouragement. Prof. Mongkol Dejnakarintra, Prof. Suriyan Tishyadhigama, and Prof. Ekachai Leelarasmee, represent the many wonderful professors who taugh me so much in my undergraduate years at Chulalongkorn University, Thailand.

Finally I would like to thank my father, Chongnum, and mother, Wanna, for their love, encouragement, and patience, without which I would not be where I am today.

The financial support provided by the Joint Services Electronics Program and by the Thailand-United States Educational Foundation is gratefully acknowledged.

Chapter 1

Introduction

1.1 Motivation and Goal

In the context of empirical inference of multi-variate functions, an artificial neural network is essentially a function represented by the composition of many simple functions, referred to in this dissertation as *basic functions*[1]. These basic functions are usually parameterized but constrained in form: typically non-linear functions of a few variables or linear functions of many variables [1]. By adjusting the parameters of these basic functions, we can alter the shapes of these functions and thus modify the overall input-output relationship of the network. Several forms of basic functions have been proposed; however, they can be divided based on the characteristics of their supports into those with *global* supports, as exemplified by the *sigmoid* function; and those with *local* supports, as exemplified by the *Gaussian radial basis* function.

Since we are interested in artificial neural network architectures which are simple to

[1]We refer to simple functions composing an artificial neural network as *basic* functions instead of *basis* functions in order to emphasize the fact that the network can have a more elaborate form of composition than a simple weighted sum of basis functions.

Figure 1.1: Examples of basic functions used in artificial neural networks.

implement with dedicated VLSI hardware, we will focus in this dissertation on architectures that have fixed replicated structures and use on-line learning algorithms. Particularly, we are interested in architectures whose learning algorithms can be expressed as simple recursive equations and do not use complicated data structures. Most of these architectures are homogeneous in the aspect that they are composed of the same types of basic functions. The architectures that are composed of *global-support basic functions*, such as, the *multi-layer perceptron* [2], tend to form very compact representations but require a long training time. Conversely, the architectures that are composed of *local-support basic functions*, such as, a *radial basis function network* [3, 4, 5, 6, 7] tend to learn rapidly, but requires more extensive hardware, i.e., a large number of processing units.

Traditional artificial neural network architectures solve their tasks by addressing the entire problem as a whole. However, this approach is not sufficient to cope with large, complex problems. Simply extending these architectures to ever larger *homogeneous* systems in order to solve larger problems is impractical. Training large-scale networks composed of global-support basic functions as *monolithic* systems is unacceptably slow because a large number of parameters have to be adjusted concurrently. Experimental studies indicate that the learning times of these monolithic networks scales poorly with the sizes of problems [8]. For example, Tesauro and Janssens [9] experimentally showed that the learning time of the multilayer perceptron trained with *error back-propagation* [2]

for the XOR problem grows exponentially with the complexity of the problem. Since there is no efficient way to bias the structures of these networks, the large amount of training data is needed [10]. On the other hand, building a large scale network composed of local-support basic functions in parallel VLSI hardware will typically be too costly because the structure of the network would have to be very large and because its connectivity requirements would be excessive. According to Akers and Walker [11], the connectivity of a network with just a few thousand processing elements would exceed the current or even projected interconnection density of ULSI system.

The existence of *heterogeneous* organizations in mammalian visual systems [12, 13, 14] suggests that artificial neural networks for solving large, complex problems should be composed of variety of modules, each dedicated to a different sub-task. Several heterogeneous architectures based on task decomposition [15, 16, 17, 18, 19, 20, 21, 22, 23] have been proposed for solving large and complex supervised learning problems. One embodiment of such architectures comprises a *gating module* that divides the assigned task into subtasks and a collection of specialized *expert modules*, each of which is assigned to solve a particular subtask.

Jacobs [22] has proposed a class of heterogeneous architectures that are based on an *associative Gaussian mixture model*. This model assumes that the error difference between the target output and the output of an expert module has a Gaussian distribution. The output of the architecture is a linear combination of the outputs of all the expert modules, weighted by the corresponding outputs from the gating module. The learning goal of this architecture is to maximize the negative log likelihood of generating the desired output under this model. This study shows that these architectures can achieve better accuracy than a single multi-layer perceptron trained by error back-propagation. However, their convergence rate, similiar to that of a single multi-layer

perceptron trained by error back-propagation, is still too slow for practical purposes, since all the parameters of a system in these architectures are adjusted concurrently, which leads to undesirable coupling that slows down the convergence rate.

Several other researchers [15, 17, 19, 21] have proposed another class of heterogeneous architectures that are based on k-means partitioning. In these architectures, the *k-means algorithm* is used by a gating network to partition the domain of an assigned task into non-overlapping sub-domains. The task defined on each sub-domain is then solved by an expert module trained by a supervised learning algorithm based on *gradient descent*. The class of heterogeneous architectures based on k-means partitioning has been shown to have a higher accuracy and faster convergence rate than a single multi-layer perceptron trained by error back-propagation, i.e., for approximating a 2-dimensional sinc function [21] and for recognizing Japanese characters [23]. In addition, systems of these heterogeneous architectures have also been shown to use less hardware than radial basis function networks, i.e, in time series prediction [17]. Because of their advantages in speed and hardware over the traditional architectures, we examine in this dissertation the on-line version of a class of heterogeneous architectures that are based on the k-means partitioning. Our investigation will concentrate on both the performance and algorithmic complexity of these architectures when addressing large, complex problems. We believe that this investigation provides a guide for the construction of large general purpose artificial neural networks.

1.2 Thesis Overview

Following this introduction, the background material for the dissertation and the mathematical notation used in the thesis are provided in chapter 2. The k-means clustering algorithm,

the supervised learning algorithms based on gradient descent, and the heterogeneous architectures based on k-means partitioning are briefly reviewed.

The performance of the heterogeneous architectures based on k-means partitioning strongly depends on the efficacy of the k-means algorithm in decomposing the assigned task. Thus in chapter 3 we investigate the k-means algorithm in general and the adaptive k-means algorithm in particular. We introduce two novel mechanisms for improving the performance of the k-means algorithm. The first mechanism is for biasing the partitioning process so that it can achieve an optimal partition. The second mechanism is for adjusting the learning rate dynamically to match the instantaneous characteristics of a problem, permitting the algorithm to converge first very rapidly and later very closely towards an optimal solution. The dynamic adjustment of the learning rate also renders the algorithm usable for non-stationary situations.

In chapter 4 we introduce an enhancement for the class of heterogeneous architectures based on k-means partitioning. The enhanced architectures are characterized by a novel k-means algorithm that not only considers the input distribution but also integrates into its partitioning process information about the goal function and the capabilities of the expert modules. The new k-means algorithm allows each individual region in the partition to adjust its size so that the representation resources in all the regions are optimally used. In order to enable the proposed k-means algorithm to achieve its optimal performance and to be usable for both stationary and non-stationary situations, we have also included the two mechanisms introduced in chapter 3.

Chapter 5, 6 and 7 present the *performance* and *complexity* evaluation of the enhanced heterogeneous architectures. In chapter 5 the performance of these enhanced heterogeneous architectures is evaluated compared against that of the radial basis function architectures [6], and against

multilayer perceptrons trained by the error back-propagation algorithm [2] using the Mackey-Glass time series prediction benchmark and a hand-written capital letter recognition task. For the Mackey-Glass problem, where the input dimension is quite low, we also compare the enhanced heterogeneous architectures with the architectures based on the lookup table and on the local model approaches [24, 25].

In chapter 6, we investigate the complexity of serial implementations of the enhanced heterogeneous architecture compared to those of the radial basis function and back-propagation architectures. We analyze the implementation of each architecture to determine the number of arithmetic operations in its training cycle, and also the time needed to perform these operations serially. In chapter 7, we examine the complexity of parallel implementations of the aforementioned three architectures. We determine for each architecture the *implementation cost*, defined as the silicon area required by the arithmetic blocks in the implementations. We also compute the time needed by each architecture to complete its training cycle assuming maximum parallel execution. Finally, we summarize the results of this study and recommend directions for future reserach in chapter 8.

Chapter 2

Heterogeneous Architecture Based on K-Means Partitioning

The heterogeneous architectures [15, 17, 19, 21, 23] that we investigate in this dissertation are composed of a *gating module* that uses the *k-means algorithm* to partition an assigned task, and a collection of specialized *expert modules* that are trained by a supervised learning algorithm based on *gradient descent*. In this chapter, we will first review the k-means algorithm in section 2.1, and the supervised learning algorithms based on gradient descent in section 2.2. We will then cover in section 2.3 the mathematical definition of the heterogeneous architectures based on the k-means partitioning.

2.1 K-Means Clustering Algorithm

The k-means clustering algorithm [26, 27, 28, 29] has been applied in many areas of applications. In the area of communications, it has been used for compressing image or speech

data. In the area of connectionist network modeling, it has been used for processing the input data of complicated classification tasks, e.g., in feature-map classifiers [30] or in radial basis function networks [29]. In this dissertation, we are interested in applying the k-means algorithm to decompose the given task for heterogeneous architectures.

The task of the k-means algorithm is to partition the domain $\mathcal{I}$ of input pattern $\vec{x}$ into K regions. When the Euclidean distance is used as a deviation measure between $\vec{x}$ and reference vector $\vec{c}_k$, the k-means clustering problem can be formulated as that of finding a partition [$\mathcal{I}_1$, ... , $\mathcal{I}_K$] and reference vectors $\vec{c}_1$, ... , $\vec{c}_K$ that minimize the *total spatial variation*:

$$TSV = \sum_{k=1}^{K} v_k \qquad \text{with} \qquad v_k = \int_{\mathcal{I}_k} p(\vec{x}) \|\vec{x} - \vec{c}_k\|^2 \, d\vec{x}, \tag{2.1}$$

where p denotes the probability distribution of $\vec{x}$, and the notation $\| \|$ denotes the l_2 norm. Quantity v_k is the *spatial variation* in region $\mathcal{I}_k$, and is defined by the expected value of the squared Euclidean distance between $\vec{c}_k$ and $\vec{x}$ in $\mathcal{I}_k$. The value of v_k thus depends on the location of $\vec{c}_k$ and on the geometrical properties of the region $\mathcal{I}_k$.

One common way for defining region $\mathcal{I}_k$ in the partition is to use a *membership indicator*. The membership indicator M_k specifies whether a given point $\vec{x}$ in $\mathcal{I}$ belongs to region $\mathcal{I}_k$. We define its value to be 1 if $\vec{x}$ is in $\mathcal{I}_k$ and 0 otherwise. For the *traditional* k-means algorithm, which is based on the Euclidean deviation measure, the membership indicator M_k is given by

$$M_k(\vec{x}) = \begin{cases} 1 \quad \text{if} \quad \|\vec{x} - \vec{c}_k\|^2 \;\leq\; \|\vec{x} - \vec{c}_i\|^2 \quad \text{for each } i \neq k \\ 0 \quad \text{otherwise} \end{cases} . \tag{2.2}$$

For the event of more than one minimum $\|\vec{x} - \vec{c}_k\|$, we set M_k with the lowest index k to 1 and the

others to 0.

In the *batch mode* of operation, the k-means algorithm is presented with an ensemble of input patterns $\vec{x}_1$, ... ,$\vec{x}_P$. It determines the reference vectors and the corresponding partition using the following algorithm:

Step 0: initialize the reference vectors $\vec{c}_1$, ... , $\vec{c}_K$.

Step 1: for each $\vec{x}_i$, determine the membership indicator $M_k(x_i)$.

Step 2: compute the total spatial variation of the partition generated in step 1.

if its value is small enough, stop.

Step 3: updating reference vectors according to

$$\vec{c}_k = \{\sum_{i=1}^{P} M_k(\vec{x}_i)\vec{x}_i\}/\{\sum_{i=1}^{P} M_k(\vec{x}_i)\} \qquad \text{for } 1 \leq i \leq K.$$

Step 4: go to step 1.

It should be noted that such an iterative improvement algorithm need not necessarily converge to an optimum solution; its performance depends on the initial positions of the reference vectors. It is often useful, therefore, to enhance the algorithm performance by providing the algorithm with good initial reference vectors, and perhaps to start it with several different initial reference vector sets.

In addition to the batch mode of operation, the k-means algorithm can also operates in the *on-line* or *adaptive mode.* In the adaptive mode of operation, where the ensemble of $\vec{x}$ is not available, the k-means algorithm derives the reference vectors and the corresponding partition through time-averaging. It iteratively computes a new value of the reference vector $\vec{c}_{k,T+1}$ after each presentation of an input vector $\vec{x}_T$ using the following equation:

$$\vec{c}_{k,T+1} = \vec{c}_{k,T} + M_k(\vec{x}_T)\{\eta_{km}(\vec{x}_T - \vec{c}_{k,T})\}, \tag{2.3}$$

where η_{km} is the learning rate. For the *traditional* adaptive k-means algorithm, the learning rate η_{km} is defined to be constant.

2.2 Supervised Learning Algorithms Based on Gradient Descent

The objective of supervised learning is to adjust the function of a network so that it best approximates the goal function provided by training data. Let $\vec{g} : R^m \rightarrow R^n$ denote the goal function and $\vec{f} : R^m \rightarrow R^n$ denote the network function. Assume that the characteristics of the network function $\vec{f}$ depend on a parameter vector $\vec{w}$. To make explicit the dependence of $\vec{f}$ on parameter $\vec{w}$, we will write $\vec{f}(\vec{w}, \cdot)$ instead of $\vec{f}$. With the *mean squared error* criterion, supervised learning can be formulated as a problem of finding $\vec{w}^*$ that minimizes the cost function:

$$MSE = \int_{\mathcal{I}} p(\vec{x}) \, \| \vec{f}(\vec{w}, \vec{x}_i) - \vec{g}(\vec{x}_i) \|^2 \, d\vec{x}, \tag{2.4}$$

where $\| \ \|$ is a Euclidean distance and p is the distribution of $\vec{x}$ defined on the input domain $\mathcal{I}$.

In most actual applications, the exact information of distribution p is not available and must be derived through the training data. Assume that the training set consists of $(\vec{x}_1, \ \vec{g}(\vec{x}_1))$,..., $(\vec{x}_P, \ \vec{g}(\vec{x}_P))$. We thus instead minimize in practice the following cost function:

$$MSE = \sum_{i}^{P} e(\vec{w}, \vec{x}_i) \qquad \text{with} \qquad e(\vec{w}, \vec{x}_i) = \| \vec{f}(\vec{w}, \vec{x}_i) - \vec{g}(\vec{x}_i) \|^2. \tag{2.5}$$

Many supervised learning algorithms have been proposed; however, most of them are developed based on the gradient descent principle. Examples of supervised learning algorithms based on gradient descent are the least mean squared (*LMS*) algorithm [31] and the error back-

propagation algorithm [2]. These supervised learning algorithms have proven to be effective for many applications despite their simplicity.

Assume network function $\vec{f}$ has continuous first partial derivatives on R^M. For the batch mode of operation, where the entire training set is available, the algorithm based on gradient descent for finding $\vec{w}^*$ that minimizes the *MSE* in equation 2.5 is defined by the iterative equation:

$$\vec{w}_{T+1} = \vec{w}_T - \eta_{gd} \sum_{i=1}^{P} \nabla e(\vec{w}_T, \vec{x}_i), \tag{2.6}$$

where η_{gd} is a nonnegative scalar constant. According to this algorithm, we take a step from point $\vec{w}_T$ along the direction of the negative gradient of *MSE* to a new starting point $\vec{w}_{T+1}$.

In some applications, the algorithm is confined to work with one sample of an input-output pair at a time. In this *on-line* or *stochastic* mode of operation, the algorithm for minimizing *MSE* based on gradient descent is defined as:

$$\vec{w}_{T+1} = \vec{w}_T - \eta_{gd} \nabla e(\vec{w}_T, \vec{x}_T). \tag{2.7}$$

where $\vec{x}_T$ is the sample of input vector at time *T*. Note that the gradient in equation 2.7 is estimated by $\nabla e(\vec{w}_T, \vec{x}_T)$. The on-line version of the supervised learning algorithm based on gradient descent may be preferable to the batch version when the training set is large. In addition, the noise due to the use of an estimated gradient can help the algorithm escape from local minima, allowing the on-line algorithm to achieve a better performance than the batch algorithm.

2.3 Structure and Learning Algorithm of Heterogeneous Architecture

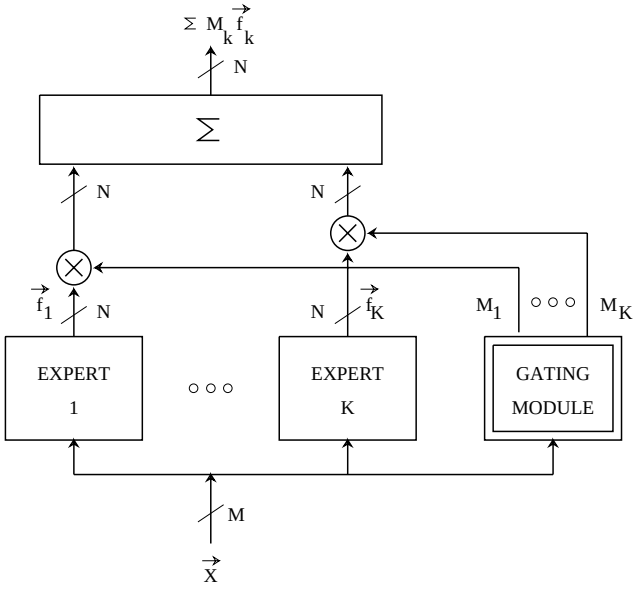


Figure 2.1: Schematic diagram of a heterogeneous architecture based on k-means partitioning.

Figure 2.1 shows the schematic diagram of a heterogeneous architecture based on k-means partitioning. It depicts a system that implements a mapping $\vec{f}$ from the input domain $\mathcal{I}$ in R^M to R^N. As shown in the diagram, this system of heterogeneous architectures is composed of a *k-means* gating module and K *expert* modules. The task of the gating module is to generate membership indicators M_k using the k-means algorithm, and the task of *expert* module k is to generate a function $\vec{f_k}$. The output of the system is defined to be

$$\vec{f}(\vec{x}) = \sum_{k=1}^{K} M_k(\vec{x})\vec{f_k}(\vec{x}). \tag{2.8}$$

When such a heterogeneous architecture is used to approximate a goal function $\vec{g}$, the system partitions the input domain $\mathcal{I}$ of $\vec{g}$ into non-overlapping regions using the k-means algorithm.

For a system with K expert modules, the input domain $\mathcal{I}$ is partitioned into K regions: $\mathcal{I}_1$, ... , $\mathcal{I}_K$, which are specified by K membership functions: M_1 , ... , M_K. In its original form, where the traditional k-means algorithm is used, the membership indicator M_k is based solely on the distribution p, and defined according to equation 2.2.

Using the partitioning generated by the k-means algorithm, the system then decomposes the goal function $\vec{g}$ into K component functions: $\{\vec{g}_1, ..., \vec{g}_K\}$, We use $\vec{g}_k$ to denote the *restriction* of $\vec{g}$ to $\mathcal{I}_k$. Each of these $\vec{g}_k$ is then approximated by an expert module $\vec{f}_k$, whose characteristics depends on parameter $\vec{w}_k$. We typically set $\vec{w}_k$ so that the *partial* mean squared error MSE_k is minimized, where the MSE_k is defined as:

$$MSE_k = \int_{\mathcal{I}_k} p(\vec{x}) \, \| \vec{f}_k(\vec{w}_k, \vec{x}) - \vec{g}_k(\vec{x}) \|^2 \, d\vec{x}. \tag{2.9}$$

These partial mean squared errors are usually minimized using supervised learning based on gradient descent.

In this dissertation, we are interested in the on-line version of the heterogeneous architectures based on k-means partitioning. In the batch mode of operation, the parameters in the system are updated only after all the patterns in the training set have been presented. In the on-line mode of operation, the parameters are adjusted after each pattern presentation. Since a system with on-line learning works with one training pattern at a time, the system does not have to store all the training data and it can run "live" with a process that generates new data continuously. Since the parameters are constantly updated in the on-line mode, a system with on-line learning can track changes in the statistics of the training data faster than a system with batch learning. In addition, a system trained by an on-line learning algorithm tends to learn faster, especially for the case of large training sets.

As a demonstration of how the heterogeneous architectures based on k-means partitioning solve their assigned problems, we apply a system in these architectures to approximate a 1-dimensional function of the form:

$$g(x) = 4x^2, \tag{2.10}$$

where $\vec{x}$ is a random variable with uniform distribution on $[-0.5, 0.5]$. The system used in this demonstration is composed of 4 expert modules each with a linear network function f_k and of a k-means module that partitions the input domain into 4 regions. Overall, the system generates the network function of the form:

$$f(x) = \sum_{k=1}^{4} M_k(x) f_k(x). \tag{2.11}$$

Membership indicator M_k is defined based on the reference points c_1, c_2, c_3, and c_4 of the k-means module, and it has the following form:

$$M_k(x) = \begin{cases} 1 \quad \text{if} \quad \|x - c_k\| \ \leq \ \|x - c_i\| \quad \text{for each } i \neq k \\ 0 \quad \text{otherwise} \end{cases} . \tag{2.12}$$

In the event of more than one minimum $\|\vec{x} - \vec{c}_k\|$, we set the M_k with the lowest index k to 1 and the others to 0. The linear function f_k is expressed in the form:

$$f_k(x) = a_k + b_k x, \tag{2.13}$$

where a_k is the constant of f_k and b_k is the coefficient of x. We start training the above system by initializing c_1 c_2, c_3 and c_4 to 0; and all the parameters of expert modules f_1, f_2 f_3 and f_4 to 10^{-10}. We then iteratively update these parameters using a random sequence of input-output pairs

generated according to equation 2.10.

Let x_T and $c_{k,T}$ denote the input x and the reference point c_k at iteration *T*. Also, let $a_{k,T}$ and $b_{k,T}$ denote the two parameters of f_k at iteration *T*. The algorithm for updating the system parameters in the *T*-th iteration is as follows:

Algorithm:

Step 1: compute the network output $f(x_T)$ according to equations 2.11.

Step 2: update the reference points c_k according to

$$c_{k,T+1} = c_{k,T} + M_k(x_T)\{\eta_{km}(x_T - c_{k,T})\},$$

where the learning rate η_{km} is defined to be 0.01.

Step 3: update the parameter of the expert modules according to

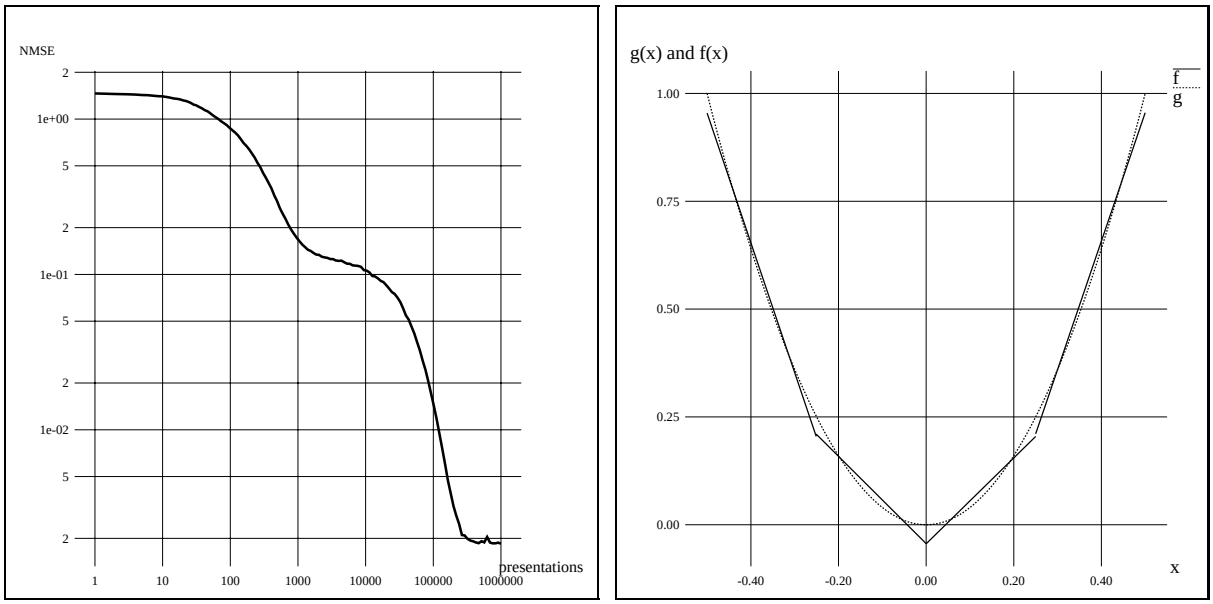
$$a_{k,T+1} = a_{k,T} + \eta_{lms} M_k(x_T)\delta_T \qquad \text{for } 1 \le k \le 4$$

$$b_{k,T+1} = b_{k,T} + \eta_{lms} M_k(x_T)\delta_T x_T \qquad \text{for } 1 \le k \le 4$$

where η_{lms} is defined to be 0.01 and δ_T is defined to be $f(x_T) - g(x_T)$.

Figure 2.2 illustrates the performance of the heterogeneous system on a randomized input-output sequence generated according to eqaution 2.10. Figure 2.2a shows the normalized mean squared error (*NMSE*) as a function of the number of patterns presented. The normalized mean squared error is defined as the mean squared error between f and g normalized by the mean squared value of g. Each curve here is the average of five runs, each with different training pattern sequences.

Figure 2.2b depicts the function f of the heterogeneous system obtained after 100,000 pattern presentations, compared to the goal function g. The performance of a heterogeneous architecture strongly depends on the partitioning of the input domain. For this simple illustration, where g is a 1-dimensional quadratic polynomial and x is uniformly distributed in the input domain, the partition that minimizes the total spatial variation defined in equation 2.1 and the partition that minimizes the mean squared error defined in equation 2.4 are the same. As a result, the traditional



(a) (b)

Figure 2.2: A simple demonstration example with a 1-dimensional quadratic function. (a) The *NMSE* vs. the number of patterns presented. (b) The goal function g and the network function f.

k-means algorithm is able to generate a partition that optimally minimizes the corresponding mean squared error between f and g. For more complicated problems, the partition with minimum total spatial variation and that with minimum mean squared error are usually different. Thus, in general, the traditional k-means algorithm cannot partition the input domain of a heterogeneous architecture optimally. In chapter 3, we introduce some mechanisms to improve the partitioning capabilities. Using the insight gained from chapter 3, we then introduce in chapter 4 an improved k-means algorithm that is well suited to partition the input domain of a heterogeneous architecture.

Chapter 3

Adaptive K-Means Algorithm with Variation-Weighted Deviation Measure

We introduce in this chapter an enhancement of the traditional k-means algorithm that attempts to minimize the *total spatial variation*:

$$TSV = \sum_{k=1}^{K} v_k \qquad \text{with} \qquad v_k = \int_{\mathcal{I}_k} p(\vec{x}) \|\vec{x} - \vec{c}_k\|^2 \, d\vec{x}, \tag{3.1}$$

This *TSV* cost function is commonly used for input feature extraction and signal compression. The new k-means algorithm approximates an optimal clustering solution with an efficient adaptive learning rate, which renders it usable even in situations where the statistics of the problem task vary slowly with time. It has been shown to perform better than other k-means variants on several tutorial examples, and also on vector quantization coding of image data.

3.1 Problems of the Traditional Adaptive K-Means Algorithm

One serious problem with most k-means algorithms is that the clustering process may not converge to an optimal or near-optimal configuration. The algorithm can assure only local optimality, which depends on the initial locations of the representative vectors [32]. Some initial reference vectors may get stuck in regions of the input domain with few or no input patterns, and may not move to where they are needed. A traditional way to deal with this under-utilization problem is to employ *leaky learning* [33] where, in addition to adjusting the closest reference vector, other reference vectors are also adjusted but with smaller learning rates. Another approach is the *conscience learning law* [34] where the determination of the closest reference vector uses a norm that favors the reference vectors that in the past have responded to fewer patterns, thus equalizing the average rates of winning for each region. However, these two methods yield partitions that are not optimal or near optimal with respect to the total spatial variation cost function, since the added mechanisms have the effect to distort the cost function. Moreover, leaky learning increases the amount of computation required for each pattern presentation since *all* the reference vectors have to be updated.

In the on-line mode, the performance of the k-means algorithm depends strongly on the learning rate. There is a trade-off between the *dynamic performance* (rate of convergence) and the *steady-state performance* (residual deviation from the optimal solution). When using a fixed learning rate, it must be sufficiently small for the adaptation process to converge. The smaller the learning rate, the smaller the residual deviation but the slower the convergence rate. Optimal learning rates cannot be determined in advance since they are problem dependent; normally a conservative value is chosen initially and then improved by trial-and-error. Because of this difficulty, adaptive

k-means algorithms with *variable* learning rates have been investigated. Darken and Moody first proposed to make the learning rate of each cluster center equal to the inverse of the square root of the number of patterns assigned to that center [35]. Since the convergence of this learning schedule is very slow, they later proposed a *search-then-converge* schedule, where $\eta_T = \eta_0/(1 + T/\tau)$ [36]. In this scheme, the learning rate stays near η_0 for a search time τ and then decreases at the rate of $1/T$. For many problems this can yield precise convergence in short times. However, it is not possible to determine an a priori best search time τ for all possible problems. Furthermore, such approaches with pre-determined learning rates are not flexible enough to handle problems with time-varying characteristics.

This chapter presents an alternative approach that solves both of the above two problems which have been attacked independently in previous work. We describe a method that has the following characteristics:

- It allows the adaptation process to escape from bad minima without distorting the cost function for asymptotically large K.
- It dynamically adjusts the learning rate based on the quality of the current clustering.
- It is applicable to situations where patterns are generated from sources with non-stationary distributions.

3.2 Optimal Criterion for an Adaptive K-Means Clustering

As mentioned in section 2.1, the total spatial variation of the k-means algorithm which we are trying to minimize is given by:

$$TSV = \sum_{k=1}^{K} v_k \qquad \text{with} \qquad v_k = \int_{\mathcal{I}_k} p(\vec{x}) \, \|\vec{x} - \vec{c}_k\|^2 \, d\vec{x}. \tag{3.2}$$

where v_k represents the spatial variation in region $\mathcal{I}_k$. Its value depends on the location of $\vec{c}_k$ and the geometrical properties of the region $\mathcal{I}_k$. Gersho [37] showed that:

For a continuous underlying probability density p and large K, all regions in an optimal Voronoi partition have the same spatial variations v_k.

Because the k-means algorithm produces a Voronoi partition, we conjecture that it is worthwhile to aim for a partition in which all the regions have the same variations v_k, even if K is small and the distribution is non-smooth. This goal is built directly into the cost function based on which the reference vectors are adjusted. In this manner we can eliminate most bad clustering configurations from the solution set and thus increase the chance of finding an optimal or near-optimal solution.

3.3 Variation-Weighted Deviation Measure

We seek to improve the capability of the k-means algorithm in partitioning an input domain by including the above optimality criterion into the deviation measure of the algorithm. Such inclusion is achieved by the use of the *variation-weighted* deviation measure. In this measure,

the deviation between $\vec{x}$ and $\vec{c}_k$ is defined as:

$$d(\vec{x}, \vec{c}_k) = v_k \|\vec{x} - \vec{c}_k\|^2. \tag{3.3}$$

This deviation measure thus results in a *variation-weighted membership indicator* $M_{k,vwgt}$ of the form:

$$M_{k,vwgt}(\vec{x}) = \begin{cases} 1 & \text{if } v_k \|\vec{x} - \vec{c}_k\|^2 \leq v_i \|\vec{x} - \vec{c}_i\|^2 \quad \text{for each } i \neq k \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

In the event of more than one minimum $v_k \|\vec{x} - \vec{c}_k\|$, we set the M_k with the lowest index k to 1 and the others to 0. Multiplying the Euclidean deviation measure by variation v_k biases the membership indicator in favor of regions with smaller variation v_k, and thus leads to the robust equalization of v_k among all the regions in the partition.

With the variation-weighted deviation measure, the k-means algorithm attempts to minimize the total v_k-weighted variation:

$$\textit{TVV} = \sum_{k=1}^{K} v_k \int_{\mathcal{I}_k} p(\vec{x}) \|\vec{x} - \vec{c}_k\|^2 \, d\vec{x} = \sum_{k=1}^{K} v_k^2, \tag{3.5}$$

assuming that v_k can be perfectly estimated. It is shown in Appendix A that the clustering process based on minimizing *TVV* is capable of attaining a solution that is optimal with respect to the total spatial variation for asymptotically large K. This indicates that in the ideal situation where K is large and v_k is perfectly estimated, the above biased deviation measure forces all regions to "equally share the load" without distorting the total spatial variation cost function, as is the case with other equalization schemes.

To obtain the estimated variation $\hat{v}_k$ for each region, we use the following simple, weighted running time-average :

$$\hat{v}_{k,\,T+1} = \alpha\ \hat{v}_{k,\,T}\ \ +\ \ (1-\alpha)\ \left\{ M_{k,vwgt}(\vec{x}_T)\,\|\vec{x}_T - \vec{c}_{k,T}\|^2 \right\}, \tag{3.6}$$

In the first term, we multiply $\hat{v}_{k,\,T}$ by α which is a constant slightly less than 1. The purpose is to reduce the value of the previous estimate $\hat{v}_{k,\,T}$. In the second term, we add to the new estimate $\hat{v}_{k,\,T+1}$ new information about the variation v_k. Note that the closer α is to 1, the more accurate the estimate of v_k, but the longer the estimation time constant. We start this estimation by initializing all $\hat{v}_{k,0}$ to the same small number. This allows the effect of the initialization to disappear quickly so that the estimated $\hat{v}_k$ is soon dominated by the actual data seen by each region.

3.4 Dynamic Adjustment of Learning Rate

An optimal value for the instantaneous learning rate could be derived from the difference between the quality of the partition at that moment and that of a known target partition: When the partition is far from its destination, the learning rate should be large so that the partition can improve quickly. As the partition gets closer to its target, the learning rate should be reduced in order to minimize the residual deviation from the target solution.

The success of such a method depends on how the *quality* of a partition is measured. For an optimal solution, the variations v_k for all regions in a target partition must be equal. Our estimate of quality is thus based on the *similarity* of the current v_k's and is derived from the entropy of the normalized values of variations v_k. Thus, the quality of a partition having variations v_1, v_2, ..., v_K

is defined to be:

$$H(v_1, v_2, ..., v_K) = \sum_{k=1}^{K} -v_{k,norm} \ \ln(v_{k,norm}) \qquad \text{with} \quad v_{k,norm} = v_k / (\sum_{i=1}^{K} v_i) \qquad (3.7)$$

This approach appears to be a "natural" choice for the quality measure, as it does not rely on any arbitrary constant or user-adjustable parameters. According to this measure, the quality of a final partition is a maximum equal to $\ln(K)$, and it occurs when $v_{1,norm} = v_{2,norm} = ... = v_{K,norm} = 1/K$. Using this measure, we can define the learning rate η at time *T* as:

$$\eta = \frac{\ln(K) - H(v_1, v_2, ..., v_K)}{\ln(K)}. \qquad (3.8)$$

This learning rate depends only on the current values of the variations v_k; it thus allows us to compute a learning rate without any knowledge of the final partition. This learning rate is automatically limited to the range of 0 and 1; it is close to 1 when the current partition given by the algorithm is far from an optimal solution, and close to 0 when it is close to a final optimal partition with all v_k being equal.

Such an automatic determination of the learning rate is preferable to any pre-determined rate or schedule. The learning rate automatically adjusts to the problem characteristics and requires neither user interaction nor prior information about the task. It is also applicable to problems whose statistics vary slowly with time, or occasionally show a sudden change.

3.5 Illustrative Experiments

In this section, we investigate the performance of the optimal k-means algorithm and compare it to other versions of the k-means algorithm on several simple tutorial examples. The training patterns used in these examples are generated from synthetic probability distributions including both stationary (section 3.5.1) and non-stationary (section 3.5.2) statistics. In section 3.6, we then evaluate the optimal k-means algorithm on a practical application: vector quantization.

3.5.1 Stationary Distributions

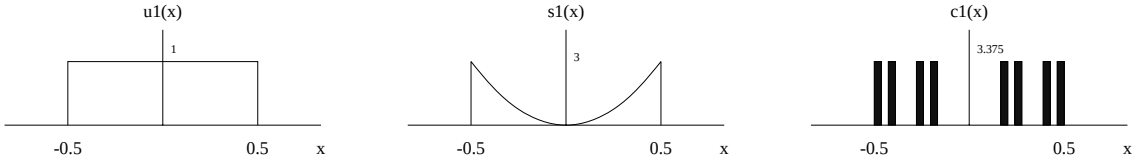
This subsection presents the results of an empirical comparison on problems whose pattern distributions are stationary, i.e., the probability distributions of the training patterns do not change with time. In this subsection, we compare the following four k-means algorithms:

- *Optm* : the proposed optimal adaptive k-means algorithm, ($\alpha = 0.9999$);
- *Cons* : the adaptive k-means algorithm with the conscience learning rule [34], ($B = 0.0001$,[1] $\eta = 0.01$);
- *Sqrt* : the adaptive k-means algorithm with the square root learning rate [35];
- *Trad* : the traditional adaptive k-means algorithm [27], (η = 0.01).

We empirically evaluate these four algorithms on seven situations corresponding to three probability distributions in 1 dimension and four distributions in 2 dimensions. The three 1-dimensional distributions used in this simulation are : uniform, square, and 3-level Cantor distributions (Fig. 3.1). The data in the uniform distribution are uniformly distributed in $[-0.5, 0.5]$. The

[1] This is the value used by Desieno [34] and it is equivalent to $\alpha = 0.9999$.

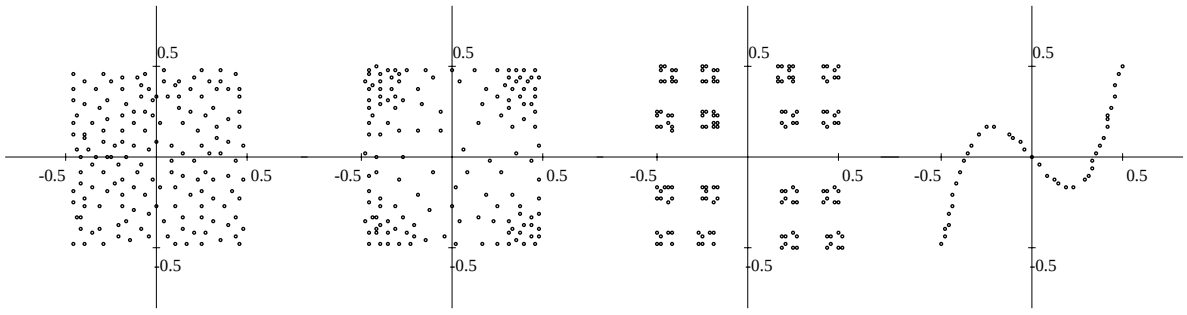
density of the square distribution is proportional to the squared distance from the origin. The distribution of the 3-level Cantor set is uniformly distributed on a fractal; it may be formed by starting with the unit interval, removing its middle third, and then recursively repeating the procedure on the two portions of the interval that are left.



(a) (b) (c)

Figure 3.1: Distributions for the 1-dimensional training sets: (a) uniform, (b) square, and (c) 3-level Cantor.

For the 2-dimensional distributions, we employ: uniform, square, 3-level Cantor, and 1-dimensional sub manifold distributions. The first three distributions are simply the products of two corresponding distributions in 1 dimension. The fourth distribution consists of data points that are restricted to lie on a S-shaped curve defined by the equation $y = 8x^3 - x$, where x and y are the horizontal and vertical ordinates, respectively, and x is uniformly distributed in the interval $[-0.5, 0.5]$. Sample sets drawn from these 2-dimensional distributions are shown in Figure 3.2.



(a) (b) (c) (d)

Figure 3.2: Data drawn from the 2-dimensional test distributions: (a) uniform, (b) square, (c) 3-level Cantor, and (d) 1-dimensional sub manifold.

Figure 3.3 shows the simulation results of the four algorithms on the 1-dimensional distribution problems. In these 1-dimensional problems, we evaluate each algorithm on a sequence of patterns randomly chosen from a set of 2000 patterns generated according to a specified test distribution. We divide the input domain into 10 regions (k = 10) and initialize the reference vectors to uniformly distributed random locations in the input domain. For the *Optm* algorithm, we initialize the variation v_k of each region to be 10^{-10}, and for the *Cons* algorithm, we initialize the winning probability of each region to be $(1/K)$ = 0.1. To improve statistical accuracy, we average the simulation results over 5 runs, each with different pattern sequences and different initial reference vectors. The same pattern sequences and the same initial reference vectors are applied to every algorithm in order to achieve a fair comparison. We measure the performance of each algorithm using the *normalized total spatial variation* (*NTSV*), defined as the total spatial variation divided by the variance of $\vec{x}$. Using the *NTSV* as the performance measure makes the simulation results invariant to the spatial scaling of input $\vec{x}$.

Figure 3.3a shows the simulation results of the four algorithms on the 1-dimensional uniform distribution problem. We plot in this figure the residual deviation of *NTSV* from the computed optimum value with respect to time measured by *the number of pattern presentations*. The simulation indicates that at the beginning, the *Optm* algorithm is overly responsive because we start the simulation with almost zero variations. Until each region has seen about 3 to 5 patterns, their boundaries change greatly in response to every new data pattern. After a reasonable estimate of the v_k's has been built up, the algorithm minimizes the *NTSV* rapidly. Its *NTSV* becomes lower than that of the other algorithms after about 1000 presentations. Asymptotically, the new algorithm approaches the optimum value more closely than any of the others.

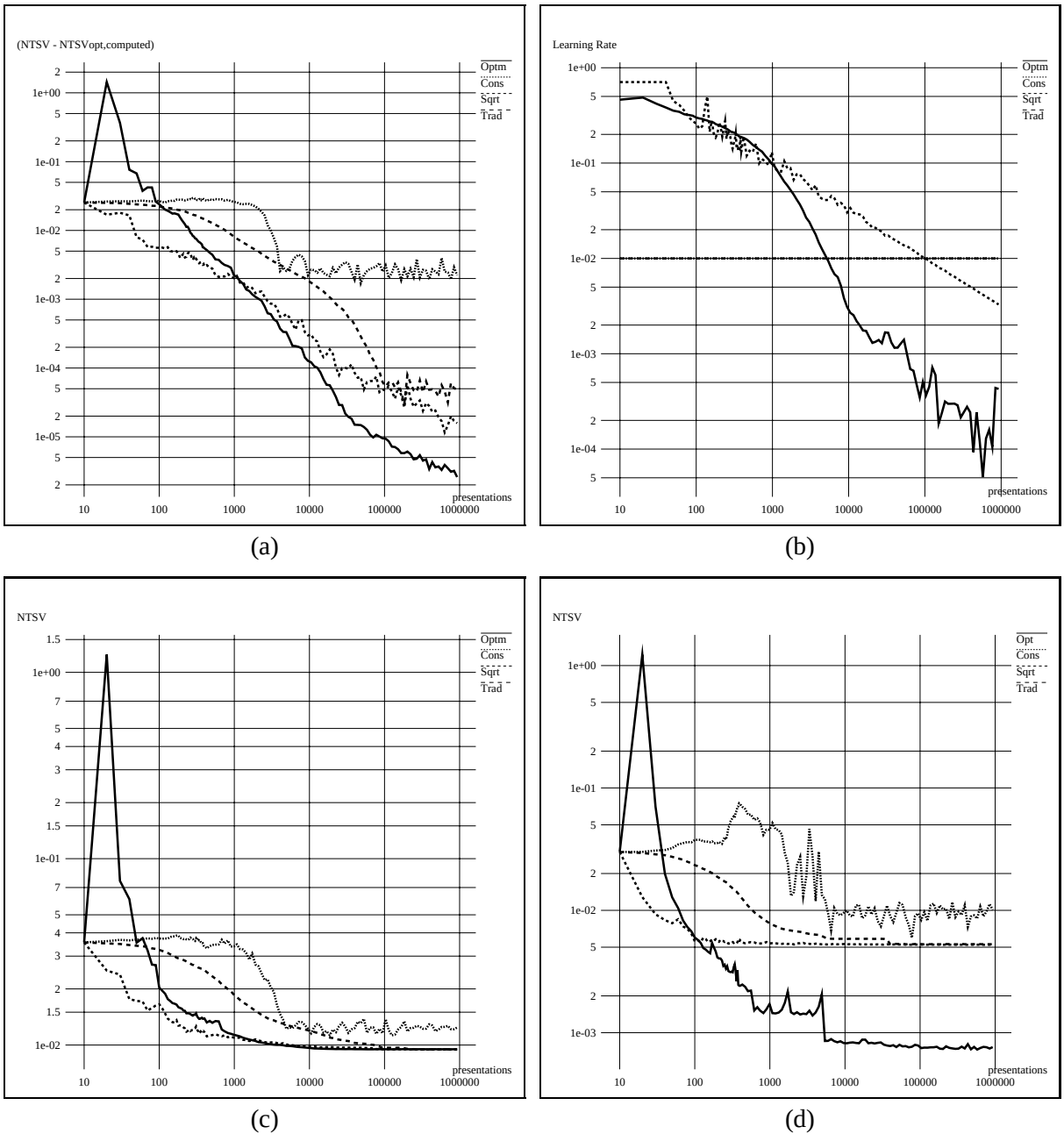


Figure 3.3: Average simulation runs of the four k-means algorithms on the 1-dimensional distribution problems. (a) The simulations on 1-dimensional uniform distribution. (b) The learning rates on 1-dimensional uniform distribution. (c) The simulations on 1-dimensional square distribution. (d) The simulations on 1-dimensional Cantor distribution.

Figure 3.3b shows the learning rates η for this 1-dimensional uniform distribution problem. *Cons* and *Trad* have fixed learning rates of magnitude 0.01. The square-root schedule is roughly a straight line with slope -1/2 in this log-log plot. For the *Optm* algorithm, its learning rate automatically behaves similar to that of a search-then-converge schedule, with an initial high learning rate, which then declines rapidly with a slope close to the inverse of the number of patterns assigned to each cluster. The time for the break-point between the two phases is determined automatically by the nature of the problem.

For the square distribution, the performance of each algorithm is similar to that of the uniform case. The new algorithm *Optm* surpasses the others after about 2000 presentations (Fig. 3.3c). For the case of the 3-level Cantor distribution, where the *NTSV* cost function has bad local minima, the *Optm* algorithm performs much better than the others; it approaches a final value which is one tenth of that approached by the *Cons* algorithm and one fifth of that approached by *Sqrt* and *Trad* (Fig. 3.3d). The *Optm* algorithm can achieve a lower *NTSV* than other algorithms because it avoids being trapped in a bad local minimum. In 5 out of 5 runs, the *Optm* algorithm can find the good clustering configuration (Fig. 3.4a), whereas, the *Trad* and *Sqrt* algorithms find good partitions only in 2 out of 5 runs (Fig. 3.4c-d). The *Cons* algorithm has difficulties locating the good solution due to the conflicting goals between the *NTSV* cost function and the bias mechanism for equalizing the winning probability of each cluster (Fig. 3.4b). The fact that the *Optm* algorithm found a good solution in 5 out of 5 runs indicates that it is insensitive to the initialization specifications.

Figures 3.5 shows the simulation results of the four algorithms on the 2-dimensional problems. We derived these simulation results using a similar procedure as that employed in the 1-dimensional problems except that we divide each input domain of these 2-dimensional problems into

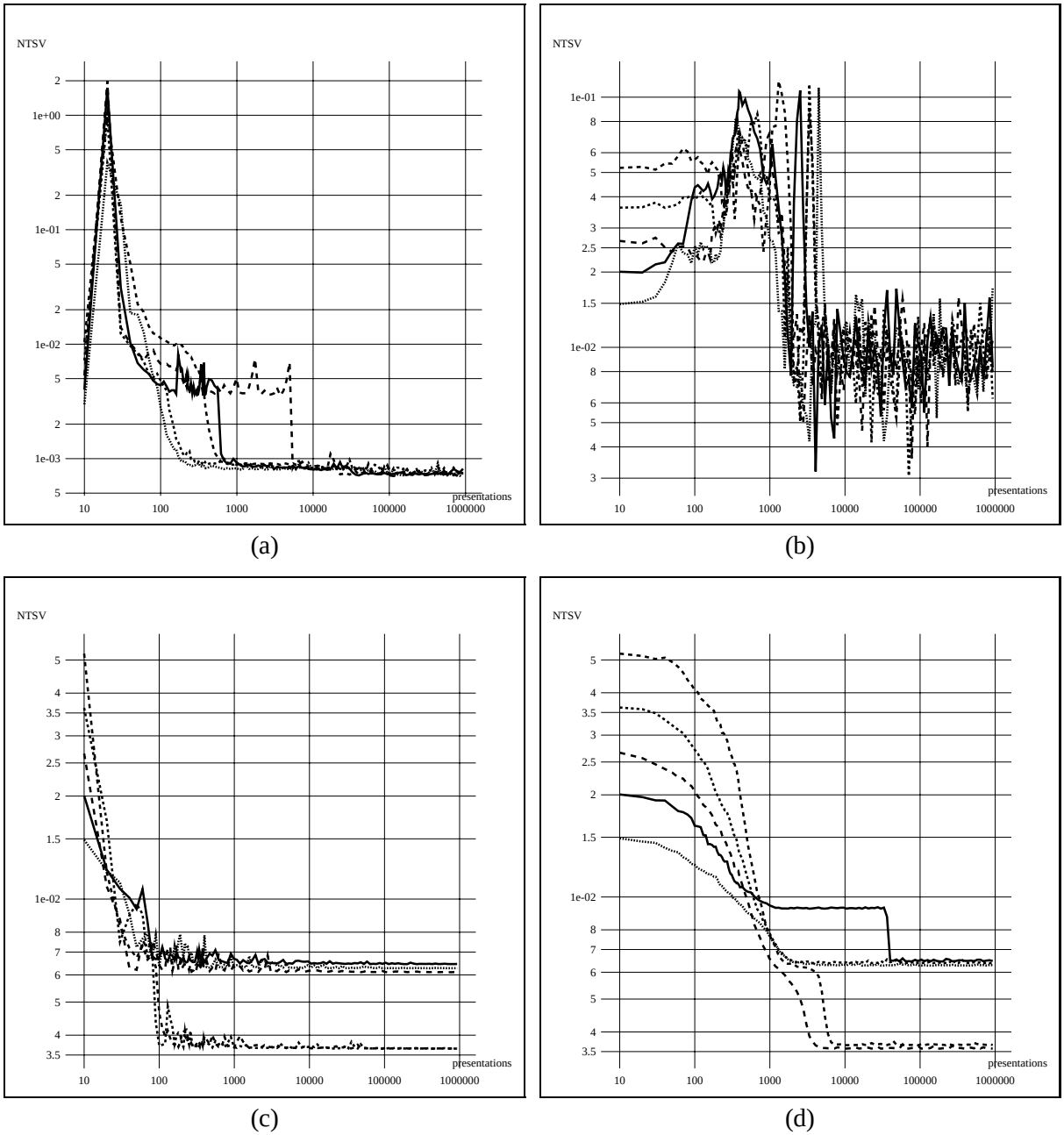


Figure 3.4: Five individual simulation runs of the four k-means algorithms on the 1-dimensional 3-level Cantor distribution problem. (a) The simulations of the *Optm* algorithm. (b) The simulations of the *Cons* algorithm. (c) The simulations of the *Sqrt* algorithm. (d) The simulations of the *Trad* algorithm.

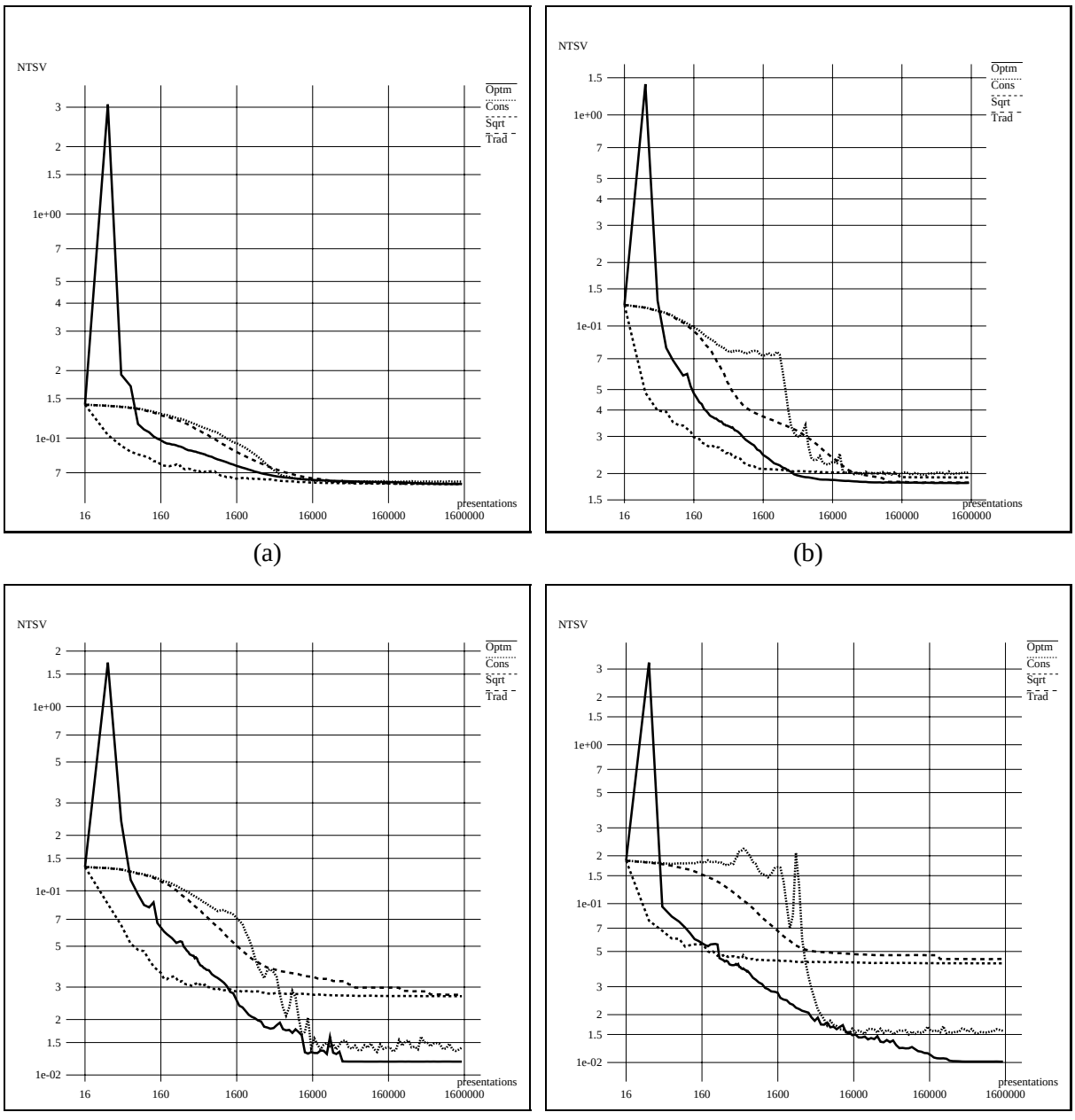


Figure 3.5: Average simulation runs of the four k-means algorithms on the 2-dimensional distribution problems. The distributions that generate the training set are (a) uniform, (b) square, (c) 3-level Cantor (4) and 1-dimensional sub-manifold.

16 regions (K = 16). Again, each curve is the average of 5 individual simulation runs with different random starting reference vectors and different pattern sequences. For the uniform distribution, the *NTSV* of the *Sqrt* algorithm drops faster than the others in the initial phase. However, in steady state the *Optm* algorithm out-performs the others by the some small degree (Fig. 3.5a). For the square distribution problem, the *NTSV* given by *Optm* falls below that of the other three algorithms after about 4000 presentations (Fig. 3.5b). In the case of Cantor distribution and the 1-dimensional sub-manifold problems, the new algorithm *Optm* performs much better than the other algorithms because it can avoid being trapped in bad local minima (Fig. 3.5c-d).

Figure 3.6 shows resulting partitions obtained from the four algorithms on the Cantor set problem after 10^6 pattern presentations. A small square indicates the center of a cluster, and the surrounding ellipse represents the size of a cluster. The ellipse associated with cluster i is defined as

$$\frac{(x - c_{x,i})^2}{\sigma_x^2} + \frac{(y - c_{y,i})^2}{\sigma_y^2} = 1, \tag{3.9}$$

where $c_{x,i}$ and $c_{y,i}$ are the horizontal and the vertical ordinates of the i-th reference vectors, and σ_x^2 and σ_y^2 are the x and y variances of the patterns in region i. The *Optm* algorithm produces an optimal partition (Fig. 3.6a). The partition obtained from *Cons* is inferior to that of *Optm* since some clusters cover more patterns than others (Fig. 3.6b). In the cases of *Trad* and *Sqrt*, their partitions are even worse since some of the clusters cover no patterns at all (Fig. 3.6c-d). These empty clusters are indicated by small squares without ellipses – two of them appear in the *Trad* case and one in the *Sqrt* case. Similar results are also observed in the case of the 1-dimensional sub manifold problem. For *Trad* and *Sqrt*, eight out of sixteen clusters cover no patterns (Fig. 3.7c-d). Also, because the bias mechanism of the *Cons* distorts the *NTSV* cost function, the partition given by *Optm* is closer

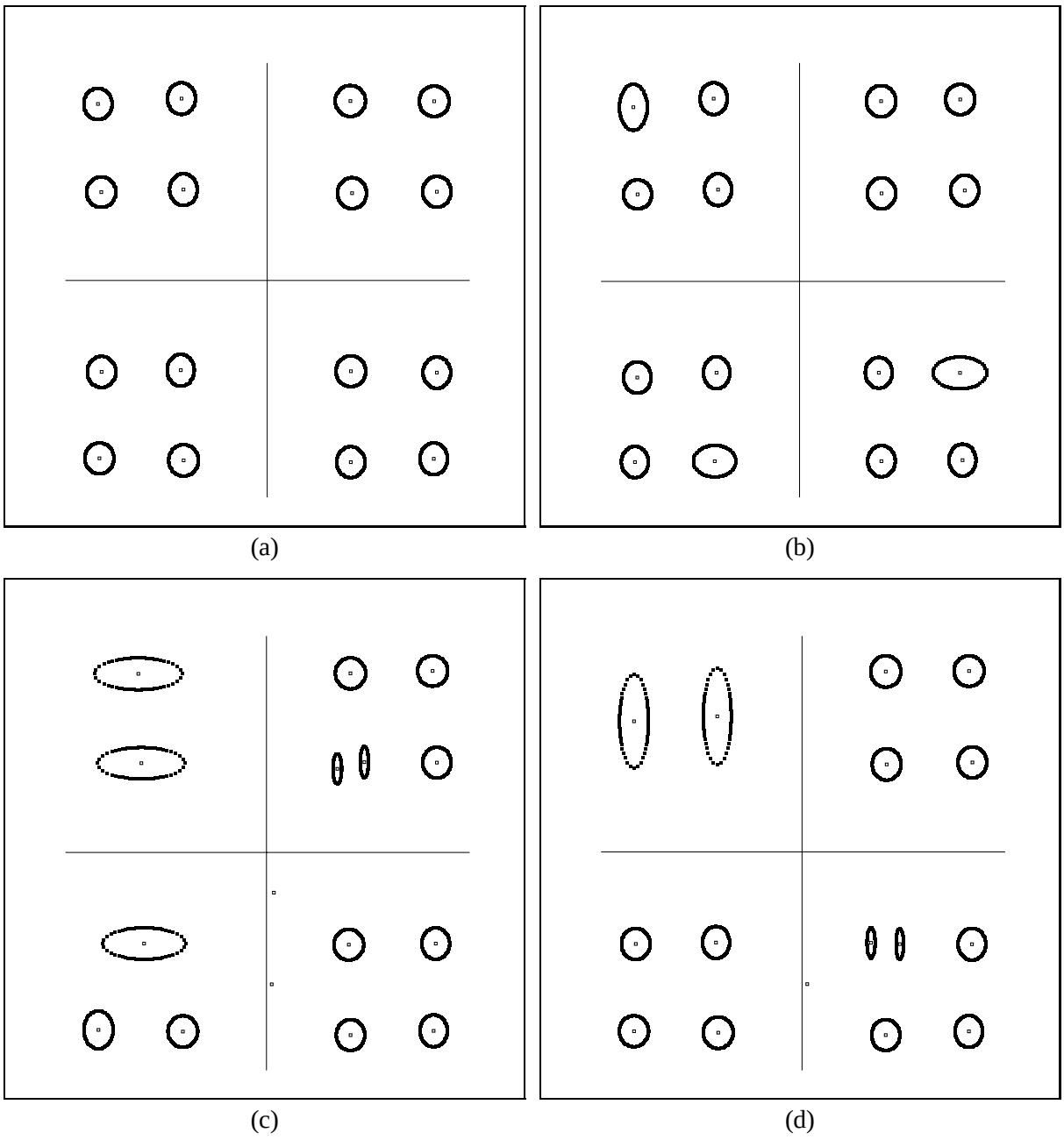


Figure 3.6: A sample of the partitions associated with the 2-dimensional 3-level Cantor distribution problem. These partitions are generated by the four k-means algorithms: (a) *Optm*, (b) *Cons*, (c) *Trad* and (d) *Sqrt*.

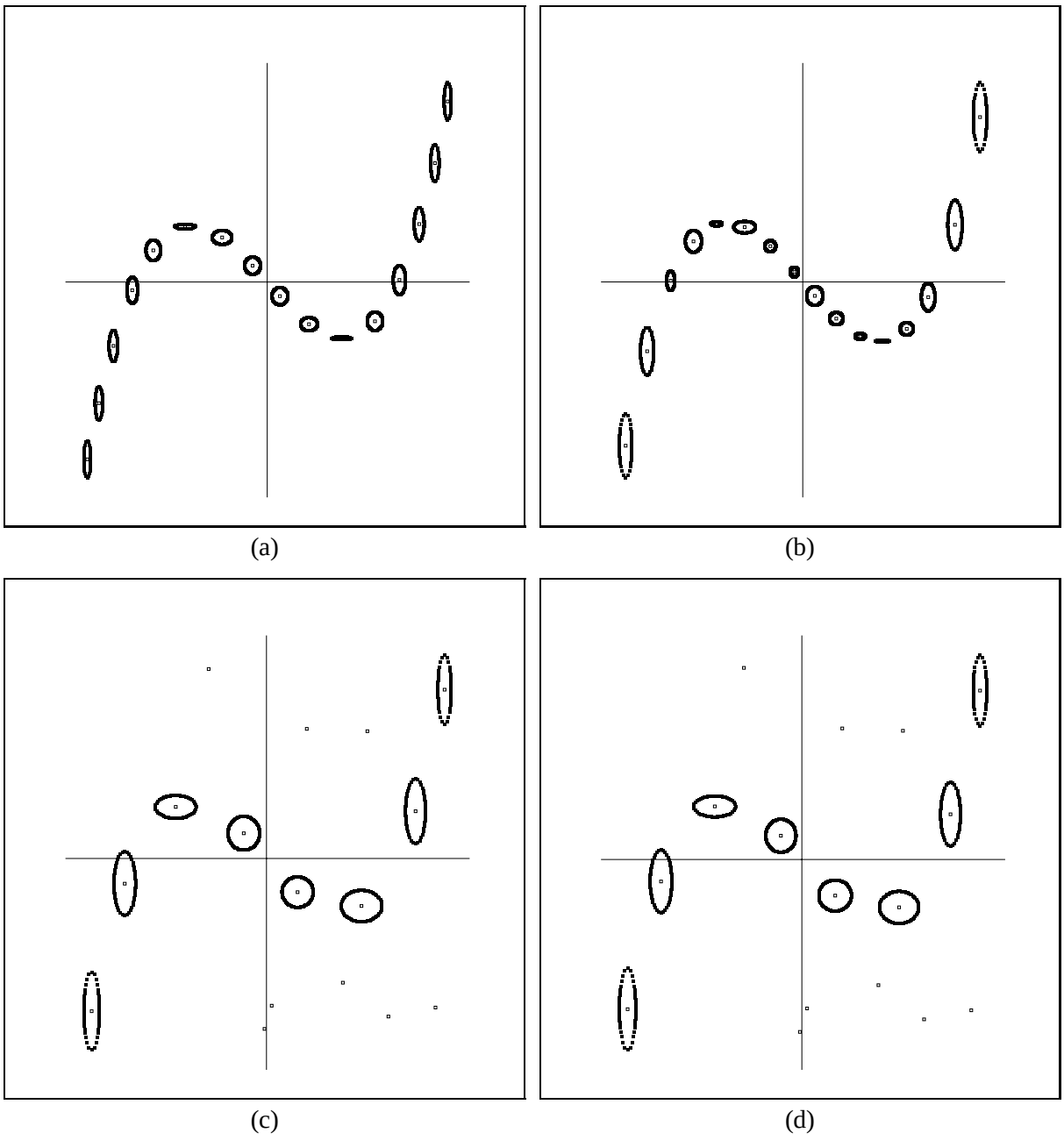


Figure 3.7: A sample of the partitions associated with the 1-dimensional sub manifold distribution problem. These partitions are generated by four versions of k-means algorithms: (a) *Optm*, (b) *Cons*, (c) *Trad* and (d) *Sqrt*. Note the eight unused reference vectors in cases (c) and (d).

to optimal than that of *Cons*.

3.5.2 Non-Stationary Distributions

This subsection evaluates the optimal k-means algorithm on problems whose patterns are derived from time-varying distributions. We compare in this subsection the performance of the following four adaptive k-means clustering algorithms:

- *Optm* : the proposed optimal k-means algorithm ($\alpha = 0.9999$).
- *Cons* : the adaptive k-means algorithm with the conscience learning rule [34], ($B = 0.0001$,[2] $\eta = 0.01$);
- *Trad1* : the traditional adaptive ("on-line") k-means algorithm [27] (η = 0.1).
- *Trad2* : the traditional adaptive ("on-line") k-means algorithm [27] (η = 0.01).

Because its learning rate is monotonically decreasing, the square root k-means algorithm is not applicable for non-stationary situations and thus is not included in this evaluation. Based on the 1-dimensional sub-manifold distribution (Fig. 3.2d) in section 3.5.1, we have constructed three non-stationary distributions. These distributions allow us to evaluate the performances of the aforementioned algorithms in the following non-stationary situations.

(1) Constantly rotating distribution: We continuously vary the underlying distribution by rotating the S-shaped curve counter-clockwise at the rate of 1 revolution per 20,000 pattern presentations.

[2] This is the value used by Desieno [34] and it is equivalent to $\alpha = 0.9999$.

(2) Constantly translating distribution: We continuously vary the underlying distribution by translating the S-shaped curve at the rate of 1 length-unit per 100,000 pattern presentations.

(3) Abrupt change in distribution: We abruptly change the distribution after learning has reached steady-state (after 10^5 presentations/cluster). We transform the S-shaped curve by mirroring the curve across the horizontal axis.

Figure 3.8a shows the simulation results of the four algorithms on the problem with constantly rotating statistics. As indicated in the figure, the *NTSV* of *Optm* and *Trad1* are lower than those of *Cons* and *Trad2*. Figure 3.8c shows the learning rates of the four algorithms for the same problem. It reveals that the learning rates of both *Optm* and *Trad1* are larger than that of *Cons* and *Trad2*. These larger learning rates allow *Optm* and *Trad1* to follow the changes more closely. Contrary to *Trad1* whose learning rate is pre-determined, the *Optm* algorithm dynamically adjust its learning rate to match the nature of the problem. Similar results are also observed in the case of the constantly translating distribution (Fig. 3.8b,d). To illustrate how well each algorithm follows the rotating distribution, we plot in Figure 3.9 the locations of the 16 reference vectors after the adaptation processes have reached stead state, i.e., after 8 full rotations of the S-shaped curve. This figure shows that only *Optm* and *Trad1* can follow the rotating distribution closely. Because of the v_k-equalization, *Optm* can produce a partition in which the reference vectors are more evenly distributed, and thus achieve a lower *NTSV* than *Trad1*. *Cons* can follow the rotation but with a substantial phase lag. *Trad2* with its small learning rate just averages over all the possible rotation angles.

Figure 3.10a shows the simulation result of the four algorithms on a data set with abruptly changing statistics. The simulation indicates that only *Optm* and *Cons* can regain a performance

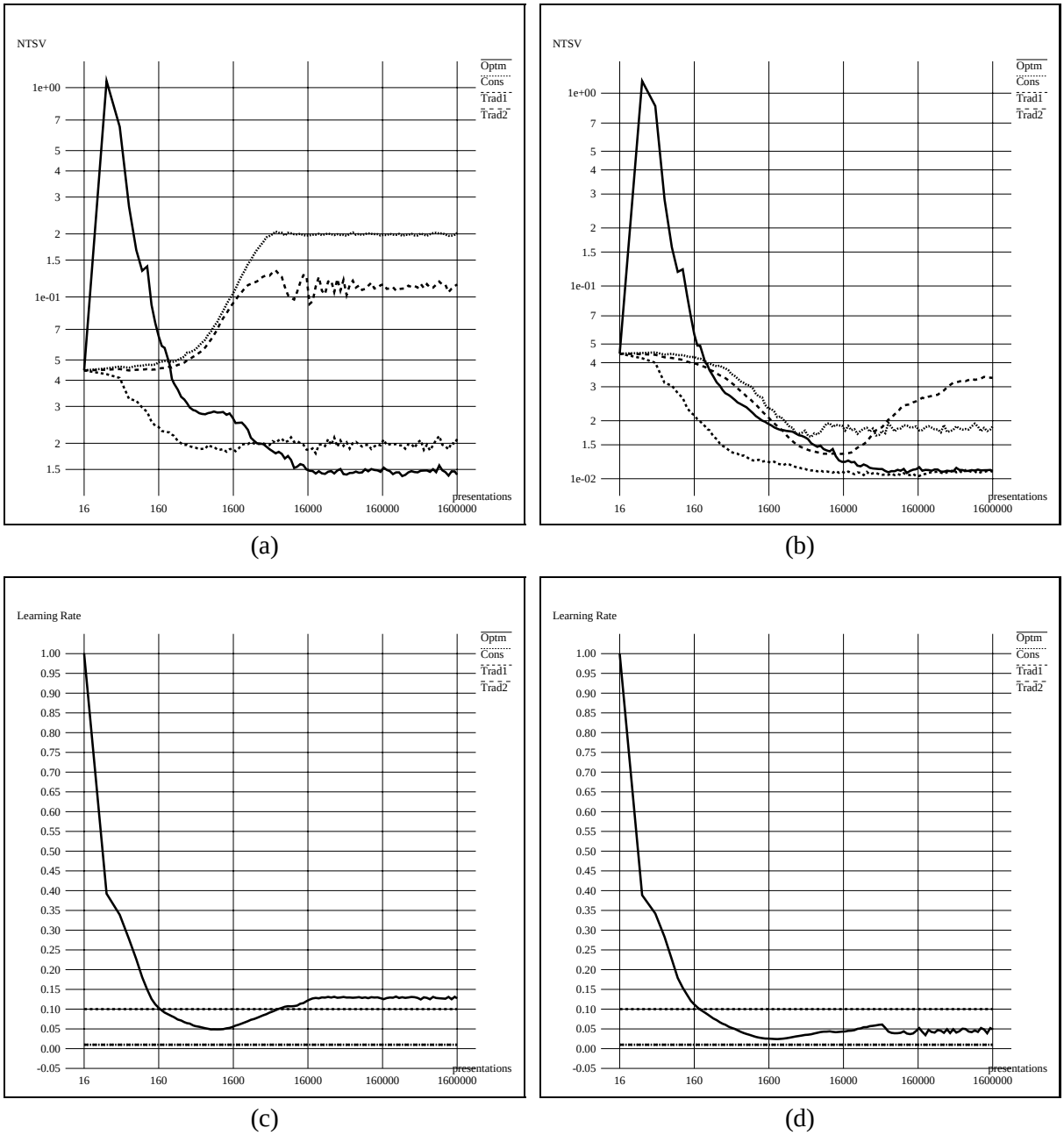


Figure 3.8: Average simulation runs of the four k-means algorithms on the problems with constantly changing statistics. (a) The simulations on the rotating pattern distribution. (b) The simulations on the translating pattern distribution. (c) The learning rates on the rotating pattern distribution. (d) The learning rates on the translating pattern distribution.

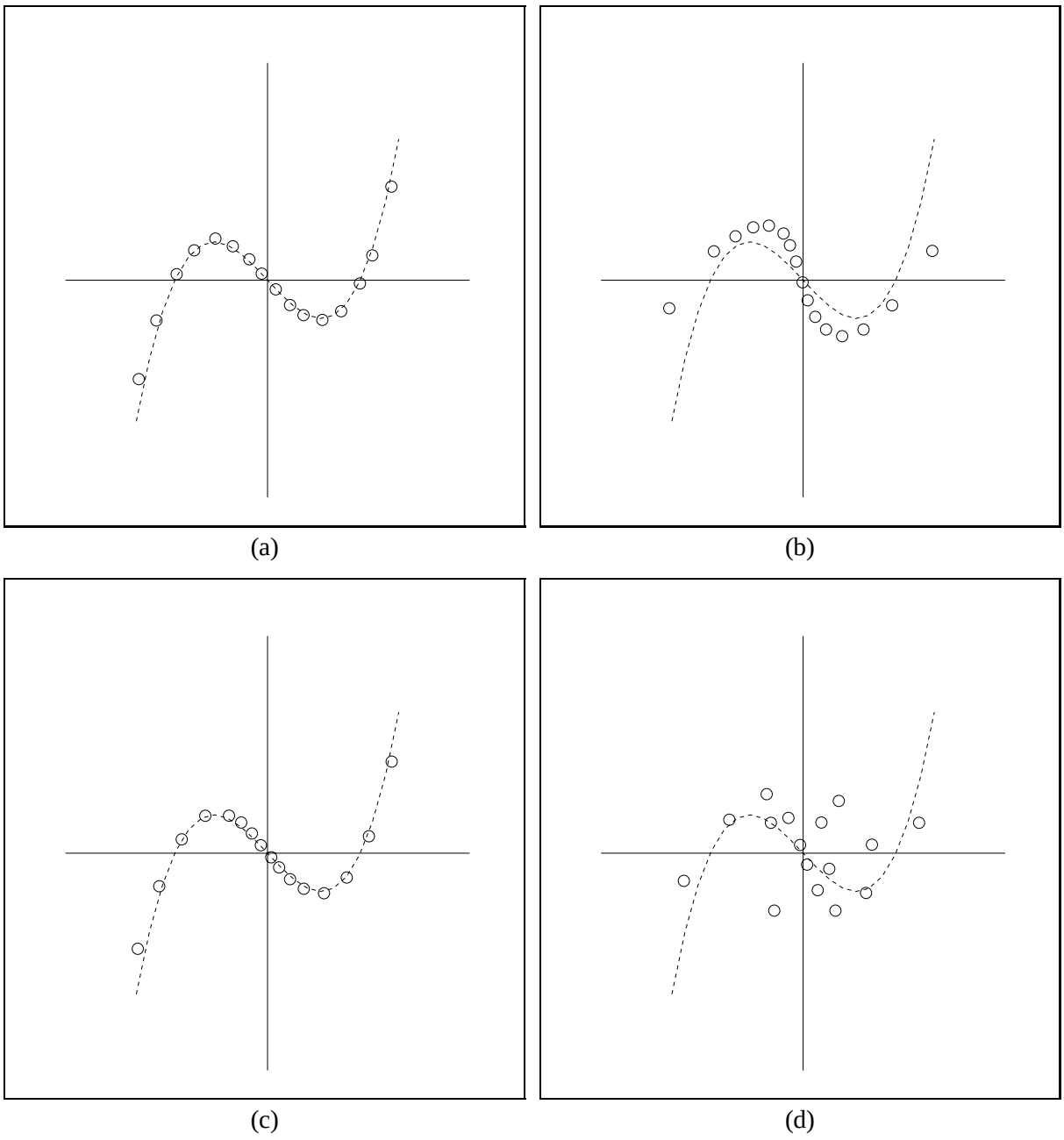
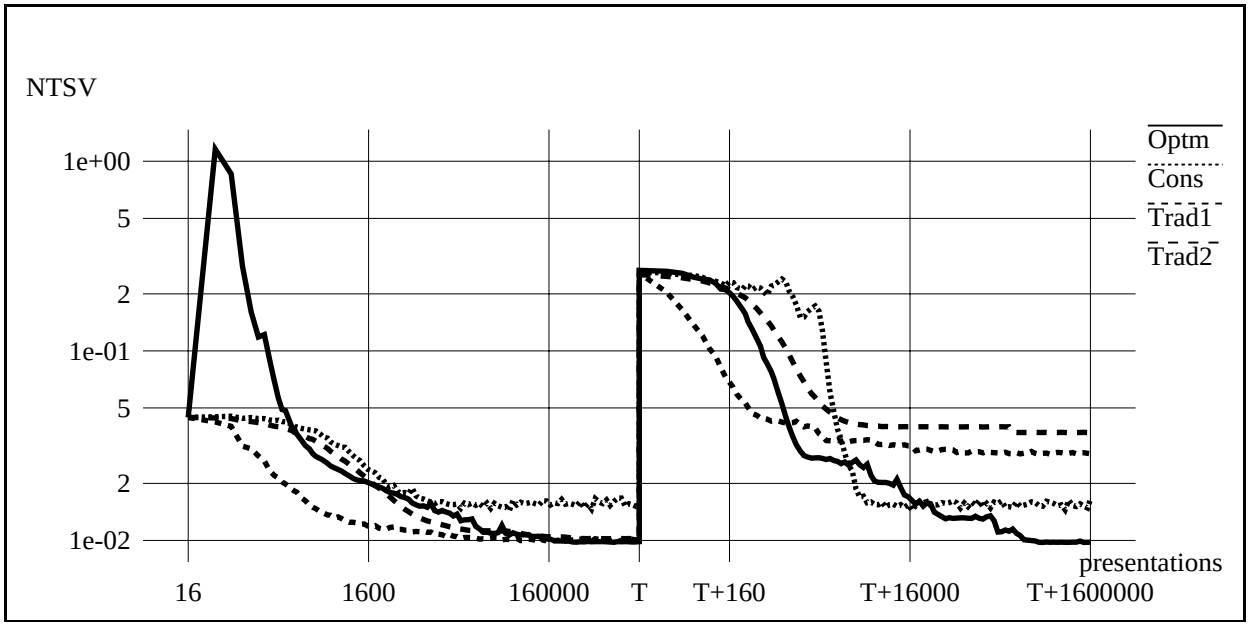
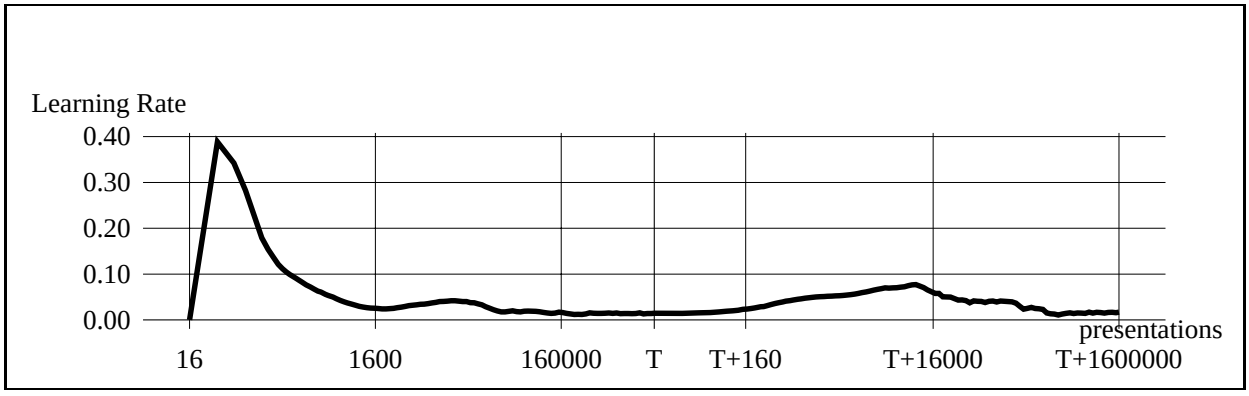


Figure 3.9: Locations of reference vectors for the rotational non-stationary problem for (a) *Optm*, (b) *Cons*, (c) *Trad1*, (d) and *Trad2*. The figures show the reference vector center locations after 160,000 presentations, i.e., after 8 full counter-clockwise rotations of the S-shaped curve.



(a)



(b)

Figure 3.10: Simulations of the four k-means algorithms on the problem where the training pattern distribution experiences an abrupt change: (a) the normalized mean square error (b) and the learning rate for the *Optm* algorithm.

level equivalent to that before the change. *Cons*, however, cannot achieve as low a *NTSV* as *Optm*, neither before nor after the change of the statistics, because its load balancing scheme distorts the *NTSV* cost function more severely. The learning rate of the *Optm* algorithm is shown Figure 3.10b. When the *Optm* algorithm first starts, the learning rate is large since the variance of the 16 normalized variations $\hat{v}_{k,norm}$ is large. As the algorithm approaches an optimal steady state, the variance decreases and the learning rate decreases correspondingly. At the moment right before the change of the input statistics, the learning rate is 0.015, and the variance and mean of the *normalized* variations $\hat{v}_{k,norm}$ is 2.7×10^{-4} and $1/16$, respectively. When the input statistics changes, the uniformity of the $\hat{v}_{k,nrom}$ is disturbed. The variance of $\hat{v}_{k,norm}$ increases gradually and thus results in an increasing learning rate. After 240 pattern presentations, i.e., after each region has been assigned an average of 15 new pattern vectors, the variance of $\hat{v}_{k,norm}$ increases to 4.3×10^{-4}, and the learning rate grows to 0.029, about twice its value before the change of the statistics.

3.6 Vector Quantization Coding of Image Data

In this section we evaluate the optimal k-means algorithm in a realistic application, specifically, vector quantization coding of a monochrome image. We have chosen this application in part because the computation is very intensive due to the large amount of data that must be processed. Furthermore, since its working principle is simple, it does not obscure the efficiency of the k-means algorithm under test. Finally, the evaluation results can be expressed both numerically and visually, providing additional intuitive understanding of the differences among various algorithms.

Vector quantization image coding is used to reduce the transmission bit rate or data storage requirements while maintaining an acceptable image quality. In this application, the images to be

encoded are decomposed into small blocks, say 4 by 4 pixels, called *vectors*. The resulting vectors are represented by the "nearest" of a limited set of prototype vectors, called *codewords*. The set of codewords used to represent an image, or a portion of an image, is called *codebook*.

The "LENA" image (Fig. 3.11) used in this section consists of 512 by 512 pixels, each digitized into 8-bit gray levels. Following [38], we partition the image into 4 quadrants, and encode each one separately with only 32 different codewords. To find the "best" 32 codewords for each quadrant, we use k-means clustering to partition the vectors observed in a given quadrant into 32 clusters, and then define the centroids of the resulting clusters as the codewords. In this experiment, each quadrant is decomposed into 4096 blocks of 4 by 4 pixels, each defining a 16 dimensional vector. The individual encoding of the 4 quadrants of the image allows us to study the adaptability of the optimal k-means algorithm when the source statistics is changing from one quadrant to the next.

Figure 3.11: The "LENA" image.

For the purpose of comparison, we test here the following four k-means algorithms on computing the codebooks for the LENA image:

- *Optm* : the proposed optimal k-means algorithm ($\alpha = 0.9999$).
- *Cons* : the adaptive k-means algorithm with the conscience learning rule [34], ($B = 0.0001$,[3] $\eta = 0.01$).
- *Trad* : the traditional adaptive ("on-line") k-means algorithm (η = 0.01).
- *LBG* : the classical batch k-means algorithm which is the generalized Lloyd clustering algorithm proposed by Linde, Buzo, and Gray [39].

We have included the *LBG* algorithm in this evaluation because it is a standard method used in image coding applications. Whereas *Optm*, *Cons* and *Trad* are adaptive *on-line* algorithms in which the parameters are updated after each pattern presentation, the *LBG* algorithm operates in *batch* mode where the parameters are updated only after each *iteration*, containing all the patterns in the training set.

To evaluate these four algorithms, we apply each of them to compute the codebooks for the upper-left, upper-right, lower-left, and lower-right quadrants, in that sequence. For the first (upper-left) quadrant, we initialize the 32 codewords to actual pattern vectors randomly drawn from this quadrant. We then continue by presenting to the algorithm 81920 patterns randomly selected from the upper-left quadrant. The 81920 pattern presentations correspond to 20 iterations of the presentation of all 4096 patterns in the quadrant. To compute the codebooks for the other three quadrants, we successively initialize the 32 codewords for the new quadrant with the codewords

[3]This is the value used by Desieno [34] and it is equivalent to $\alpha = 0.9999$.

generated from the previous one.

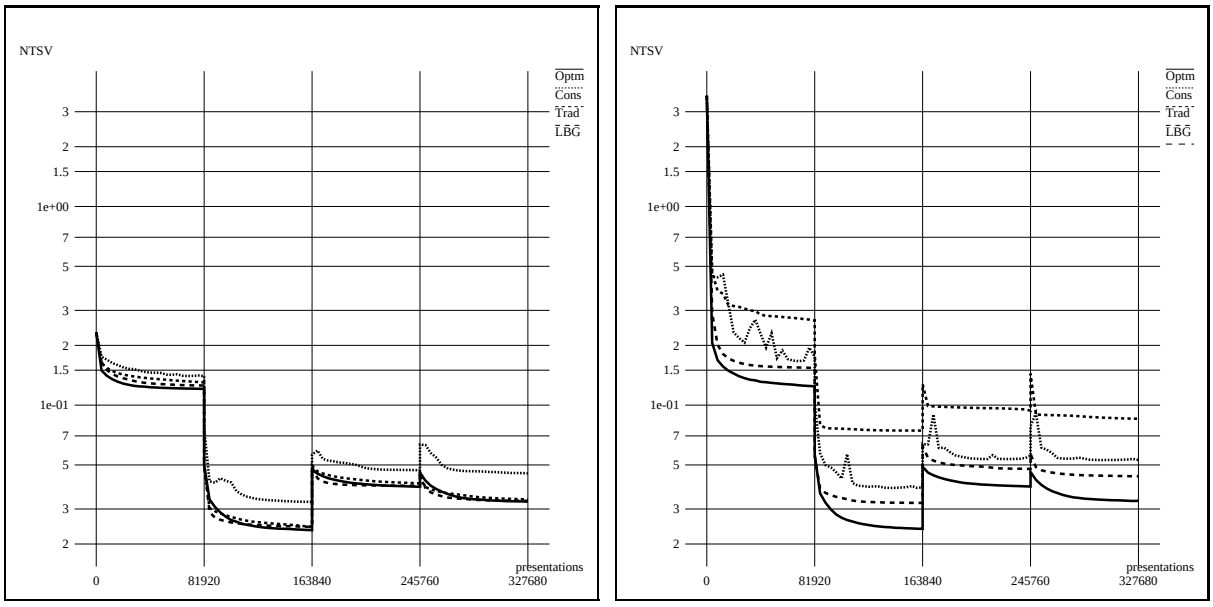


Figure 3.12: The normalized total spatial variation (*NTSV*) of the four k-means algorithms: (a) The initial codewords are assigned to pattern vectors randomly selected from the upper-left quadrant. (b) The initial codewords are assigned to uniformly distributed random locations in the pattern domain.

Figure 3.12a shows the normalized total spatial variation (*NTSV*) of the four k-means algorithms obtained from the above experiment. The results indicate that in steady state, *Optm* performs better than the other three algorithms; it outperforms *Trad* and *LBG* by a small amount, and *Cons* by a larger margin. In order to demonstrate the steady-state performance of these four algorithms visually, we display in figure 3.13 the images encoded by the codebooks generated in the above simulation. The visual quality of image 3.13a (encoded by *Optm*) appears to be slightly better in the areas of the shoulder, forehead and nose, than images 3.13b-d (encoded by *Cons*, *Trad* and *LBG*, respectively). This observation also corresponds to the fact that image 3.13a has the

SNR = 15.7094

(a)

SNR = 14.7047

(b)

SNR = 15.4853

(c)

SNR = 15.2332

(d)

Figure 3.13: The encoded image with initial codewords assigned to randomly selected patterns in the upper-left qudrant: (a) *Optm*, (b) *Cons*, (c) *Trad*, and (d) *LBG*.

higher signal-to-noise ratio (SNR) than the other three images, where SNR is defined as:

$$SNR = 10 \log \left(\frac{255^2}{TSV} \right) \text{ dB}. \tag{3.10}$$

With respect to the dynamic performance, figure 3.12a reveals that the *LBG* algorithm adapts somewhat faster than the *Optm* algorithm after an abrupt change such as switching from one quadrant to the next; the *NTSV* of the *LBG* algorithm drops slightly faster than that of the *Optm* algorithm after the switching of each training set. This is a consequence of the batch operation of *LBG* which tends to have a shorter effective memory of the initial codewords than an adaptive algorithm such as *Optm*. The learning rate of the three "on-line" algorithms could be increased to show a similarly fast response, but at the cost of some increase in the final steady-state error. However, for an application such as image coding, we are primarily interested in the steady-state performance since the clustering process is terminated when the error starts to show insignificant reduction with each new iteration. In other applications where the problem statistics are non-stationary and unknown in advance, *Optm* is more appropriate than *LBG*, because it can track the statistics changes constantly and automatically without resorting to an external agent to re-initiate the training process.

Figure 3.12b shows the *NTSV* resulting from a simulation similar to the first one. However, the starting codewords in this case are chosen from uniformly distributed random locations in the pattern domain of $[0, 255]^{16}$. This initialization method is inefficient since some of the initial codewords may lie in regions with no pattern, and thus may results in some unused codewords. By comparing the results of this simulation with that of the first simulation, we can study the robustness of each algorithm to the selection of the initial codewords.

Comparing the *NTSV*s of the four algorithms in the second simulation, we see that the *NTSV* for the *Optm* algorithm is always lower than for the other three algorithms. At steady state, the *NTSV* of *Optm* is about 80% of that generated by *LBG*, which is the second best. Thus the *Optm* algorithm clearly outperforms the other three algorithms when the starting codewords are poorly initialized. When comparing the *NTSV* in this figure with that in figure 3.12a, we see that only the *Optm* algorithm maintains near-optimal performance in spite of the inefficient initial codewords.

Figure 3.14 shows the encoded images using the codewords generated from the second simulation. Comparing image 3.14a with image 3.13a, we find that the visual quality of both images is about the same. The SNR of image 3.14a is 99.5% of that of image 3.13a. This demonstrates that *Optm* can find good codewords even with inefficient initialization.

Comparing images 3.13b-d with images 3.14b-d, we find obvious degradation of the visual quality around the shoulders, faces, and the hats in images 3.14b-d. Especially, image 3.14c obtained with *Trad* exhibits annoying edges and shading steps in many area. It also has the lowest SNR of all the images corresponding to about 75% of the SNR for *Optm*. Images 3.14b and 3.14d exhibit about 89% to 93% of the SNR for *Optm*. This degradation indicates that the *Cons*, and *LBG*, and in particular the *Trad* algorithms will perform far from optimal if they are inappropriately initialized.

3.7 Summary

The optimality criterion for this new algorithm is derived from the assumption that the distribution p is smooth and K is large. Even though this is not true for many practical cases, our simulation results show that the algorithm has better dynamic and static performances than other

SNR = 15.6310

(a)

SNR = 13.8924

(b)

SNR = 11.7090

(c)

SNR = 14.5800

(d)

Figure 3.14: The encoded images with initial codewords assigned to uniformly distributed random locations in the pattern domain: (a) *Optm*, (b) *Cons*, (c) *Trad*, and (d) *LBG*.

k-means variants. The superior performance is attributed to two novel mechanisms employed. The first one guides the partition towards an optimal solution. It allows the new algorithm to out-perform the traditional and the square-root k-means algorithms. Since the new biasing mechanism is capable of attaining the optimal partition for asymptotical large K, the resulting algorithm can also generate a partition with lower total spatial variation than the algorithm based on the conscience rule which simply equalizes the number of patterns in each region.

The second mechanism dynamically adjusts the learning rate and thereby makes it possible to learn very quickly initially, without sacrificing accuracy in approximating the final optimal solution. As the partition approaches an optimal solution, the learning rate decreases; this in turn allows the partition to move even closer to the optimal value. A small threshold value in the computed learning rate can be used as a simple stopping criterion for the adjustment of the cluster centers. However, it might be advisable to continue to monitor the input statistics by computing the entropy of the normalized within-region variations, so that the system can react automatically to any changes in input behavior and resume adjusting the reference vectors. Should the density distribution of the input patterns suddenly change, the resulting imbalance in the v_k's would quickly increase the learning rate, and the partition can be adjusted to the new situation with good response times.

Chapter 4

Heterogeneous Architecture Based on Error-Weighted K-Means Partitioning

In chapter 2, we have described the class of heterogeneous architectures that are based on k-means partitioning. When the traditional k-means algorithm is used to partition the input domain, only the input distribution is considered in the partitioning process. As a result, the representation power of the expert modules in the architecture is not fully utilized. In this chapter, we introduce an enhancement of the heterogeneous architectures which can fully utilize the representation power of all the expert modules and which is suitable for both stationary and non-stationary situations. The enhanced architecture is characterized by a novel k-means algorithm that integrates into its partitioning process information about the input distribution, the structure of the goal function and about the capabilities of the expert modules. The new k-means algorithm allows each individual region in the partition to adjust its size so that the representation resources in all the regions are optimally used.

4.1 Objective of the Input Partitioning

For the heterogeneous architecture described in section 2.3 to perform its task efficiently, its partitioning process should also take into account the *goal* $\vec{g}$ of the network. In function approximation, the aim of a connectionist network is to minimize the *mean squared error* between the goal function $\vec{g}$ and the network function $\vec{f}$ defined as:

$$MSE = \sum_{k=1}^{K} MSE_k \qquad \text{with} \qquad MSE_k = \int_{\mathcal{I}_k} p(\vec{x}) \| \vec{f}_k(\vec{x}) - \vec{g}_k(\vec{x}) \|^2 \, d\vec{x} \tag{4.1}$$

where MSE_k is the contribution to the *MSE* from region $\mathcal{I}_k$, and referred to as the *partial mean squared error* in $\mathcal{I}_k$. Quantity MSE_k can be expressed as the product of p_k and ϵ_k, where p_k is defined as $\int_{\mathcal{I}_k} p(\vec{x}) \, d\vec{x}$ and ϵ_k is defined as $1/p_k \int_{\mathcal{I}_k} p(\vec{x}) \| \vec{f}_k(\vec{x}) - \vec{g}_k(\vec{x}) \|^2 \, d\vec{x}$. Quantity p_k is the probability of $\vec{x}$ being in $\mathcal{I}_k$. It reflects the density of input points in $\mathcal{I}_k$. Quantity ϵ_k is the mean squared error in $\mathcal{I}_k$ and is defined by the conditional expected value of the squared difference between $\vec{f}$ and $\vec{g}$ for $\vec{x}$ in $\mathcal{I}_k$. It reflects the mismatch between $\vec{f}_k$ and $\vec{g}_k$ on $\mathcal{I}_k$. The fact that MSE_k is the product of p_k and ϵ_k suggests that the k-means algorithm should integrate into its process information about the input distribution as well as about the mismatch between $\vec{f}_k$ and $\vec{g}_k$. Since the traditional k-means algorithm divides an input domain based only on the distribution p, it produces a partition that is usually not optimal for a heterogeneous architecture to approximate a specific output function.

Two schemes for improving the capability of the k-means algorithm for the decomposition of the input domain have been proposed previously. The first scheme is based on the use of an *extended metric* [40, 41, 42] which attempts to equalize the output variations in the various regions.

The output variation in region $\mathcal{I}_k$ is defined as the square of the difference between $\vec{g}(\vec{x})$ and $<\vec{g}(\vec{x})>$ for all $\vec{x}$ in $\mathcal{I}_k$, where $<\vec{g}(\vec{x})>$ is the mean of $\vec{g}(\vec{x})$ in $\mathcal{I}_k$. One serious problem with this scheme is that it disregards the shape of the network function $\vec{f}$. To solve this problem, a second scheme, referred to as the *error-driven* k-means algorithm, has been developed [21]. In this scheme, the magnitude of the learning rate is scaled by the squared error between the network output $\vec{f}(\vec{x})$ and the target value $\vec{g}(\vec{x})$. This modulation of the learning rate attracts relatively more reference vectors into areas where the approximation error between $\vec{f}$ and $\vec{g}$ is high compared to areas with low approximation error. Since the density of the reference vectors in areas with high approximation error is higher than that in areas with low error, the regions in the areas with high approximation error are smaller than those in the areas with low error. The *error-driven* k-means algorithm thus allows a heterogeneous architecture to allocate more representation resources into areas where they are more effective in improving the desire fit. However, this scheme requires that the reference vectors $\vec{c}_k$ neighboring the winning subdomain be adjusted too; it thus increases the computational complexity, especially for high-dimensional cases. Furthermore, because the learning rate is modulated by $\{\vec{f}(\vec{x}) - \vec{g}(\vec{x})\}^2$, the adjustment term may be too large and may cause the adaptation process to become unstable if the output is not properly scaled with respect to the input.

In this chapter, we present a heterogeneous architecture with a modified version of the k-means algorithm that integrates into its partitioning process information about the distribution p and about the mismatch between $\vec{f}_k$ and $\vec{g}_k$. The new k-means algorithm is based on a weighted squared Euclidean distance measure that attempts to equalize the time-averaged squared error evenly among all the expert modules in a network. It requires less computation than the *error-driven* k-means

algorithm and is fast enough for computation in real time, since only the winning reference vector needs to be adjusted. In addition, the new k-means algorithm does not modulate its learning rate with the output error. It thus avoids the scaling sensitivity problem of the *error-driven* k-means algorithm.

4.2 Error-Weighted Deviation Measure

To improve the capabilities of the k-means algorithm in partitioning the input domain of a heterogneous architecture, we use the *error-weighted deviation measure*, where the deviation between $\vec{x}$ and reference vector $\vec{c}_k$ is defined as:

$$d(\vec{x}, \vec{c}_k) = \epsilon_k \|\vec{x} - \vec{c}_k\|^2. \tag{4.2}$$

When the squared Euclidean distance measure is used to determine the winning region, only the input distribution p is considered in determining the extent of each region. By weighting this deviation measure with the mean squared errors ϵ_k, we allow the k-means algorithm to incorporate both the input distribution p and the output mismatch ϵ_k into its partitioning process, thus enabling each individual region to adjust its size according to both considerations. With this error-weighted deviation measure, the k-means algorithm thus attempts to minimize the *total error-weighted variation*:

$$TEV = \sum_{k=1}^{K} \nu_k \qquad \text{with} \qquad \nu_k = \int_{\mathcal{I}_k} p(\vec{x}) \epsilon_k \|\vec{x} - \vec{c}_k\|^2 \, d\vec{x}, \tag{4.3}$$

where ν_k is an *error-weighted variation* in $\mathcal{I}_k$.

In the following, we will investigate the characteristics of an optimal partition that mini-

mizes the total error-weighted variation. The investigation will consider only the asymptotic case where K, the number of regions (or "clusters") in the partition, is very large. We start with Gersho's theorem [37], which states that for large K and a smooth underlying input distribution, all variations v_k must be the same for an optimal partition that minimizes the total spatial variation.

Let q be a real-valued function of the form:

$$q(\vec{x}) = \begin{cases} p(\vec{x})\epsilon_1/Q & \text{if } \vec{x} \in \mathcal{I}_1 \\ \vdots & \vdots \\ p(\vec{x})\epsilon_K/Q & \text{if } \vec{x} \in \mathcal{I}_K \end{cases} \tag{4.4}$$

where Q is a normalization factor so that the integral of q on $\mathcal{I}$ becomes one. This function q can be interpreted as a weighted probability function induced from p and ϵ. With this notation, the total error-weighted variation can be expressed as:

$$TEV = \sum_{k=1}^{K} \nu_k \qquad \text{with} \qquad \nu_k = Q \int_{\mathcal{I}_k} q(\vec{x}) \|\vec{x} - \vec{c}_k\|^2 \, d\vec{x}, \tag{4.5}$$

Let $\{\vec{c}_k^{\,*}\}$ be a set of reference vectors that minimizes the total error-weighted variation. The region $\mathcal{I}_k$ associated with $\vec{c}_k$ will be defined by the inequality:

$$\mathcal{I}_k = \{\, \vec{x} \;:\; \epsilon_k \|\vec{x} - \vec{c}_k^{\,*}\|^2 \leq \epsilon_i \|\vec{x} - \vec{c}_i^{\,*}\|^2, \quad \text{for each } i \neq k \,\}. \tag{4.6}$$

As K becomes large, the values of ϵ_i in the regions neighboring to $\mathcal{I}_k$ approach that of ϵ_k. The corresponding increase in uniformity of ϵ_i in the neighborhood of $\mathcal{I}_k$ reduces the bias of the error-weighted deviation measure, and results in a boundary of $\mathcal{I}_k$ that closely resembles that generated

based on the non-weighted squared distance measure. This interpretation together with equation 4.5 allows us to view an optimal partition that minimizes the total *error-weighted* variation with respect to p as an optimal partition that minimizes the total *spatial* variation with respect to q.

The increase in uniformity of ϵ_k also smoothens function q. For a smooth distribution p, function q can be assumed to be smoothly varying for asymptotically large K. Since the definition of *TEV* in equation 4.5 is the same as that of *TSV* in equation 3.2, applying Gersho's criterion to equation 4.5 results in the following statement: For asymptotically large K and smooth distribution p, all error-weighted variations ν_k must be the same for an optimal partition that minimize the total error-weighted variation. This theoretical result leads us to conjecture that even for small K, equalizing the variations ν_k might lead in a robust manner to near-optimal partitions. Our experimental results in section 4.4 support this conjecture.

4.3 K-Means Algorithm with Error-Weighted Deviation Measure

Even though the error-weighted deviation measure allows us to integrate into the k-means algorithm information about the input distribution and about the mismatch between $\vec{f}_k$ and $\vec{g}_k$, the algorithm still may have difficulties converging to an optimal or near-optimal configuration. As has been successfully demonstrated for unsupervised input partitioning in chapter 3 equalizing the spatial variations v_k in the partition enables the k-means algorithm to converge to an optimal partition. Following the scheme used in 3.3 for v_k-equalization, we define an *effective deviation* between $\vec{x}$ and reference vector $\vec{c}_k$ as

$$d(\vec{x}, \vec{c}_k) = \hat{\nu}_k \, \hat{\epsilon}_k \|\vec{x} - \vec{c}_k\|^2. \tag{4.7}$$

Multiplying the error-weighted deviation measure by $\hat{\nu}_k$ biases the membership indicator in favor of regions with smaller variations $\hat{\nu}_k$, and it leads to a robust equalization of these values in the various regions in the partition.

To obtain the estimates $\hat{\nu}_k$, we use the weighted running time-average:

$$\hat{\nu}_{k,\,T+1} = \alpha\, \hat{\nu}_{k,\,T} \;+\; (1-\alpha) M_k(\vec{x}_T)\, \hat{\epsilon}_k\, \|\vec{x}_T - \vec{c}_{k,T}\|^2, \tag{4.8}$$

where α is a constant slightly less than 1. This equation is the modification of equation 3.6 used for estimating v_k. To estimate ϵ_k, we use

$$\hat{\epsilon}_{k,\,T+1} = \hat{\epsilon}_{k,\,T} \;+\; (1-\alpha) M_k(\vec{x}_T)[\hat{\epsilon}_{k,\,T} \;+\; \{\vec{f}(\vec{x}_T) - \vec{g}(\vec{x}_T)\}^2], \tag{4.9}$$

This equation allows us to update $\hat{\epsilon}_k$ of the winning expert by adding into $\hat{\epsilon}_{k,\,T+1}$ new information about its approximation error, and maintain the previous value of $\hat{\epsilon}_k$ for other expert module. We start this estimation by initializing all $\hat{\nu}_{k,0}$ and $\hat{\epsilon}_{k,0}$ to some small number, allowing the effect of the initialization to disappear quickly, so that the estimates are soon dominated by the actual data seen by each region.

In order for our modified k-means algorithm to attain both adaptation speed and approximation accuracy, we adjust the learning rate of the algorithm dynamically, based on the *entropy* of the estimated variations $\hat{\nu}_k$ 3.4. According to this entropy measure, the learning rate (η_{km}) is defined to be:

$$\eta_{km} = 1 - H(\hat{\nu}_1, ..., \hat{\nu}_K)/ln(K), \tag{4.10}$$

where $H(\hat{\nu}_1, ..., \hat{\nu}_K) = \sum_{k=1}^{K} -\hat{\nu}_{k,norm}\; ln(\hat{\nu}_{k,norm})$ with $\hat{\nu}_{k,norm} = \hat{\nu}_k/(\sum_{i=1}^{K} \hat{\nu}_i)$. This dynamic

adjustment of the learning rate, together with the mechanism for equalizing the variations, has experimentally shown to improve the performance of the k-means algorithm, especially in the cases of non-stationary input statistics 3.5.2 These mechanisms are used again to improve the performance of the new k-means algorithm.

In the course of adaptation, the locations of the reference vectors $\vec{c}_k$ are constantly varying. When these reference vectors change, the boundaries of the input domains will change accordingly, resulting in non-stationary training data for each expert module. To assist an expert module to cope with this non-stationary situation, we correlate the learning rate η_{lms} of the LMS algorithm with the learning rate η_{km} of the k-means algorithm by setting it to:

$$\eta_{lms} = \eta_{km} + \kappa. \tag{4.11}$$

The first term, η_{km}, is the learning rate of the k-means algorithm defined by equation 4.10. It allows us to vary the learning rate of an expert module according to the stationarity of the input domain of the expert module. The second term, κ, is a constant used to prevent η_{lms} from becoming too small. Since the adaptation of the expert module cannot be be completed before the input domain of the expert module has been stabilized, the constant κ allows the expert module to complete its adaptation after its input domain has reached stationary state ($\eta_{km} \rightarrow 0$).

4.4 Empirical Demonstration

This section presents an empirical demonstration of the enhanced heterogeneous architectures introduced in this chapter. It illustrates the performance of this new architecture through a

system that implements piecewise linear approximation of the goal function; the expert modules in the system are restricted to be linear functions trained by the *least mean square* (*LMS*) algorithm [31]. The following four versions of the heterogeneous architecture based on k-means partitioning are compared:

- *Ewgt*: the *Ewgt* architecture uses the k-means algorithm based on the error-weighted deviation measure and the *LMS* algorithm with $\eta_{lms} = \eta_{km} + 0.01$. The parameters of the k-means algorithm are set as follows: $\alpha = 0.9999$, $\hat{w}_{k,0} = 10^{-10}$ and $\hat{\epsilon}_{k,0} = 10^{-10}$.

- *Vwgt*: the *Vwgt* architecture uses the k-means algorithm based on the *variation*-weighted squared distance measure [43] and the *LMS* algorithm with $\eta_{lms} = \eta_{km} + 0.01$. This k-means algorithm partitions the input domain based *only* on the input distribution. Its parameters are set as follows: $\alpha = 0.9999$ and $\hat{w}_{k,0} = 10^{-10}$.

- *Extd*: the *Extd* architecture uses the k-means algorithm based on the *extended* Euclidean metric [40, 41, 42] and the *LMS* algorithm with $\eta_{lms} = 0.01$. The learning rate η_{km} of the k-means algorithm is set to 0.01.

- *Errd*: the *Errd* architecture uses the *error*-driven k-means algorithm [21] and the *LMS* algorithm with $\eta_{lms} = 0.01$. The learning rate η_{km} of the k-means algorithm is set to 0.01.

In this evaluation, we first demonstrate the performance of the above four architectures on two tutorial 1-dimensional approximation problems. We then examine these architectures on a more challenging 4-dimensional problem: the prediction of the Mackey-Glass time series [44, 45].

Figure 4.1 illustrates the performance of the above four architectures on the first one-dimensional problem. The goal function, shown in figure 4.1a, is defined as:

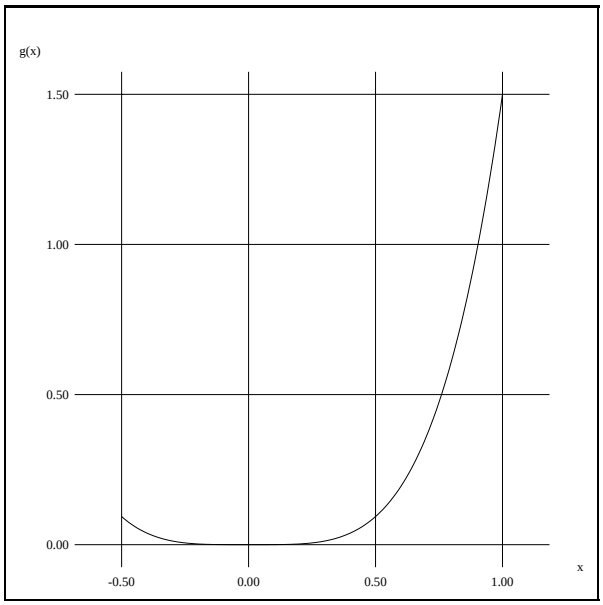
$$g(x) = 1.5x^4 \qquad \text{with} \qquad x \in [-0.5, 1.0]. \tag{4.12}$$

For this demonstration, we divide the input domain into two subdomains so that a partition is characterized by a single boundary point X_B. The overall approximation error of the heterogeneous architecture for a given boundary point X_B is measured by the *normalized mean squared error* (*NMSE*), defined as the mean squared error between f and g normalized by the mean squared value of g. Figure 4.1b shows the *NMSE* of the above four architectures on an input-output sequence generated according to (4.12). Each curve is the average of five runs with different pattern sequences. For each architecture, we initialize all c_k of the k-means algorithm to 0.5 and all the parameters of expert module f_k to 10^{-10}. The simulation results indicate that the final error generated by *Ewgt* is about one fifth of those generated by *Vwgt*, *Extd*, and *Errd*. Notice that the *NMSE* of *Errd* appears to increase after 20,000 presentations, indicating a conflict between the goals of minimizing the mean squared error and of evenly partitioning the input domain.

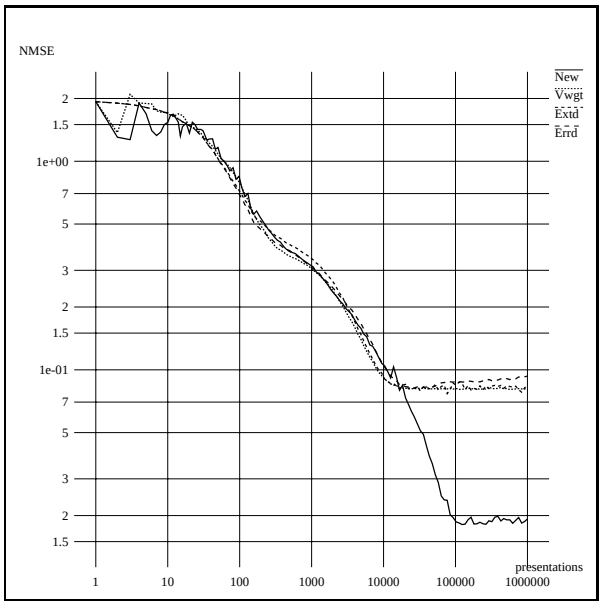
Figure 4.1c shows the calculated *NMSE* as a function of the partitioning point X_B. It indicates that the optimum performance is achieved when the partitioning point is at $X_B = 0.60$ and the resultant *NMSE* is equal to 0.015. Our results show that the partition produced by *Ewgt* ($X_B = 0.57$) comes much closer to the optimum partition than the other three algorithms.

To evaluate scaling sensitivity, we have also applied the four algorithms to a function:

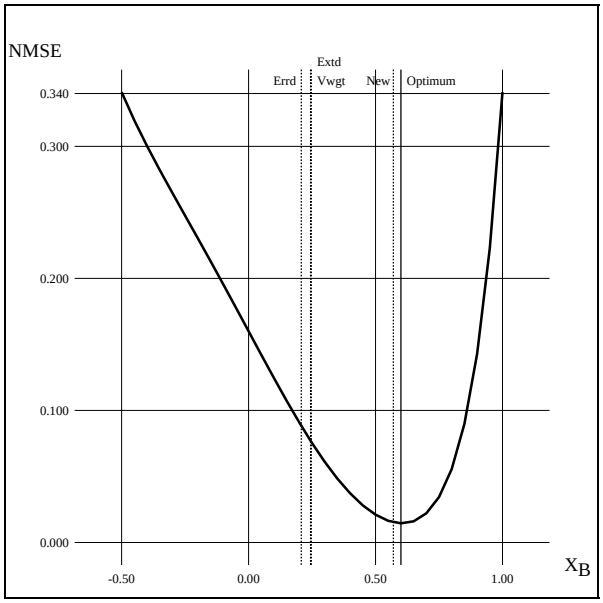
$$G(x) = 150x^4 \qquad \text{with} \qquad x \in [-0.5, 1.0]. \tag{4.13}$$



(a)



(b)



(c)

Figure 4.1: Performance comparison of the four heterogeneous architectures on a 1-dimensional quadratic problem.
(a) The goal function: $g(x) = 1.5x^4$.
(b) The *NMSE* vs. the number of pattern presentations.
(c) The *NMSE* vs. the partitioning point X_B. Vertical lines mark the partitioning points for various architectures.

The performance of *Ewgt*, *Vwgt* and *Extd* are similar to their performance on $g(x)$, indicating that the scaling of the output with respect to the input does not affect their stability. The simulation of *Errd* could not be completed since it became unstable due to the scaling sensitivity of this algorithm.

As a more challenging test of the new algorithm, we use the Mackey-Glass time series prediction task [45]. This time series is obtained by integrating the Mackey-Glass differential-delay equation [44]:

$$\frac{dx[t]}{dt} = \frac{ax[t-\tau]}{1+x^{10}[t-\tau]} - bx[t], \tag{4.14}$$

where a, b and τ are defined to be 0.2, 0.1 and 17 respectively. In this problem, we use the 4-dimensional input vector $\vec{x}_i = (x[i], x[i-6], x[i-12], x[i-18])$ to predict the output $y_i = x[i+85]$, a value of the time series in the future. We train each of the heterogeneous architectures on a sequence of input-output patterns randomly selected from a training set of 10,000 input-output pairs. We evaluate each system on a test set containing 500 input-output pairs that differ from those in the training set. We again measure the performance using the normalized mean squared error (NMSE) on the test set, defined as the mean squared error on the test set normalized by the mean squared value of the time series.

Figure 4.2 compares the performance of the above four architectures on the Mackey-Glass problem. We test these architectures on three instances where the input domain is divided into 4, 8, and 16 regions respectively. For each instance, we initialize all the reference vectors $\vec{c}_k$ to 0.93, the mean of the time series, and all the parameters in expert module $\vec{f}_k$ to be 10^{-10}.

Figure 4.2a shows the *NMSE* of the four architectures where the input domain is partitioned into 4 regions. Each curve here is again the average of five runs with different pattern sequences. The simulation results indicate that all four architectures have comparable performance with respect

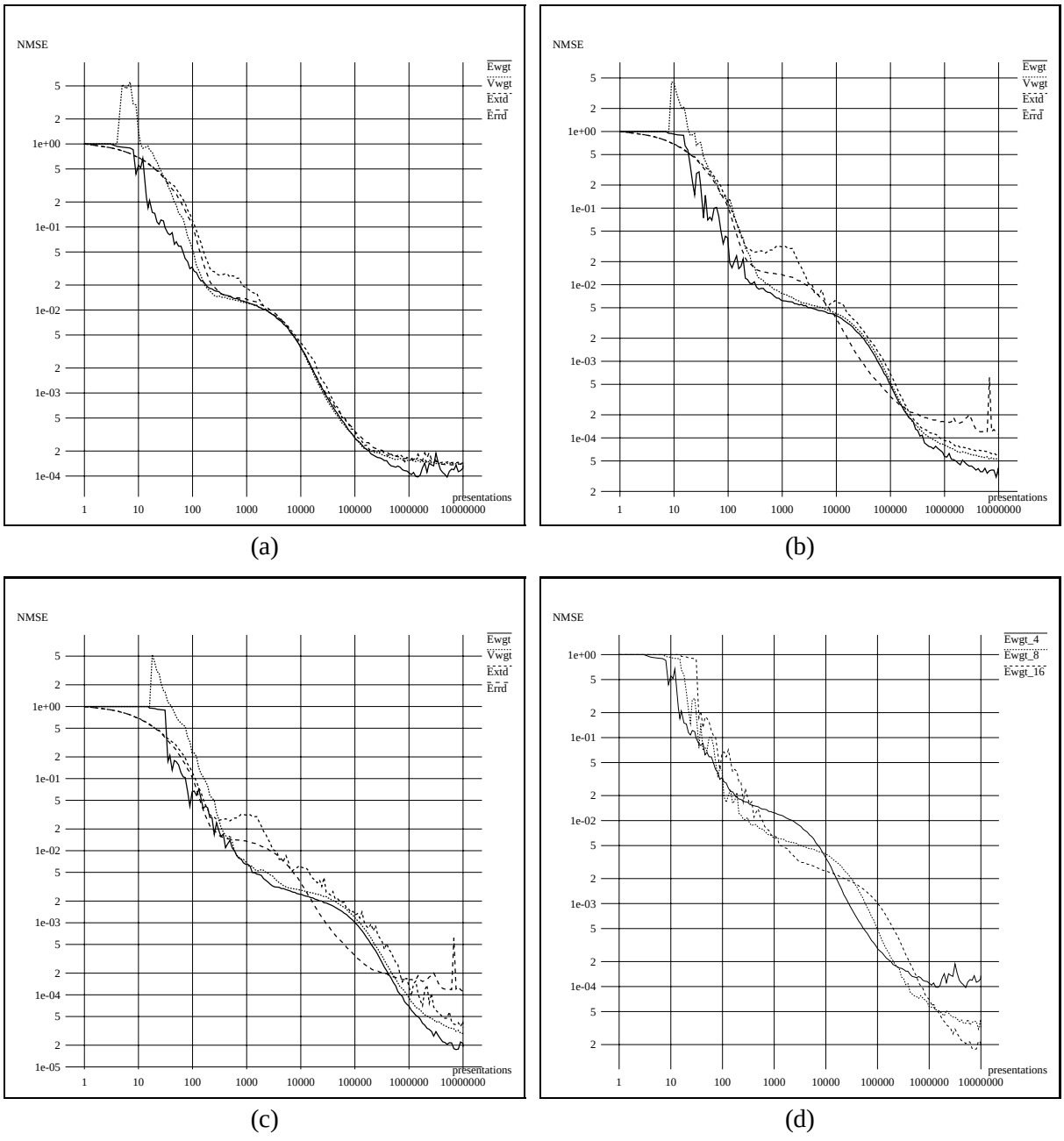


Figure 4.2: Performance comparison on the Mackey-Glass time series prediction.
(a) The input domain is partitioned into 4 regions. (b) The input domain is partitioned into 8 regions. (c) The input domain is partitioned into 16 regions. (d) Comparison of the *NMSE* of *Ewgt* from a, b and c.

to both convergence rates and the steady state accuracy.

Figure 4.2b shows the *NMSE* of the four architectures where the input domain is partitioned into 8 regions. The simulations indicate that the *Ewgt* algorithm has better generalization abilities. The *NMSE* of *Ewgt* at the steady state is about six tenths that of *Vwgt*, about half that of *Extd*, and about three tenths that of *Errd*. Note that the *NMSE* of *Errd* fluctuates by a large degree even when the algorithm reaches the steady state. With respect to learning speed, the *Ewgt* algorithm has a faster convergence rate than *Vwgt* and *Extd*; the *NMSE* of *Ewgt* always stays below those of *Vwgt* and *Extd* except for the first few pattern presentations. Comparing *Ewgt* to *Errd*, we find that the NMSE of *Ewgt* decreases faster than that of *Errd* at both the initial and final stages.

Figure 4.2c shows the *NMSE* of the four architectures where the input domain is partitioned into 16 regions. The *relative* shapes of the *NMSE* curves in this figure are resembling those in figure 4.2b, indicating that the four architectures have similar relative behaviors.

The simulations in all these three instances indicate that *Ewgt* has better performance. It can achieve lower steady state *NMSE* compared to the other three architectures. The steady state *NMSE*s of both *Ewgt* and *Vwgt* are lower than those of *Extd* and *Errd*, indicating that both *Ewgt* and *Vwgt* can partition the input domain more effectively. Also, since *Ewgt* takes into account the output information of the goal function, it can achieve better accuracy than *Vwgt*.

Figure 4.2d compares the *NMSE*s of *Ewgt* from figure 4.2a, 4.2b and 4.2c. It indicates that the more regions in the partition, the slower a convergence rate but the higher the steady state accuracy. The same behavior is also observed for *Vwgt* and *Extd*. However, for *Errd*, we find that both of its *NMSE* curves in figure 4.2c and 4.2b are identical. Further investigation reveals that 8 out of 16 regions cover no data points. As a result, only 8 out of 16 expert modules are utilized. All

these evidences indicates that *Ewgt*, *Vwgt* and *Extd* scale quite well while *Errd* scales poorly with respect to the number of regions in the partition.

4.5 Summary

This chapter has introduced an enhanced version of the heterogeneous architecture based on k-means partitioning. The enhanced architecture is based on a modified the k-means algorithm that partitions the input domain by integrating into its process information about the input distribution *and* about the mismatch between the network function and the goal function. The new k-means algorithm uses an error-weighted deviation measure that aims at equalizing the average approximation errors in all regions of the partition. This scheme of equalizing the errors in the different regions has several advantages. It is simple enough to be performed in real-time. It does not require a critical scaling of the output with respect to the input. Moreover, since the error weighting factor represents the key goal of the algorithm most directly, the new k-means algorithm out-performs other k-means algorithms with different innovative but more adhoc approaches.

The scheme of equalizing the approximation errors in all regions of the partition is based on the assumption that a distribution $p(\vec{x})$ is smooth and K is large. For the case of small K and non-smooth distribution, equalizing the approximation errors might distort the cost function that the k-means algorithm attempts to minimize. However, this error-equalization allows the new k-means algorithm to fully utilize the given resources robustly, and thus offsets the disadvantage caused by the distortion of the goal function. The equalization of the approximation errors therefore permits the new k-means algorithm to partition the input domian more effectively. As evident from our demonstration on the Mackey-Glass problem, the larger number of cluster, the better the heteroge-

neous architecture with the new k-means algorithm performs compared to other architecture. For such problems, with a larger number of clusters, the asymptotic assumptions underlying our scheme are approximated even better, and should thus result in partitions that lie very close to the optimum.

A key insight gained in this work is that the k-means algorithm is not restricted to unsupervised learning tasks; deviations from a desired goal function can be used to bias the partition in such a way that the overall error is reduced. Finally, the scheme for integrating the mismatch between the network function and the goal function presented in this paper is not limited to the described multi-module architecture but can also be applied to other supervised learning architectures that use the k-means algorithm.

Chapter 5

Performance Evaluation of Heterogeneous Architectures

5.1 Scope of Evaluation

In chapter 1 we have argued that traditional artificial neural network architectures are not sufficient to cope with large complex problems, and that heterogeneous architectures are needed to solve such problems. In this chapter and the two that follow, we support the above argument with a comparative analysis of the *performance* and *complexity* of heterogeneous architectures and traditional architectures for large, complex problems. The heterogeneous architectures analyzed in this study are restricted to those based on error-weighted k-means partitioning, as introduced in chapter 4.

In this chapter we compare the performance of a heterogeneous architecture that implements a piecewise linear function *(Het)*, against that of traditional architectures on the Mackey-

Glass time series prediction [45], and on a hand-written character recognition task [46]. These two problems are considered by a number of connectionist researchers [45, 29, 17, 47, 48, 46] to be benchmarks for evaluating the approximation and classification capabilities of artificial neural networks.

For the Mackey-Glass problem, where the input dimension is quite low, we compare the *Het* architecture with the following 4 traditional architectures:

- *RBF* : the radial basis function architecture [29];
- *Tbl* : the architecture based on a lookup table approach;
- *Loc* : the architecture based on a local model approach [24, 25];
- and *BP* : the back-propagation architecture, which is a multilayer perceptron trained with the on-line back-propagation algorithm [2].

The *RBF*, *Tbl* and *Loc* architectures have been chosen to represent traditional, homogeneous architectures composed of local-support basic functions. The *BP* architecture represents traditional, homogeneous architectures composed of global-support basic functions. The performance of *Het* with these four architectures on the Mackey-Glass problem is described in section 5.3. In section 5.4, the *Het*, *RBF*, and *BP* architectures are evaluated on the hand-written capital letter recognition problem. The results from these two evaluations are then discussed in section 5.5.

An analysis of serial and parallel implementations of the heterogeneous architectures are given in chapter 6 and 7, respectively. Since we are interested in artificial neural network architectures which are simple to implement with parallel VLSI hardware, we compare the implementations of *Het* to only those of *RBF* and *BP*. We have chosen these two types of architectures because of

their popularity and suitability for dedicated hardware implementation.

5.2 Review of Architectures under Test

This section gives a brief review of the following three architectures:

- *Het* : the heterogeneous architecture based on error-weighted k-means partitioning,
- *RBF* : the radial basis function architecture,
- *Tbl* : the architecture based on a lookup table approach,
- *Loc* : the architecture based on a local model approach,
- and *BP* : the back-propagation architecture.

5.2.1 Heterogeneous Architecture

Figure 5.1 depicts the schematic diagram of a heterogeneous architecture based on error-weighted k-means partitioning (*Het*). For an architecture that represents a piecewise linear mapping from R^M to R^N, its network function is of the following form:

$$\vec{f}(\vec{x}) = \sum_{i=1}^{K} M_k(\vec{x}) \vec{f}_k(\vec{x}) \tag{5.1}$$

where K is the number of expert modules. The membership indicator M_k is defined as:

$$M_k(\vec{x}) = \begin{cases} 1 & \text{if} \quad \hat{\nu}_k \hat{\epsilon}_k \|\vec{x} - \vec{c}_k\|^2 \ \leq \ \hat{\nu}_i \hat{\epsilon}_i \|\vec{x} - \vec{c}_i\|^2 \quad \text{for each } i \neq k \\ 0 & \text{otherwise} \end{cases} . \tag{5.2}$$

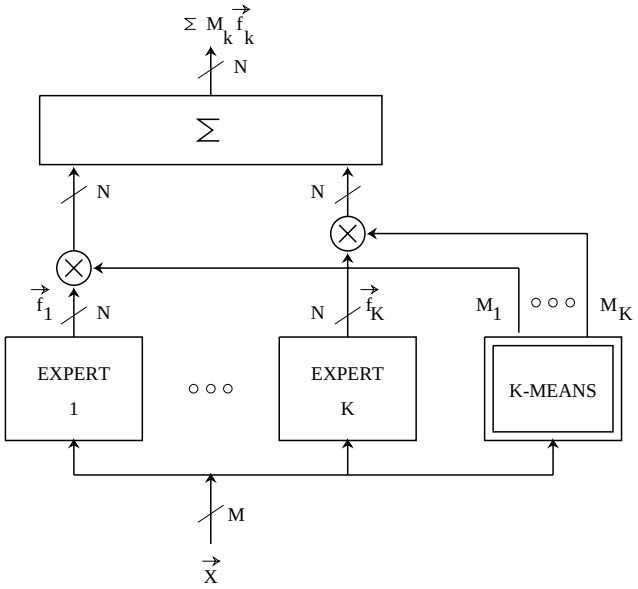


Figure 5.1: The schematic diagram of a heterogeneous architecture based on error-weighted k-means partitioning.

In the event of more than one minimum $\hat{\nu}_k \hat{\epsilon}_k \|\vec{x} - \vec{c}_k\|$, we set the M_k with the lowest index k to 1 and the others to 0. The membership indicators M_1,...,M_K are determined using the k-means algorithm based on error-weighted deviaiton measure described in section 4.3. Each expert module implements a linear function $\vec{f}_k$ of the form giving below:

$$f_{k,i}(\vec{x}) = w_{k,i0} + \sum_{j=1}^{M} w_{k,ij} x_j, \qquad \text{for } 1 \leq i \leq N. \tag{5.3}$$

Function $f_{k,i}$ represents the i-th component of $\vec{f}_k$. The parameters $w_{k,i0}$, ... , $w_{k,iM}$ of $f_{k,i}$ are adjusted using the *least mean square* (*LMS*) algorithm [31]. For the experiments in this chapter, we set the parameter α in the partitioning module of the *Het* architecture to be 0.9999. We initialize its $\hat{\nu}_k$ and $\hat{\epsilon}_k$ to 10^{-10}. We also set the learning rate η_{lms} of the *LMS* algorithm to $\eta_{km} + 0.01$ and initialize $w_{k,ij}$ to 10^{-10}.

5.2.2 Radial Basis Function Architecture

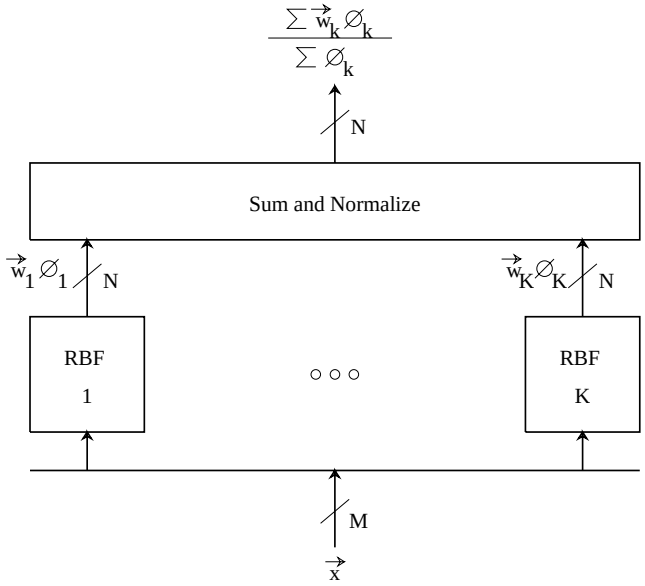


Figure 5.2: The schematic diagram of the radial basis function architecture.

The classical radial basis function architecture is a connectionist architecture based on basic functions with local supports. For an architecture that implements a function from R^M to R^N, as shown in figure 5.2, its network function is defined to be

$$\vec{f}(\vec{x}) = \sum_{k=1}^{K} \vec{w}_k \phi_k, \qquad \text{with} \qquad \phi_k = \phi(\|\vec{x} - \vec{c}_k\|), \tag{5.4}$$

where ϕ represents a radial basis function and K is the number of the radial basis functions in the architecture. The typical form of ϕ_k is a Gaussian function, which is defined as:

$$\phi_k = \phi(\|\vec{x} - \vec{c}_k\|) = \exp(-\|\vec{x} - \vec{c}_k\|^2/\sigma^2), \tag{5.5}$$

where σ is a parameter defining the width of the Gaussian function.

Several radial basis function architectures [3, 4, 5, 29, 7] have been developed. In this comparison, we primarily rely on the architecture described by Moody and Darken [29]. The network function in this architecture is given by:

$$\vec{f}(\vec{x}) = \frac{\sum_{k=1}^{K} \vec{w}_k \phi_k}{\sum_{k=1}^{K} \phi_k}. \tag{5.6}$$

This normalized form of the network function generalizes better than the one defined by equation 5.4. The training algorithm used by Moody and Darken [29] for determining the parameters $\vec{c}_k$, σ and $\vec{w}_k$ is divided into 3 independent successive stages:

- Place all the Gaussian centers $\vec{c}_k$ using an adaptive k-means algorithm. In order to improve the allocation of the Gaussian centers, we use in this investigation the adaptive k-means algorithm based on the variation-weighted deviation measure instead of the traditional adaptive k-means algorithm.

- Define the width σ of all the Gaussian functions using the *global nearest neighbor* rule [29]:

$$\sigma = \{1/K \sum_{k=1}^{K} \|\vec{c}_k - \vec{c}_{k,nearest}\|^2\}^{1/2}, \tag{5.7}$$

  where $\vec{c}_{k,nearest}$ is the nearest Gaussian center $\vec{c}_i$ to $\vec{c}_k$.

- Determine the heights $\vec{w}_k$ of all the Gaussian functions using the *LMS* adaptation rule [31].

For the experiments in this chapter, we set the parameter α in the k-means algorithm to be 0.9999, and initialize $\hat{v}_k$ to 10^{-10}. We set the learning rate η_{lms} of the *LMS* algorithm to be 0.01, and initialize all components of height $\vec{w}_k$ to 10^{-10}.

5.2.3 Architecture based on a Lookup Table Approach

The architecture based on a lookup table approach uses a regular grid of local support basic functions. In this architecture, the input domain is partitioned into disjoint cells: $\mathcal{I}_1$, ... , $\mathcal{I}_K$. Associated with cell $\mathcal{I}_k$ is parameter $\vec{z}_k$ that represents the output of the cell. For a given input $\vec{x}$, the system generates an output of the form:

$$\vec{f}(\vec{x}) = \sum_{k=1}^{K} \vec{z}_k M_k(\vec{x}), \tag{5.8}$$

where M_k is a membership function defined to be 1 if $\vec{x}$ belongs to $\mathcal{I}_k$ and 0 otherwise. The value of $\vec{f}(\vec{x})$ is thus equal to $\vec{z}_k$, where $M_k(\vec{x})$ = 1. To train the system so that the mean squared error (*MSE*) between $\vec{f}$ and the goal function $\vec{g}$ is minimized, the following recursive equations are used:

$$\vec{z}_{k,T+1} = \vec{z}_{k,T} + M_k(\vec{x}_T)\{\vec{g}(\vec{x}_T) - \vec{z}_{k,T})/(n_{k,T} + 1)\} \tag{5.9}$$

$$n_{k,T+1} = n_{k,T} + M_k(\vec{x}_T) \tag{5.10}$$

where n_k is the number of data points seen by cell k, and where $\vec{z}_{k,0}$ and $n_{k,0}$ are set to 0.

The architecture based on a lookup table approach is suitable only for problems with a low dimensional input domain since its hardware scales exponentially with the number of input dimensions. This type of architecture can learn very rapidly but it generalizes poorly because it normally has too many degree of freedom. Thus, comparing *Het* with this architecture is a good tool to investigate how much speed the heterogeneous architecture has to sacrifice in exchange for its generalization power.

5.2.4 Architecture based on a Local Model Approach

The architecture based on a local model approach [24, 25] has successfully addressed many approximation problems of low input dimensions. In this architecture, all the incoming training samples are stored and used as reference vectors $\vec{c}_k$. Associated with each reference vector is an *influence* function φ_k, whose value is defined to be 1 at $\vec{c}_k$ and is gradually vanishing with distance from the sample position. A typical form of φ_k is a Gaussian function, defined as:

$$\varphi_k(\vec{x}) = \exp(\|\vec{x} - \vec{c}_k\|^2/\sigma_k^2), \tag{5.11}$$

and σ^2 is a parameter defining the width of the Gaussian function.

Associated with each reference vector is also a local model f_k, assumed to be linear for this study. The coefficients of f_k are determined using the weighted least squares fit among all the training samples; the squared difference between the target output and the value of f_k at each sample point is weighted by the value of the influence function at that point. To generate the output, we combine the local models f_k of various training samples according to the equation:

$$f(\vec{x}) = \frac{\sum_{k=1}^{K} \varphi_k(\vec{x}) f_k(\vec{x})}{\sum_{k=1}^{K} \varphi_k(\vec{x})}. \tag{5.12}$$

where K is the number of stored training samples in the architecture.

For the experiment in this chapter, the batch version of the local model architecture is used. The sample points are organized using the *bumptree* data structure [25]. This allows us to quickly prune away training samples whose influence values are insignificant, i.e., less than a pre-determined threshold, thus speeding up the computation of f for a given input $\vec{x}$,

5.2.5 Back-Propagation Architecture

The back-propagation architecture is probably the most popular artificial neural network architecture. In this architecture, a multi-layer perceptron, which is a network with global-support basic functions, is trained with the *back-propagation* algorithm [2], which is a supervised learning procedure based on gradient descent. The goal of the back-propagation learning algorithm is to find a set of network parameters that minimizes the mean squared error between the network function $\vec{f}$ and the goal function $\vec{g}$. For the experiments performed in this chapter, we use the on-line version of the back-propagation algorithm and set its learning rate to 0.01. Figure 5.3a shows the back-propagation network used in the Mackey-Glass problem. The network has two hidden layers of perceptrons, each consisting of 20 sigmoidal units, and it represents a mapping from R^4 to R of the form:

$$z_{1,k}(\vec{x}) = 1 + \exp\{-w_{1,k0} - \sum_{i=1}^{4} w_{1,ki}x_i\}, \qquad \text{for } 1 \leq k \leq 20 \tag{5.13}$$

$$z_{2,k}(\vec{x}) = 1 + \exp\{-w_{2,k0} - \sum_{i=1}^{20} w_{2,ki}z_{1,i}(\vec{x})\}, \qquad \text{for } 1 \leq k \leq 20 \tag{5.14}$$

$$f(\vec{x}) = w_{3,0} + \sum_{i=1}^{20} w_{3,i}z_{2,i}(\vec{x}), \tag{5.15}$$

where x_i denote the i-th component of input vector $\vec{x}$.

Figure 5.3b shows the back-propagation network used in the hand-written capital letter recognition problem. The network has one hidden layer of 10 sigmoidal units and 26 sigmoidal outputs. It implements a mapping from R^{100} to R^{26} of the form:

$$z_{1,k}(\vec{x}) = 1 + \exp\{-w_{1,k0} - \sum_{i=1}^{100} w_{1,ki}x_i\}, \qquad \text{for } 1 \leq k \leq 10 \tag{5.16}$$

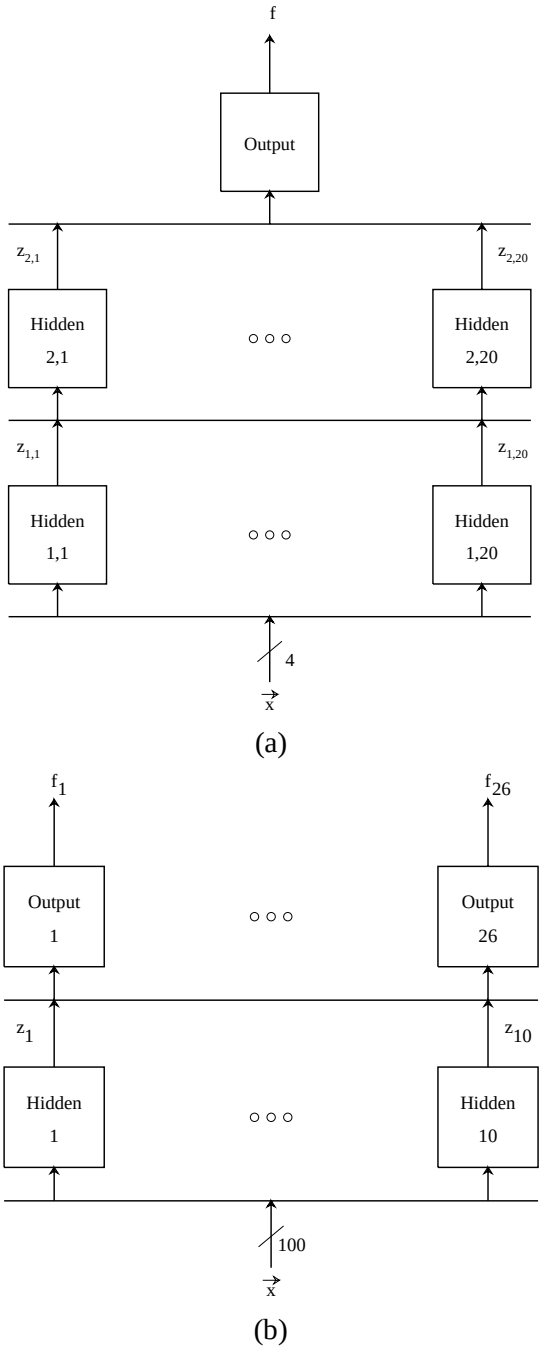


Figure 5.3: The schematic diagrams of the back-propagation architectures: (a) a network with two hidden layers of perceptrons for addressing the Mackey-Glass time series prediction problem, (b) a network with one hidden layer of perceptrons for addressing a hand-written capital letter recognition task.

$$f_k(\vec{x}) \quad = \quad 1 + \exp\{-w_{2,k0} - \sum_{i=1}^{10} w_{2,ki} z_{1,i}(\vec{x})\}, \qquad \text{for } 1 \leq k \leq 26 \ . \tag{5.17}$$

5.3 Mackey-Glass Time Series Prediction

This section reports the empirical evaluation of the *Het*, *RBF* and *BP* architectures on the Mackey-Glass time series prediction [45]. The Mackey-Glass time series used in this evaluation is derived by integrating the differential-delay equation [44, 45]:

$$\frac{dx[t]}{dt} = \frac{0.2x[t-17]}{1+x^{10}[t-17]} - 0.1x[t]. \tag{5.18}$$

Using this time series $x[t]$, we then define a training set of 10,000 input-output pairs, where input $\vec{x}_i$ is defined as $(x[i], x[i-6], x[i-12], x[i-18])$ and output y_i as $x[i+85]$.

In this simulation, we use a *Het* architecture that partitions the input domain into 8 regions. We initialize each reference vector $\vec{c}_k$ to $(0.93, 0.93, 0.93, 0.93)$, where 0.93 is the mean of the Mackey-Glass time series. For the *RBF* architecture, we use a system consisting of 64 RBF units. We initialize $\vec{c}_k$ to $(0.93, 0.93, 0.93, 0.93)$. For the *BP* architecture, we initialize all the weight parameters to random values ranging from -0.5 to 0.5. We train each of the above systems on a sequence of input-output patterns randomly selected from our training set of 10,000 input-output pairs. We evaluate each architecture on a test set containing 500 input-output pairs that differ from those in the training set. We measure the performance using a *normalized mean squared error* (*NMSE*), defined as the mean squared error divided by the mean squared value of the time series.

Figure 5.4a shows the *NMSE* of the *Het*, *RBF* and *BP* systems on the test set as a function of the number of patterns presented during training. Each curve here is the average of 5 runs, where

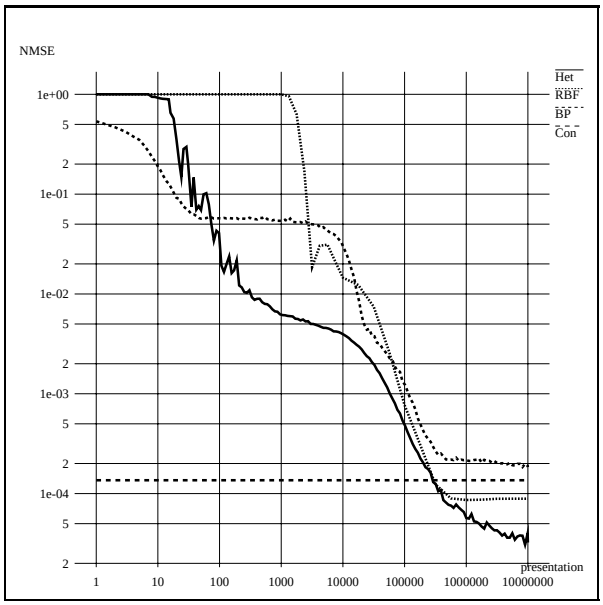
each run uses a different training sequence. For *Het* or *BP*, the *NMSE* plotted at *N* presentations is the value obtained after the system has been presented with *N* patterns. For *RBF*, the value is obtained after *2N* pattern presentations. The first *N* presentations are used to find the centers of the Gaussian functions, and the other *N* patterns are needed to adjust the heights of the Gaussian functions. The figure also shows a horizontal line *Con* representing the result of Lapedes and Farbes's experiment [45]. Using the conjugate gradient method, they were able to train a multilayer perceptron with the above topology to achieve the *NMSE* of 0.000136.[1] This line is shown here for visual reference and does not illustrate any dynamic characteristics.

This figure indicates that the *Het* system has better generalization capabilities. The steady state error of *Het*, measured after 10^7 pattern presentation, is about a third of that for *RBF*, and about a sixth of that for *BP*. With respect to learning speed, the *Het* system has a faster convergence rate than the *RBF* and *BP* systems. Furthermore, the *NMSE* of *Het* always stays below that of *RBF* and falls below that of *BP* after about 70 pattern presentations.

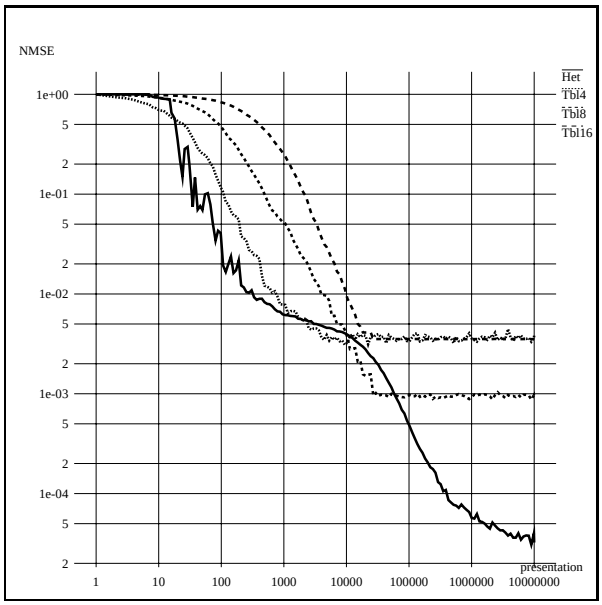
Figure 5.4b compares the *NMSE* of the *Het* system to that of the architecture based on the lookup table approach on the test set. This type of architecture can learn very rapidly but it generalizes poorly because it normally has too many degree of freedom. Thus, comparing *Het* with this architecture is a good tool to investigate how much speed the heterogeneous architecture has to sacrifice in exchange for its generalization power.

For our evaluation, we define the input domain of the lookup table approach to be $[0.41, 1.32]^4$, the smallest 4-dimensional box containing all the patterns in our training set. Three versions of the lookup table architecture, each with different levels of quantization, are tested:

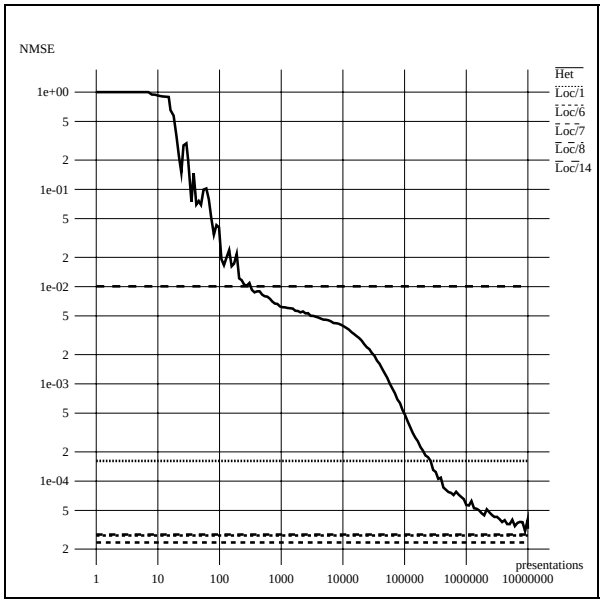
[1] In Lapedes and Farber's simulation [45], the performance of a network is measured by the normalized error, defined as the root mean squared value of the prediction error normalized by the standard deviation of the time series. The *NMSE* of 0.000136 is equivalent to the normalized error of 0.05.



(a)



(b)



(c)

Figure 5.4: Performance comparison on the Mackey-Glass time series prediction for:
(a) *Het*, *RBF* and *BP*,
(b) *Het* and the architecture based on the lookup table approach.
(c) *Het* and the architecture based on the local model approach.

- *Tbl4*: each dimension is quantized into 4 levels, resulting in a total of 256 cells;
- *Tbl8*: each dimension is quantized into 8 levels, resulting in a total of 4,096 cells;
- *Tbl16*: each dimension is quantized into 16 levels, resulting in a total of 65,536 cells;

It is evident from figure 5.4b that a lookup table with fewer cells can learn faster than one with more cells. Reducing the number of cells increases the learning rate by reducing the number of parameters that need adjustment. However, the steady state error does not improve monotonically with the number of cells. When the number of cells is reduced, the size of each cell increases. Consequently, the variation of the output data in each cell increases. As a result, the estimation of z_k becomes slower, thus delaying the convergence rate of the entire system by some degree.

The *NMSE* of *Tbl8* at steady state is lower than that of *Tbl4*. Since the cells in *Tbl8* are smaller than those in *Tbl4*, *Tbl8* can adjust its output to more closely match the variation of the target function. However, the *NMSE* of *Tbl16* at steady state is higher than that of *Tbl8* even though it has more cells. This occurs because there are too few input-output patterns in the training set. Some testing patterns fall in cells that have never seen a data sample during training–thus resulting in a large error.

Comparing the performance of *Het* to that of the look-up table approach, we see that the *NMSE* of *Het* decreases faster than those of the lookup tables for the first few hundred pattern presentations. However, the rate of decrease then gradually slows down compared to those of *Tbl4* and *Tbl8*, and the *NMSE*s of *Tbl4* and *Tbl8* drop below that of *Het* at about 2,000 and 10,000 pattern presentations, respectively. This slow-down happens because *Het* attempts to re-partition its input domain to evenly distribute the load among all the expert modules. After the input domain is re-partitioned, *Het* then rapidly reduces its *NMSE* and again drops below the curves of *Tbl4* and

Tbl8 after 12,000 and 60,000 pattern presentations, respectively. At steady state, the *NMSE* of *Het* is only about 1% of that of *Tbl4*, and about a thirtieth of that of *Tbl8*. Comparing the performance of *Tbl16* with that of *Het*, we see that the *NMSE* of *Het* is always below that of *Tbl16* and the *NMSE* of *Het* at steady state is about 1% of that of *Tbl16*.

In addition to the aforementioned comparisons, we also compare the *Het* architecture with the architecture based on the local model approach [24, 25]. This comparison allows us to evaluate the performance of the *Het* architecture, which operates in the "on-line" fashion on a simple hardware, against that of an efficient local approximation scheme, which operates in the "batch" mode on a more elaborated hardware.

Figure 5.4c shows the *NMSE*s of the *Het* architecture and the local model approaches on the test set. The horizontal lines representing the *NMSE*s of the local model approachs are shown for visual reference, and do not illustrate any dynamic characteristics. In this comparison, all 10,000 input-output patterns in the training set are used as reference vectors. We set the width of each influence Gaussian function σ_k^2 to $\|\vec{c}_k - \vec{c}_{k,nearest}\|^2/\kappa$, where $\vec{c}_{k,nearest}$ is the nearest reference vector to $\vec{c}_k$, and κ is a constant. Five values of κ are tested: 1, 6, 7, 8, and 14; and the corresponding results are depicted by *Loc/1*, *Loc/6*, *Loc/7*, *Loc/8*, and *Loc/14*, respectively. As evident from this comparison, the local model architecture can perform better than the *Het* architecture for a proper choice of κ ($\kappa = 6, 7, 8$), but only by a slight margin. This result is very promising, considering that the *Het* architecture uses simple recursive rules to adjust its parameters while the local model method employs complex algorithms, such as, the singular value decomposition to perform the weighted least squares fit and the bumptree data structure to organize the stored data.

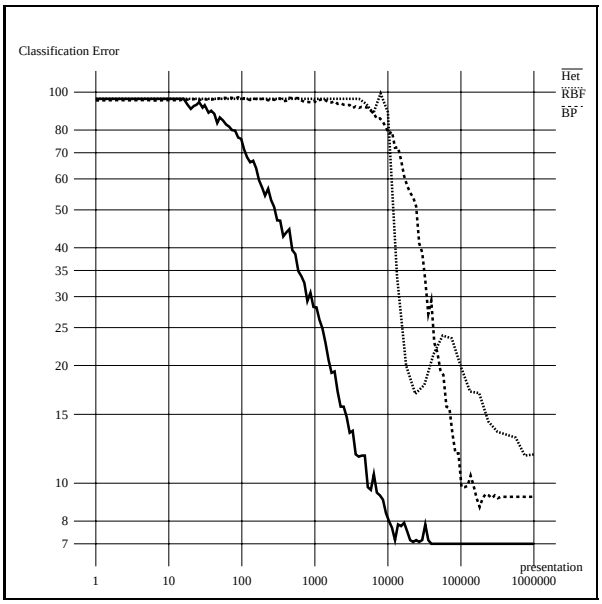
5.4 Hand-Written Letter Recognition

In this section, we test the *Het*, *RBF*, and *BP* architectures on the task of classifying hand-written capital letters. We exclude both the lookup table and local model approaches from this evaluation because of the high dimensionality of the input domain. These two approaches have a large number of parameters that need to be adjusted and thus require excessive amounts of training data. The available data set is not large enough for training these two architectures. In addition, the local model method needs to perform the weighted least squares fit to sample data, and the complexity of this computation scales proportionally to the cube of the input dimension. It is thus too expensive for this high-dimensional problem.

The character data used in this simulation are obtained from the experiment performed in [46]. These characters are hand-written on a 80 $\times$ 120 pixel window with a 5 pixel-wide brush. All characters are approximately centered and scaled to the full size of the window. Following the character entry, the window is divided into 100 regions of 8 $\times$ 12 pixels. Each of these regions is an input whose value is the percentage of "on" pixels in the region. There are thus 100 inputs, each of which could have any of 8 $\times$ 12 = 96 distinct values.

In this character recognition problem, we use the *Het* architecture that partitions the input domain into 8 regions. We initialize each component of the reference vector $\vec{c}_k$ to 10^{-10}. For the *RBF* architecture, we use a system that consists of 128 RBF units. We initialize each component of $\vec{c}_k$ to 10^{-10}. For the *BP* architecture, similar to the Mackey-Glass problem, we initialize all the weight parameters to random values ranging from -0.5 to 0.5.

These *Het*, *RBF*, and *BP* systems are used in this evaluation to estimate the membership function of an input pattern. Each system has 26 outputs, one for each capital letter. The value of



(a)

Figure 5.5: Performance comparison of the *Het*, *RBF*, and *BP* architectures on the hand-written character recognition task.

each output indicates the relative likelihood that the input belongs to the corresponding category. The input is classified by selecting the category with the maximum output value. We train each of these systems using a sequence of labeled input patterns randomly selected from a training set, consisting of 50×26 capital letters. We evaluate each system on a test set consisting of 26×10 letters, not found in the training set. The system performance is measured by the classification error, defined as the percentage of misclassifications.

Figure 5.5 shows the classification error of the *Het*, *RBF*, and *BP* systems on the test set as a function of the number of patterns presented during training. Each curve in this figure is the average of 5 runs, each using a separate training sequence. This figure shows that the classification error of *Het* decreases faster than those of *RBF* and *BP*. The learning curve of *Het* always remains below those of *RBF* and *BP*. *Het* can correctly classify all the patterns in the training set after 40,000 pattern presentations, as indicated by absence of adaptation after that time point. By comparison, *BP* needs 350,000 presentations, almost 9 times as many, and *RBF* needs 8,000,000 presentations,

or about 20 times as many.

To evaluate the generalization power of each system, we use the steady state classification error, measured after all the training patterns are correctly classified. We find that the steady state classification error of *Het* is only 7%, while those of BP and RBF are 9.2% and 11.7%, respectively.

5.5 Summary and Discussion

In this chapter, we have evaluated a heterogeneous architecture that is based on error-weighted k-means partitioning and on linear experts for addressing the subtasks. This architecture has been compared to radial basis function and back-propagation architectures on the Mackey-Glass time series prediction and on a hand-written capital letter recognition task. For both problems, we have found that the heterogeneous architecture exhibits a faster convergence rate and better generalization than other on-line architectures. Given that the input dimension of the Mackey-Glass problem is low and that of the character recognition problem is high, we conjecture that the heterogeneous architecture will perform well on problems with a wide range of input dimensions.

For the Mackey-Glass problem, where the input dimension is low, we have also compared the heterogeneous architecture with the architectures based on the lookup table and on the local model approaches. This experiment indicates that both the heterogeneous architecture and the lookup table approach have comparable speeds, but the former has much higher generalization capability. It also shows that the architecture based on the local model approach, with its complicated algorithm, can perform slightly better than the heterogeneous architecture.

Comparing the performance of the radial basis function and back-propagation architectures, we have found that the former performs better on the Mackey-Glass problem while the latter

performs better on the character recognition. This indicates that the radial basis function architecture is more suited to address problems with low input dimension while the back-propagation architecture is more suited to address problems with high input dimension.

In problems with low input dimension, such as the Mackey-Glass problem, the goal function, in general, tends to be complicated, and the input data tend to distribute over large extent of the input domain. These problems are more suited for architectures with local-support basic functions, such as a look-up table approach. For the problem with high input dimension, such as the character recognition problem, the input data tend to scatter in clusters that are quite easily separated by hyper-planes. These problems can thus be more effectively addressed by architectures with global-support basic functions, such as a multi-layered perceptron. However, both these two classes of problems can be addressed reasonably well by a heterogeneous architecture. In the heterogeneous architecture, the assigned task is divided into sub-tasks, each solved by a different expert module. This architecture thus has some flexibility in defining the input domain of each expert module, allowing the granularity of the sub-task to match the characteristics of the basic function in the expert module.

Chapter 6

Serial Implementations and Complexity Evaluation

In the previous chapter, we have evaluated the performance of the heterogeneous architectures *(Het)*, which is based on error-weighted k-means partitioning, with the examples of the Mackey-Glass time series prediction and of the hand-written capital letter recognition task. We have compared in this evaluation the performance of the *Het* architecture to those of the radial basis function *(RBF)* architecture, of the architecture based on lookup table approach *Tbl*, of the architecture based on local model approach *Loc*, and of the back-propagation *(BP)* architectures in terms of their *generalization capabilities* and *convergence rates*.

In this chapter and the next, we will evaluate the implementation complexities of the *Het*, *RBF*, and *BP* architectures for the Mackey-Glass time series prediction and for the hand-written capital letter recognition problem. The *Tbl* and *Loc* architectures are excluded from this evaluation because they require large amounts of hardware when addressing problems with high-

dimensional inputs. In addition, the *Loc* architecture utilizes a complicated data structure and its algorithm involves large amount of computation. Their hardware implementations are relatively more complicated than that of the *Het* architecture.

An artificial neural network architecture can be implemented with different amount of concurrency of execution, ranging from a *serial* implementation at one end of the spectrum to a *maximally parallel* implementation at the other end. In order to gain insight into the relative implementation complexities of the *Het*, *RBF*, and *BP* architecture, we investigate the implementation complexities of these three architectures for two extreme cases: serial and maximally parallel. In this chapter, we first describe the serial implementation of each architecture by its learning algorithm. We then evaluate the complexity of each implementation by determining the number of arithmetic operations in its learning algorithm. The learning time of each implementation, defined as the time needed to perform all the arithmetic operations in the training process, is also computed. In the next chapter, we evaluate the complexities for the parallel implementation of the *Het*, *RBF*, and *BP* architecture in terms of the hardware costs and running times.

6.1 Heterogeneous Architectures

This section describes the serial implementations of heterogeneous architectures based on error-weighted k-means partitioning *(Het)* for both function approximation and classification. It also determines the number of arithmetic operations required by the implementations.

6.1.1 Function Approximation

For a *Het* architecture representing a piecewise linear mapping from R^M to R^N, its serial implementation is given by the algorithm *Het-I*.

Het-I Algorithm

Notation

Let K denote the number of expert module in the system.

Let $\vec{x}$ denote an input vector and x_i denote its i-th component.

Let $\vec{c}_k$ denote the k-th reference vector and $c_{k,i}$ denote its i-th component.

Let $\vec{g}$ denote the goal function and g_i denote its i-th component.

Let $\vec{f}$ denote the network function of *Het* and f_i denote its i-th component.

Let $\vec{f}_k$ denote the network function of the k-th expert module, and $f_{k,i}$ denote its i-th component.

Let $w_{k,ij}$ denote the coefficient of x_j for generating output $f_{k,i}$.

Let $d(\vec{x}, \vec{c}_k)$ denote the deviation between $\vec{x}$ and $\vec{c}_k$.

Let k_w denote the index of the winning reference vector, to which input $\vec{x}$ is closest.

Let α and β denote the constants used in the running average for computing $\hat{\nu}_k$, where $\alpha + \beta = 1$.

Begin

Step 1: find the winning index k_w.

Step 1.1: compute deviation $d(\vec{x}, \vec{c}_k) = \hat{\epsilon}_k \hat{\nu}_k \sum_{i=1}^{M} (x_i - c_{k,i})^2$, for $1 \le k \le K$.

Step 1.2: find k_w such that $d(\vec{x}, \vec{c}_{k_w}) \le d(\vec{x}, \vec{c}_k)$, for $1 \le k \le K$.

Step 2: compute the network output $\vec{f}$.

Step 2.1: compute $\vec{f}_{k_w}$ according to:

$$f_{k_w,i}(\vec{x}) = w_{k_w,i0} + \sum_{j=1}^{M} w_{k_w,ij}\, x_j \qquad \text{for } 1 \le i \le N.$$

Step 2.2: define $\vec{f}(\vec{x}) = \vec{f}_{k_w}(\vec{x})$.

Step 3: compute the learning rates η_{km} and η_{lms} using the equations:

$$\eta_{km} = \ln(\textstyle\sum_{k=1}^{K} \hat{\nu}_k) + \{\sum_{k=1}^{K} -\hat{\nu}_k \ln \hat{\nu}_k\} / \{\sum_{k=1}^{K} \hat{\nu}_k\},$$

$$\eta_{lms} = \eta_{lms} + 0.01.$$

Step 4: update parameters $w_{k_w ij}$ of the k_w-th expert module using the equations:

$$w_{k_w ij} = \begin{cases} w_{k_w ij} + \eta_{lms}\{g_i(\vec{x}) - f_{k_w i}(\vec{x})\} & \text{for } 1 \le i \le N, \text{ and } j = 0 \\ w_{k_w ij} + \eta_{lms}\{g_i(\vec{x}) - f_{k_w i}(\vec{x})\} x_j & \text{for } 1 \le i \le N \text{ and } 1 \le j \le M \end{cases}$$

Step 5: update parameter $\hat{\epsilon}_{k_w}$ according to:

$$\hat{\epsilon}_{k_w} = \alpha \hat{\epsilon}_{k_w} + \beta \sum_{i=1}^{N} \{g_i(\vec{x}) - f_{k_w i}(\vec{x})\}^2.$$

Step 6: update the winning reference vector $\vec{c}_{k_w}$ using the equation:

$$c_{k_w i} = c_{k_w i} + \eta_{km}(x_i - c_{k_w i}) \qquad \text{for } 1 \le i \le M.$$

Step 7: update parameters $\hat{\nu}_k$ according to the equations:

$$\hat{\nu}_k = \begin{cases} \alpha \hat{\nu}_k + \beta \hat{\epsilon}_k \sum_{i=1}^{N} (x_i - c_{k_w i})^2 & \text{if } k = k_w \\ \alpha \hat{\nu}_k & \text{otherwise} \end{cases}$$

End

In the following, we analyze step by step the *Het-I* algorithm for the number of the arithmetic operations performed in its training cycle. For convenience, we categorize these operations, based on the approximate complexity of their hardware realization, into 4 classes:

- comparison;
- addition: which includes subtraction;
- multiplication: which includes division and squaring operation;
- and nonlinear-function computation: which includes the computations of sigmoid, exponential, and logarithm functions.

Step 1: Step 1.1 needs $2KM-K$ additions and $KM+2K$ multiplications, and step 1.2 needs $K-1$ comparisons.

Step 2: This step needs *MN* additions and *MN* multiplications.

Step 3: This step can be subdivided as follows:

Step 3.1: compute $s_1 = \sum_{k=1}^{K} \hat{\nu}_k$.

Step 3.2: compute $s_2 = \sum_{k=1}^{K} -\hat{\nu}_k \ln \hat{\nu}_k$.

Step 3.3: set $\eta_{km} = \ln(s_1) + s_2/s_1$, and set $\eta_{lms} = \eta_{km} + 0.01$.

This implementation needs $2K$ additions, $K+1$ multiplications, and $K+1$ logarithmic-function computations.

Step 4: This step can be subdivided as follows:

for $i = 1$ to N do

Step 4.1: compute $\delta = \eta_{lms}\{g_i(\vec{x}) - f_{k_w,i}(\vec{x})\}$

Step 4.2: set $w_{k_w,ij} = \begin{cases} w_{k_w,ij} + \delta & \text{for } j = 0 \\ w_{k_w,ij} + x_j\delta & \text{for } 1 \le j \le M \end{cases}$

This implementation needs $MN+2N$ additions and $MN+N$ multiplications.

Step 5: This step can be subdivided as follows:

Step 5.1: compute $\Delta^2 = \sum_{i=1}^{N}\{g_i(\vec{x}) - f_{k_w,i}(\vec{x})\}^2$.

Step 5.2: set $\hat{\epsilon}_{k_w} = \alpha\hat{\epsilon}_{k_w} + \beta\Delta^2$.

Step 5.1 needs only $N-1$ additions and N multiplications since all the differences between $g_i(\vec{x})$ and $f_{k_w,i}(\vec{x})$ are already computed in step 4.1,

Step 5.2 needs 1 addition and 2 multiplications.

Step 6: This step needs $2M$ additions and M multiplications.

Step 7: This step can be subdivided as follows:

Step 7.1: for $k = 1$ to K do

set $\hat{\nu}_k = \alpha\hat{\nu}_k$.

Step 7.2: compute $\Delta^2 = \hat{\epsilon}_{k_w} \sum_{i=1}^{M} (x_i - c_{k_w,i})^2$.

Step 7.3: set $\hat{\nu}_{k_w} = \hat{\nu}_{k_w} + \beta\Delta^2$.

Step 7.1 needs K multiplications.

Step 7.2 needs only $M-1$ additions and $M+1$ multiplications since all the differences between x_i and $c_{k_w,1}$ are already computed in step 6.

Step 7.3 needs 1 addition and 1 multiplications.

End of Analysis

This analysis indicates that one training cycle of *Het-I* consists of

- $K-1$ comparisons,
- $2MN+2KM+K+3M+3N+2$ additions,
- $KM+2MN+4K+2M+2N+5$ multiplications,
- and $K+1$ logarithm-function calculations.

6.1.2 Classification

For a heterogeneous architecture that classifies input $\vec{x} \in R^M$ to N categories, its serial implementation is specified by the algorithm *Het-II*. Note the *Het-II* is similar to *Het-I* except for step 2, 4 and 5.

Het-II Algorithm

Notation

Let C_{sys} denotes the category indicated by the heterogeneous system.

Let C_{goal} denote the category specified by the goal function.

Begin

Step 1: find the winning index k_w.

Step 1.1: compute deviation $d(\vec{x}, \vec{c}_k) = \hat{\epsilon}_k \hat{\nu}_k \sum_{i=1}^{M} (x_i - c_{k,i})^2$, for $1 \leq k \leq K$.

Step 1.2: find k_w such that $d(\vec{x}, \vec{c}_{k_w}) \leq d(\vec{x}, \vec{c}_k)$, for $1 \leq k \leq K$.

Step 2: determine the category of input $\vec{x}$.

Step 2.1: compute the output of the k_w-th expert module according to the equation:

$$f_{k_w,i}(\vec{x}) = w_{k_w,i0} + \sum_{j=1}^{M} w_{k_w,ij}\, x_j, \qquad \text{for } 1 \leq i \leq N.$$

Step 2.2: find C_{sys} such that $f_{k_w,C_{sys}}(\vec{x}) \geq f_{k_w,i}(\vec{x})$, for $1 \leq i \leq N$.

Step 3: compute the learning rates η_{km} and η_{lms} using the equations:

$$\eta_{km} = \ln(\textstyle\sum_{k=1}^{K} \hat{\nu}_k) + \{\sum_{k=1}^{K} -\hat{\nu}_k \ln \hat{\nu}_k\} / \{\sum_{k=1}^{K} \hat{\nu}_k\},$$

$$\eta_{lms} = \eta_{lms} + 0.01.$$

if $C_{sys} = C_{goal}$ then do step 4 and 5.

Step 4: update parameters $w_{k_w,ji}$ of the k_w-th expert module using the equations:

$$w_{k_w,ij} = \begin{cases} w_{k_w,ij} + \eta_{lms}\{g_i(\vec{x}) - y_{k_w,i}(\vec{x})\} & \text{for } i = C_{sys}, C_{goal}; \text{ and } j = 0 \\ w_{k_w,ij} + \eta_{lms}\{g_i(\vec{x}) - y_{k_w,i}(\vec{x})\} x_j & \text{for } i = C_{sys}, C_{goal}; \text{ and } 1 \leq j \leq M \end{cases}$$

where $g_{C_{sys}}(\vec{x}) = 0$ and $g_{C_{goal}}(\vec{x}) = 1$.

Step 5: update parameter $\hat{\epsilon}_{k_w}$ according to:

$$\hat{\epsilon}_{k_w} = \alpha\hat{\epsilon}_{k_w} + \beta.$$

Step 6: update the winning reference vector $\vec{c}_{k_w}$ using the equation:

$$c_{k_w,i} = c_{k_w,i} + \eta_{km}(x_i - c_{k_w,i}) , \qquad \text{for } 1 \leq i \leq M.$$

Step 7: update parameters $\hat{\nu}_k$ according to the equations:

$$\hat{\nu}_k = \begin{cases} \alpha\hat{\nu}_k + \beta\hat{\epsilon}_k \sum_{i=1}^{N}(x_i - c_{k_w,i})^2 & \text{if } k = k_w \\ \alpha\hat{\nu}_k & \text{otherwise} \end{cases}$$

End

Following the method used to analyze the *Het-I* algorithm, we find that one training cycle of the *Het-II* algorithm consists of

- $K+N-2$ comparison,
- $2KM+MN+K+5M+7$ additions,
- $KM+MN+4K+4M+6$ multiplications,
- and $K+1$ logarithm function computations.

6.2 Radial Basis Function Architectures

6.2.1 Function Approximation

As reviewed in section 5.2.2, the computation performed by the radial basis function architecture used in this dissertation is described by 3 procedures:

- The adaptive k-means algorithm based on the variation-weighted deviation measure for placing the centers of the Gaussian radial basis functions.

- The *global nearest neighbor* rule [29] for computing the Gaussian width. With this rule, the widths of all the Gaussian functions are defined to be

$$\{1/K \sum_{k=1}^{K} \|\vec{c}_k - \vec{c}_{k,nearest}\|^2\}^{1/2}, \tag{6.1}$$

  where $\vec{c}_{k,nearest}$ is the nearest Gaussian center $\vec{c}_i$ to $\vec{c}_k$.

- The *least mean square (LMS)* algorithm for determining the heights of the Gaussian functions that minimize the mean squared error between the network function and the goal function.

The serial implementation of the *RBF* architecture is thus specified by *K-Means*, *Width* and *Height-I*, which are described in the following.

K-Means Algorithm

Notation

Let K denote the number of the Gaussian RBF's the system.

Let $\vec{c}_k$ is the k-th Gaussian center and $c_{k,i}$ denote its i-th component.

Let ϕ_k denote the value of the k-th Gaussian function.

Let w_{ik} denote the coefficient of ϕ_k for generating output $f_i(\vec{x})$.

Let $d(\vec{x}, \vec{c}_k)$ denote the variation-weighted deviation between $\vec{x}$ and $\vec{c}_k$.

Let k_w denote the index of the winning Gaussian center, to which input $\vec{x}$ is closest.

Begin

Step 1: find the winning index k_w.

Step 1.1: compute deviation $d(\vec{x}, \vec{c}_k) = \hat{v}_k \sum_{i=1}^{M} (x_i - c_{k,i})^2$, for $1 \leq k \leq K$.

Step 1.2: find k_w such that $d(\vec{x}, \vec{c}_{k_w}) \leq d(\vec{x}, \vec{c}_k)$, for $1 \leq k \leq K$.

Step 2: compute the learning rates η_{km} using the equations:

$$\eta_{km} = \ln(\textstyle\sum_{k=1}^{K} \hat{v}_k) + \{\sum_{k=1}^{K} -\hat{v}_k \ln \hat{v}_k\} / \{\sum_{k=1}^{K} \hat{v}_k\},$$

Step 3: update the winning Gaussian center $\vec{c}_{k_w}$ using the equation:

$$c_{k_w,i} = c_{k_w,i} + \eta_{km}(x_i - c_{k_w,i}) \,, \qquad \text{for } 1 \leq i \leq M.$$

Step 4: update parameters $\hat{v}_k$ according to the equation:

$$\hat{v}_k = \begin{cases} \alpha\hat{v}_k + \beta\sum_{i=1}^{M}(x_i - c_{k_w,i})^2 & \text{if } k = k_w \\ \alpha\hat{v}_k & \text{otherwise} \end{cases}$$

End

Analysis of K-Means

Step 1: Step 1.1 needs $2KM-K$ additions and $KM+K$ multiplications, and step 1.2 needs $K-1$ comparisons.

Step 2: Step 2 needs $2K-1$ additions, $K+1$ multiplications, and $K+1$ logarithmic-function computations.

Step 3: This step needs $2M$ additions and M multiplications.

Step 4: This step needs M additions and $M+K+1$ multiplications.

End of Analysis

The total operations in one cycle of the *K-Means* algorithm consists of

- $K-1$ comparisons,
- $2KM+K+3M-1$ adiitions,
- $KM+3K+2M+2$ multiplications,
- and $K+1$ logarithmic-function computations.

Width Algorithm

Begin

Step 1: for $k = 1$ to K do

find $d_{k,min}$ which is the minimum of $||\vec{c}_k - \vec{c}_i||^2$ for $1 \leq i \leq K$ and $i \neq k$.

Step 2: compute the width σ^2 defined as $(\sum_{i=1}^{K} d_{i,min})/K$.

End

Analysis of Width

Step 1: This step needs $K^2 - 2K$ comparisons, $2K^2M - K^2 - 2KM + K$ additions, and $K^2M - KM$ multiplications.

Step 2: This step needs $K - 1$ additions and 1 multiplication.

End of Analysis

This analysis indicates that the total number of operations required for determining the Gaussian width are

- $K^2 - 2K$ comparisons,
- $2K^2M - K^2 - 2KM + K$ additions,
- and $K^2M - KM + 1$ multiplications.

Height-I Algorithm

Begin

Step 1: for $k = 1$ to K do

compute ϕ_k = exp$(-\|\vec{x} - \vec{c}_k\|^2/\sigma^2)$.

Step 2: compute $\sum_{i=1}^{K} \phi_k$.

Step 3: for $i = 1$ to N do

compute output $y_i = \{\sum_{k=1}^{K} w_{ik}\phi_k\}/\{\sum_{k=1}^{K} \phi_k\}$.

Step 4: update parameters w_{ik} using the equations:

$$w_{ik} = w_{ik} + \eta_{lms}\{g_i(\vec{x} - y_i(\vec{x})\}\phi_k, \qquad \text{for } 1 \leq i \leq N \text{ and } 1 \leq k \leq K.$$

End

Analysis of Height-I

Step 1: This step needs 2*KM*−*K* addititions, *KM*+*K* multiplications, and *K* exponential function computations.

Step 2: This step needs *K*−1 additions.

Step 3: This step needs *KN*−*K* additions and *KN*+*N* multiplications.

Step 4: This step needs 2*KN* additions and 2*KN* multiplications.

End of Analysis

The total number of operations in one cycle of *Height-I* consists of

- 2*KM*+3*KN*−*K*−1 additions,
- *KM*+3*KN*+*K*+1 multiplications,

- and *K* exponential-function computations.

6.2.2 Classification

When we apply the radial basis function architecture to classification problems, we need to modify the algorithm *Height-I* so that the output of the architecture can be used to estimate the likelihoods of an input pattern belonging to the various categories. The modified algorithm, refered as *Height-II*, as well as its analysis, is described in the following.

Height-II Algorithm

Notation

Let C_{sys} denotes the category indicated by the *RBF* system.

Let C_{goal} denote the category indicated by the goal function.

Begin

Step 1: for $k = 1$ to K do

compute ϕ_k = $\exp(-\|\vec{x} - \vec{c}_k\|^2/\sigma^2)$.

Step 2: compute $\sum_{i=1}^{K} \phi_k$.

Step 3: determine the category of input $\vec{x}$.

Step 3.1: for $i = 1$ to N do

compute output $f_i(\vec{x})$ = $\{\sum_{k=1}^{K} w_{ik}\phi_k\}/\{\sum_{k=1}^{K} \phi_k\ \}$.

Step 3.2: find C_{sys} such that $f_{C_{sys}}(\vec{x}) \geq f_i(\vec{x}),$ for $1 \leq i \leq N$.

Step 4: if $C_{sys} = C_{goal}$ do

update parameters w_{ik} using the equations:

$$w_{ik} = w_{ik} + \eta_{lms}\{g_i(\vec{x}) - f_i(\vec{x})\}\phi_k, \qquad \text{for } i = C_{sys}, C_{goal}; \text{ and } 1 \leq k \leq K,$$

where $f_{C_{sys}}(\vec{x}) = 0$ and $f_{C_{goal}}(\vec{x}) = 1$.

End

The total number of operations in one training cycle of the *Height-II* algorithm consist of

- $N-1$ comparisons,
- $2KM+KN+3K-1$ additions,
- $KM+KN+5K+N$ multiplications,
- and K exponential function computations.

6.3 Back Propagation Architectures

6.3.1 Function Approximation

This section describes the serial implementations and the analyses of the back-propagation architectures for function approximation. It is concentrated on the class of architectures with a network of two hidden layers and linear outputs. We assume that the architecture implements a mapping from R^M to R^N, and that there are H_1 sigmoidal units in the first hidden layer, H_2 sigmoidal units in the second hidden layer, and N linear output units in the output layer.

BP-I Algorithm

Notation

Let f_i denote the i-th component of network function $\vec{f}$.

Let g_i denote the i-th component of goal vector $\vec{g}$.

Let $z_{1,i}$ denote the output of the i-th unit in the first hidden layer.

Let $z_{2,i}$ denote the output of the i-th unit in the second hidden layer.

Let $w_{1,ij}$ denote the co-efficient of x_j for generating output $z_{1,i}$.

Let $w_{2,ij}$ denote the co-efficient of $z_{1,j}$ for generating output $z_{2,i}$.

Let $w_{3,ij}$ denote the co-efficient of $z_{2,j}$ for generating output f_i.

Let $s(\alpha)$ denote the sigmoid function of α defined as $\{1+\exp(-\alpha)\}^{-1}$.

Begin

Step 1: generate the outputs of the first hidden layer:

$$z_{1,i} = s(w_{1,i0} + \sum_{j=1}^{M} w_{1,ij} x_j) \qquad \text{for } 1 \le i \le H_1.$$

Step 2: generate the outputs of the second hidden layer:

$$z_{2,i} = s(w_{2,i0} + \sum_{j=1}^{H_1} w_{2,ij} z_{1,j}) \qquad \text{for } 1 \le i \le H_2.$$

Step 3: generate the outputs of the network:

$$f_i = w_{3,i0} + \sum_{j=1}^{H_2} w_{3,ij} z_{2,j} \qquad \text{for } 1 \le i \le N.$$

Step 4: update the parameters in the output layer.

for $i = 1$ to N do

Step 4.1: compute $\delta_{3,i} = \eta\{g_i(\vec{x}) - f_i(\vec{x})\}$.

Step 4.2: set $w_{3,ij} = \begin{cases} w_{3,ij} + \delta_{3,i} & j = 0 \\ w_{3,ij} + z_{2,j}\delta_{3,i} & \text{for } 1 \le j \le H_2 \end{cases}$

Step 5: update the parameters in the second layer.

for $i = 1$ to H_2 do

Step 5.1: compute $\delta_{2,i} = z_{2,i}(1 - z_{2,i}) \sum_{k=1}^{N} w_{3,ki}\delta_{3,k}$.

Step 5.2: set $w_{2,ij} = \begin{cases} w_{2,ij} + \delta_{2,i} & j = 0 \\ w_{2,ij} + z_{1,j}\delta_{2,i} & \text{for } 1 \le j \le H_1 \end{cases}$

Step 6: update the parameters in the first layer.

for $i = 1$ to H_1 do

Step 6.1: compute $\delta_{1,i} = z_{1,i}(1 - z_{1,i}) \sum_{k=1}^{H_2} w_{2,ki}\delta_{2,k}$.

Step 6.2: set $w_{1,ij} = \begin{cases} w_{1,ij} + \delta_{1,i} & j = 0 \\ w_{1,ij} + x_j\delta_{1,i} & \text{for } 1 \le j \le M \end{cases}$

End

Analysis of BP-I

Step 1: This step needs H_1M additions, H_1M multiplications, and H_1 sigmoid function computations.

Step 2: This step needs H_1H_2 additions, H_1H_2 multiplications, and H_2 sigmoid function computations.

Step 3: This step needs H_2N additions and H_2N multiplications.

Step 4: Step 4.1 needs 1 addition and 1 multiplication.

Step 4.2 needs 1 addition.

Step 4.3 needs H_2 additions and H_2 multiplications.

Since these three steps have to be performed N times, the total operations performed in step 4 are H_2N+2N additions and H_2N+N multiplications.

Step 5: Step 5.1 needs N additions and $N+2$ multiplications.

Step 5.2 needs 1 addition.

Step 5.3 needs H_1 additions and H_1 multiplications.

Since these steps are to be repreated H_2 times, the total operations performed in step 5 are $H_1H_2+H_2N+H_2$ additions and $H_1H_2+H_1N+2H_2$ multiplications.

Step 6: Step 6.1 needs H_2 additions and H_2+2 multiplications.

Step 6.2 needs 1 addition.

Step 6.3 needs M additions and M multiplications.

Since these steps are to be repreated H_1 times, the total operations performed in step 6 are

$H_1H_2+H_1M+H_1$ additions and $H_1H_2+H_1M+2H_1$ multiplications.

End of Analysis

The computation in one cycle of *BP-I* consists of

- $3H_1H_2+2H_1M+3H_2N+H_1+H_2+2N$ additions,
- $3H_1H_2+2H_1M+H_1N+2H_2N+H_1+H_2$ multiplications,
- and H_1+H_2 sigmoid function computations.

6.3.2 Classification

The serial implementation of the back-propagation architecture used in the character recogniton problem is described by the *BP-II* algorithm. To allow the implementation to be more general, we assume that the architecture is used for a classification task involving N categories, and that its network has one hidden layer containing *H* perceptrons.

BP-II Algorithm

Notation

Let C_{sys} denote the category of the input pattern indicated by the back-propagation system.

Let C_{goal} denote the category of the input pattern specified by the goal function.

Begin

Step 1: generate the outputs of the first hidden layer:

$$z_{1,i} = s(w_{1,i0} + \sum_{j=1}^{M} w_{1,ij} x_j) \qquad \text{for } 1 \leq i \leq H_1.$$

Step 2: generate the outputs of the network:

$$f_i(\vec{x}) = s(w_{2,i0} + \sum_{j=1}^{H_1} w_{2,ij} z_{1,j}) \qquad \text{for } 1 \leq i \leq H_2.$$

Step 3: classify the category of the input by selecting C_{sys} such that $f_{C_{sys}}(\vec{x}) \geq f_i(\vec{x})$ for $1 \leq i \leq N$.

if $C_{sys} = C_{goal}$ then do steps 4 and 5.

Step 4: update the parameters in the output layer.

for $i = 1$ to N do

Step 4.1: compute $\delta_{2,i} = \eta(1 - y_i)y_i(g_i(\vec{x}) - f_i(\vec{x}))$.

where $g_i(\vec{x}) = 1$ if $i = C_{goal}$ and 0 otherwise.

Step 4.2: set $w_{2,ij} = \begin{cases} w_{2,ij} + \delta_{2,i} & j = 0 \\ w_{2,ij} + z_{1,j}\delta_{2,i} & \text{for } 1 \leq j \leq H_1 \end{cases}$

Step 5: update the parameters in the first layer.

for $i = 1$ to H_1 do

Step 5.1: compute $\delta_{1,i} = z_{1,i}(1 - z_{1,i}) \sum_{k=1}^{N} w_{2,ki}\delta_{2,k}$.

Step 5.2: $w_{1,ij} = \begin{cases} w_{1,ij} + \delta_{1,i} & j = 0 \\ w_{1,ij} + x_j\delta_{1,i} & \text{for } 1 \leq j \leq M \end{cases}$

End

The computation requirement in one training cycle of *BP-II* consists of

- $N-1$ comparisons,
- $2H_1M+3H_1N+H_1+3N$ additions,
- $2H_1M+3H_1N+2H_1+3N$ multiplications,
- and H_1+N sigmoid function computations.

6.4 Complexity Comparison

This section compares the complexity of the *Het*, *RBF* and *BP* architectures for the Mackey-Glass problem described in section 5.3 and for the hand-written capital letter recognition

problem described in section 5.4. The complexity of each architecture is measured by the number of arithmetic operations needed by its learning algorithm to perform one training cycle.

As described in section 5.3, the Mackey-Glass problem is formulated as the approximation of a function that maps a point in 4 dimensional input domain to a scalar value (M=4 and N=1). In this particular evaluation, we use a *Het* architecture that partitions the input domain into 8 regions (K=8). We compare this *Het* architecture against a *RBF* architecture with 64 RBF units (K=64) and against a *BP* architecture with two hidden layers, each of 20 sigmoidal units (H_1=H_2=20). Using these specifications and the results derived from sections 6.1, 6.2 and 6.3, we compute the number of the arithmetic operations needed by the three architectures to complete their training cycles. The results from this computation are listed in table 6.1. Table 6.1 also lists the numbers of arithmetic operations required by the learning algorithms of the *Het*, *RBF*, and *BP* architectures for the hand-written capital letter recognition described in section 5.4. These three architecture are used to classify an input of 100 dimensions into 26 capital letters (M=100 and N=26). In this problem, we use a *Het* architecture that partitions the input domain into 8 regions (K=8). We compare this *Het* architecture against a *RBF* architecture with 128 RBF units (K=128) and against a *BP* architecture with one hidden layer of 10 sigmoidal units (H=10).

6.5 Convergence Rate vs. Computation Time

In chapter 5, we have evaluated the convergence rates of the *Het*, *RBF* and *BP* architectures with respect to the number of pattern presentations. Because the computational requirements for each pattern presentation for the various architectures are different, it is also necessary to compare the computation time required in each pattern presentation in order to evaluate the overall convergence

Table 6.1: Arithmetic Operations Required in One Training Cycle.

Mackey-Glass				
Algorithm	Comparison	Addition	Multiplication	Nonlinear
Het-I	7	97	87	9
RBF(K-Means)	63	588	467	65
RBF(Width)	3968	28224	16129	0
RBF(Height-I)	0	639	513	64
BP-I	0	1462	1500	40
Character				
Algorithm	Comparison	Addition	Multiplication	Nonlinear
Het-II	32	4715	3838	9
RBF(K-Means)	127	26028	13387	129
RBF(Width)	16128	3234944	1625601	0
RBF(Height-II)	25	29311	16779	128
BP-II	25	2868	2878	36

rates. For an artificial neural network architecture with on-line learning, its computation time is equivalent to the product of the number of pattern presentations and the time needed for performing one training cycle.

To determine a *training-cycle* time, it is necessary to know the execution times of the operations performed in the algorithm. In order to establish such times, we assume that all comparisons and additions are carried out for two 32-bit numbers, and each multiplication is for two 16-bit numbers. We also assume that each non-linear function computation is performed by looking up an entry in a ROM having 256 entries each of 32 bits. The execution times of these operations are measured in term of *addition time units*, where 1 addition time unit *(atu)* is the time required to perform one addition. Using the estimated specifications of the SPERT chip [49, 50], which is a micro-processor for artificial neural network computation being implemented at International Computer Science Institute (ICSI), we assume the execution time of a comparison operation to be equal to 1 *atu*, the execution time of multiplication to be equal to 2 *atu*s, and the

Table 6.2: Time for Completing One Training Cycle in Serial Mode

Application	Algorithm	Time (*atu*)	Ratio
Mackey-Glass	*Het-I*	296	1.00
	RBF(K-Means)	1715	5.79
	RBF(Width)	64450	217.74
	RBF(Height-I)	1793	6.06
	BP-I	4542	15.34
Character	*Het-II*	12441	1.00
	RBF(K-Means)	53187	4.28
	RBF(Width)	3251202	261.33
	RBF(Height-II)	63150	5.08
	BP-II	8721	0.70

time required for retrieval of an entry from ROM to be 2 *atu*s.

Table 6.2 lists the training-cycles of the learning algorithms in the *Het*, *RBF*, and *BP* architectures for the Mackey-Glass problem and for the hand-written capital letter recognition task. It also lists the ratio of a training-cycle time for each algorithm to that of *Het* of the same problem.

Figure 6.1a and 6.1b show the *NMSE* as the function of the number of pattern presentations for the Mackey-Glass and letter recognition problems. These two figures are the results from the evaluations performed in section 5.3 and 5.4, and they are shown here again for the purpose of comparison with figure 6.1c and 6.1d.

Figure 6.1c shows the *NMSE* of the *Het*, *RBF*, and *BP* architectures with respect to the *computation time* for the Mackey-Glass problem, where the computation time is measured in terms of *training-cycle* times of *Het-I*. The overall computation time of *Het* is equal to $N_p T_{Het}$, where N_p stands for the number of pattern presentations, and T_{Het} stands for the training-cycle time of *Het-I*. The overall computation time of *BP* is equal to $N_p T_{BP}$, where T_{BP} stands for the training-cycle time of *BP-I*. For *RBF*, its training procedure is divided into 3 stages. The computation time for

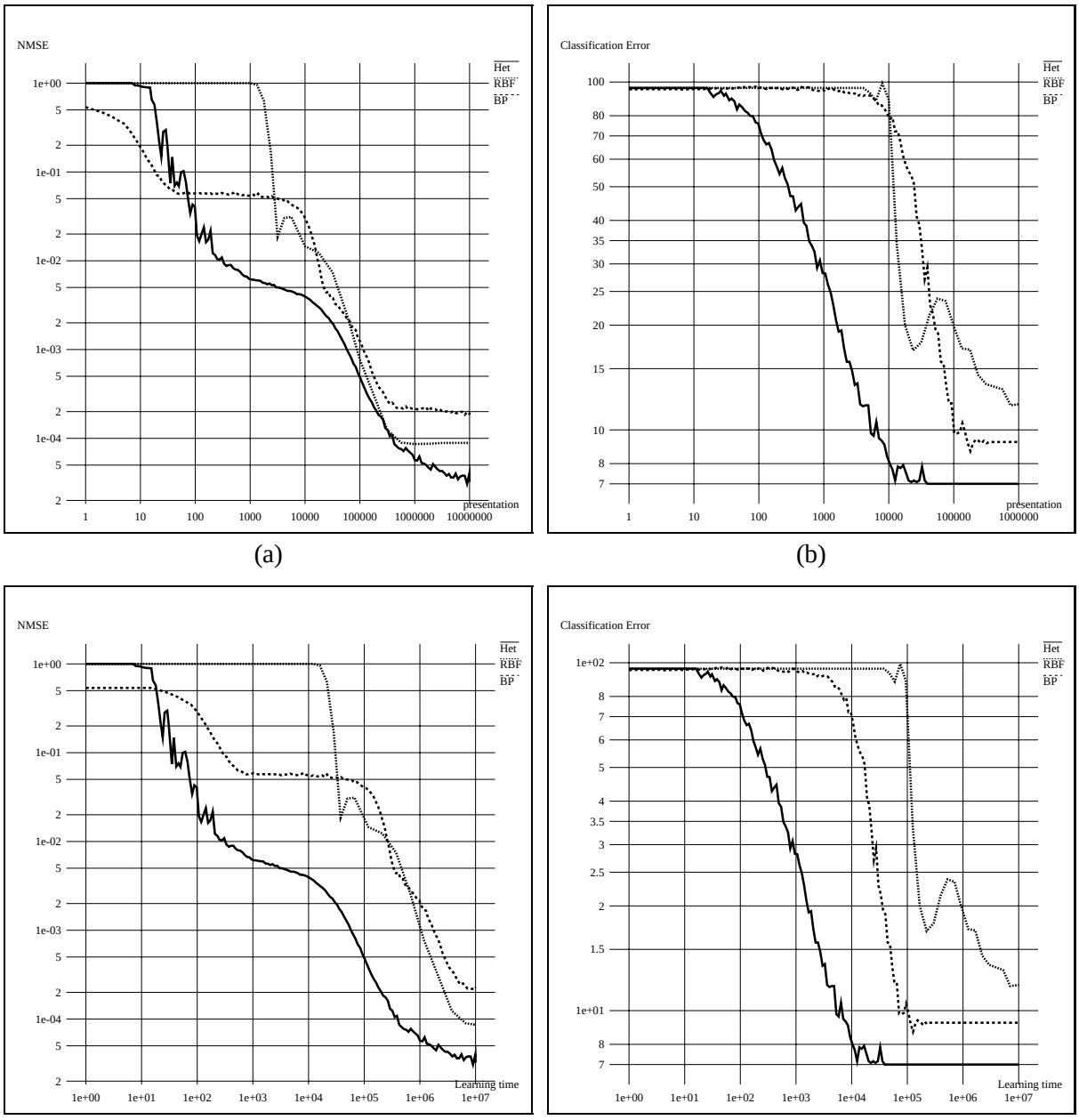


Figure 6.1: Performance comparison of *Het*, *RBF*, and *BP* architectures.
(a) *NMSE* vs. number of pattern presentations for the Mackey-Glass problem.
(b) *NMSE* vs. number of pattern presentations for the hand-written letter recognition problem.
(c) *NMSE* vs. computation time for the Mackey-Glass problem.
(d) *NMSE* vs. computation time for the hand-written letter recognition problem.

the first stage is equal to $N_p T_{KM}$, where T_{KM} stands for the training-cycle time of *K-Means*. The computation time for the second stage is equal to T_{Wd}, the time for computing the widths of Gaussian functions. The computation time for the third stage is equal to $N_p T_{Ht}$, where T_{Ht} stand for the training-cycle time of *Height-I*. The total computation time of the *RBF* architecture is thus equal to

$$\text{computation time} = N_p(T_{KM} + T_{Ht}) + T_{Wd}, \tag{6.2}$$

Figure figure 6.1d shows the *NMSE* of the *Het*, *RBF*, and *BP* architectures with respect to the *training-cycle* time of *Het-II* for the letter recognition problem. The computation times of these three architectures are calculated in the similar manner to their counterpart in the figure 6.1c.

The results shown in figure 6.1c and 6.1d indicate that *Het* has much better performance than *RBF* and *BP*, when measured in terms of the computation time. Comparing figure 6.1a and 6.1c, reveals that the convergence rate of *Het* is much faster than those of *RBF* and *BP* when measured by the computation time for the Mackey-Glass problem. For the letter recognition problem, figure 6.1b and 6.1d show that the difference between the convergence rates of *Het* and *BP* becomes smaller when measured by the computation time than when measured by the number of pattern presentations. However, the difference between the convergence rates of *Het* and *RBF* becomes much larger when measured by the computation time than when measured by the number of pattern presentations.

Chapter 7

Parallel Implementation and Complexity Evaluation

In this chapter we investigate the parallel implementations of the heterogeneous architecture based on error-weighted k-means partitioning *(Het)*, the radial basis function *(RBF)* architecture, and the back-propagation *(BP)* architecture. The parallel implementations of these three architectures are described in sections 7.1, 7.2, and 7.3. These implementations are restricted to those that capture the maximal parallelism of the architectures. In section 7.4, we estimate the hardware cost of each implementation for the Mackey-Glass time series prediction and a hand-written capital letter recognition task. The hardware cost of an implementation is approximately defined as the silicon area required for realizing all the operators in the implementation. In section 7.5, we determine for each architecture the computation time needed to complete one training cycle in a maximally concurrent manner for both the Mackey-Glass and letter recognition problems. Finally the complexities of these implementations are compared in section 7.6.

7.1 Heterogeneous Architectures

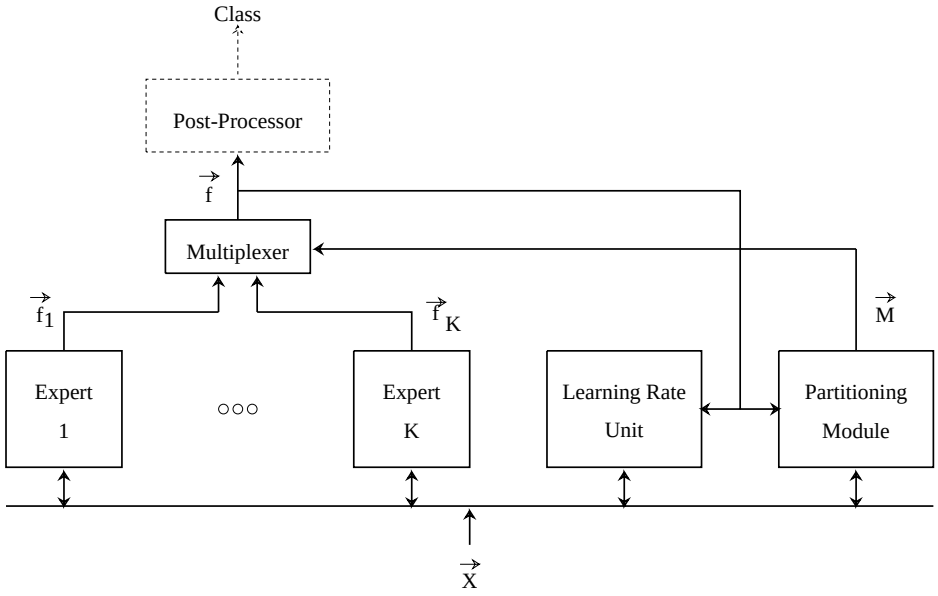


Figure 7.1: The block diagram of the parallel implementation of the heterogeneous architecture based on error-weighted k-means partitioning.

This section describes a parallel implementation of the heterogeneous architecture based on error-weighted k-means partitioning (*Het*) which represents a piecewise linear mapping from R^M to R^N. The block diagram of this implementation, as shown in figure 7.1, consists of the following functional blocks:

- a partitioning module, for generating membership functions $M_k(\vec{x})$;
- *K* expert modules, each with linear network function $\vec{f_k}$;
- a learning rate unit, for calculating the learning rates η_{km} and η_{lms};
- a multiplexer, for generating the system output $\vec{f}(\vec{x}) = M_k(\vec{x})\vec{f_k}(\vec{x})$;
- and a post processor, for determining the class indicated by the network output.

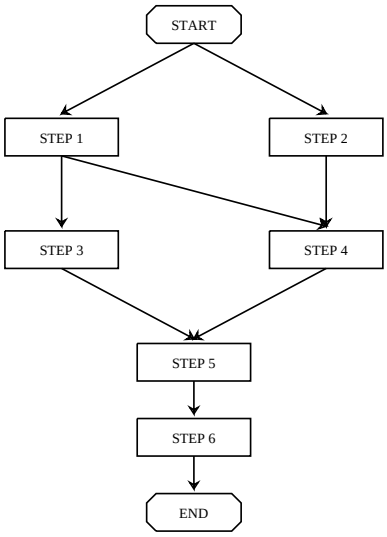
Note that a post processor is only needed when used in a classification task. The detail implementations of these building blocks are derived from the algorithms performing the corresponding tasks. We analyze each of the building blocks for its hardware requirement, measured by the number of arithmetic operators used in the implementation, and its computation time, defined as the minimal time needed to perform all necessary arithmetic operations in its algorithm in a maximally concurrent manner.

7.1.1 The Learning Rate Unit

The task of the learning rate unit is to calculate the learning rates η_{km} and η_{lms} according to the equations:

$$\eta_{km} = \ln(\sum_{k=1}^{K} \hat{\nu}_k) + \{\sum_{k=1}^{K} -\hat{\nu}_k \ln \hat{\nu}_k\} / \{\sum_{k=1}^{K} \hat{\nu}_k\} \qquad \text{and} \qquad \eta_{lms} = \eta_{lms} + 0.01. \tag{7.1}$$

The learning rate unit computes these two learning rates using the the procedure illustrated by the flowchart in figure 7.2.



Step 1: compute $t_1 = \sum_{k=1}^{K} \hat{\nu}_k$.
Step 2: compute $t_2 = \sum_{k=1}^{K} \hat{\nu}_k \ln \hat{\nu}_k$.
Step 3: compute $t_3 = \ln(t_1)$.
Step 4: compute $t_4 = t_2/t_1$.
Step 5: set $\eta_{km} = t_3 - t_4$.
Step 6: set $\eta_{lms} = \eta_{km} + 0.01$.

Figure 7.2: Flow chart of the computation performed by the learning rate unit.

The step-by-step analysis of this procedure is provided in the following.

Analysis

Step 1: To compute $\sum_{k=1}^{K} \hat{\nu}_k$, we use $K-1$ adders arranged in the form of a binary tree with $\lceil K/2 \rceil$ leaves, as shown in figure 7.3. The notation $\lceil r \rceil$ represents the smallest integer larger than or equal to r. This summing operation requires $\lceil \log_2 K \rceil$ addition time. One addition time is defined to be the time needed for performing one addition.

Step 2: This step needs K multipliers, K log-circuits and a binary tree of $K-1$ adders. The time needed to perform this step is 1 log-time plus 1 multiplication time plus $\lceil \log_2 K \rceil$ addition times.

Step 3: This step needs 1 log-circuit and is completed in 1 log-time, the time required for computing one logarithm function.

Step 4: This step needs 1 division circuit and is completed in 1 division time.

Step 5: This step needs 1 subtractor and is completed in 1 subtraction time.

Step 6: This step needs 1 adder and is completed in 1 addition time.

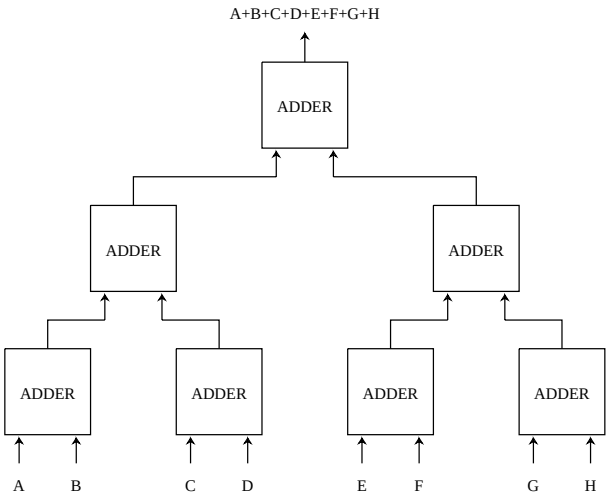


Figure 7.3: The block diagram of a adder tree with 4 leaves.

Based on this analysis, we determine the number of operators required by the learning rate unit by summing up all the operators in each step. We also determine the time needed by the learning rate unit to complete its computation. This execution time is equal to the time of the critical path in the computation. For this specific case, the critical path consists of steps 2, 4, 5 and 6. Similar to chapter 6, we categorize these operations based on their hardware and time complexities into 4 categories:

- comparison;
- addition: which includes subtraction;
- multiplication: which includes division and sqauring operation;

- and non-linear function computation: which includes the computations of sigmoid, exponential, and logarithm functions.

The resultant hardware requirement and execution time are

Hardware requirement: $2K$ adder, $K+1$ multipliers, and $K+1$ log circuit.

Computation time: $2 + \lceil \log_2 K \rceil$ addition times plus 2 multiplication times plus 1 log time.

7.1.2 The Partitioning Module

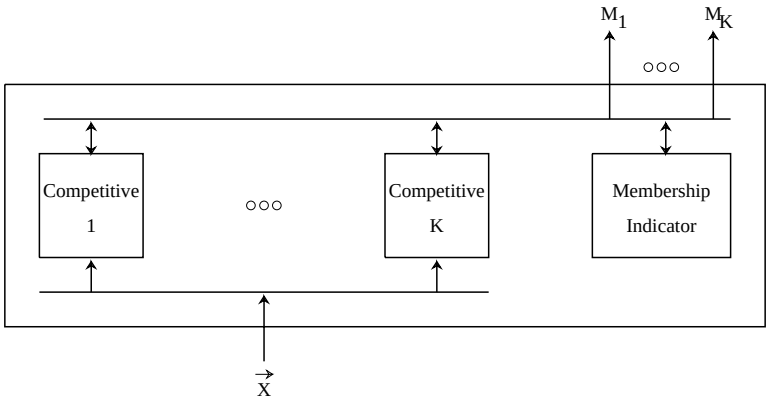
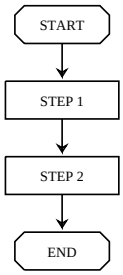


Figure 7.4: The block diagram for the parallel implementation of the partitioning module.

Figure 7.4 shows the block diagram of the partitioning module that partitions the input domain into K regions. The module consists of a membership indicator and K competitive units. The task specifications of these blocks, as well as the analysis of their hardware requirements and execution times, are described in the following.

7.1.2.1 A Membership Indicator

A membership indicator determines the membership functions M_1,..., M_K based on the error-weighted deviations $\hat{\epsilon}_k \hat{\nu}_k \|\vec{x} - \vec{c}_k\|^2$ generated by the K competitive units. Its task can be described by the flowchart in figure 7.5.



Step 1: find index k_w such that
$\hat{\epsilon}_{k_w}\hat{\nu}_{k_w}\|\vec{x}-\vec{c}_{k_w}\|^2 \leq \hat{\epsilon}_k\hat{\nu}_k\|\vec{x}-\vec{c}_k\|^2$,
for $1 \leq k \leq K$.

Step 2: set $M_{k_w}(\vec{x})$ to 1 and other $M_k(\vec{x})$ to 0.

Figure 7.5: Flow chart of the computation performed by the membership indicator.

Analysis

Step 1: This step needs $K-1$ comparators. By arranging these comparators in the form of a binary tree with $\lceil K/2 \rceil$ leaves, we can achieve these comparisons in $\lceil \log_2 K \rceil$ comparison times.

Step 2: Since it takes a comparatively short time to open and close the switch according to $M_k(\vec{x})$, we assume that there is no delay in this step.

Hardware requirement: $K-1$ comparators.

Computation time: $\lceil \log_2 K \rceil$ comparison times.

7.1.2.2 A Competitive Unit

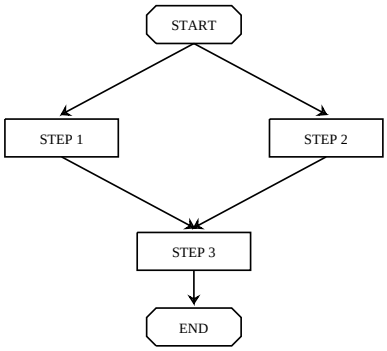
A competitive unit is composed of 3 sub-units:

- a sub-unit for calculating an error-weighted deviation between $\vec{x}$ and $\vec{c}_k$,
- a sub-unit for updating $\vec{c}_k$ and $\hat{\nu}$,
- and a sub-unit for updating $\hat{\epsilon}_k$.

7.1.2.2.1 A Sub-Unit for Calculating an Error-Weighted Deviation

The function of this sub-unit is to calculate the error-weighted deviation between $\vec{x}$ and $\vec{c}_k$, using

the procedure described by the flowchart in figure 7.6.



Step 1: compute $t_1 = \|\vec{x} - \vec{c}_k\|^2$.
Step 2: compute $t_2 = \hat{\epsilon}_k \hat{\nu}_k$.
Step 3: compute $t_1 t_2$.

Figure 7.6: Flow chart for computing the error-weighted deviation.

Analysis

Step 1: The detail algorithm for computing $\|\vec{x} - \vec{c}_k\|^2$ is as follows:

Step 1.1: for $i = 1$ to M do in parallel

compute $(x_i - c_{i,k})^2$

Step 1.2: compute $\sum_{i=1}^{M}(x_i - c_{i,k})^2$.

where x_i and $c_{i,k}$ denote the i-th component of $\vec{x}$ and $\vec{c}_k$ respectively.

Step 1.1 needs *M* adders and *M* multipliers, and is completed in 1 addition time plus 1 multiplication time.

Step 1.2 needs $M-1$ adders. By connecting these adders as a binary tree with $\lceil M/2 \rceil$ leaves, this step is completed in $\lceil \log_2 M \rceil$ addition times.

Step 2: This step needs 1 multiplier and is completed in 1 multiplication time.

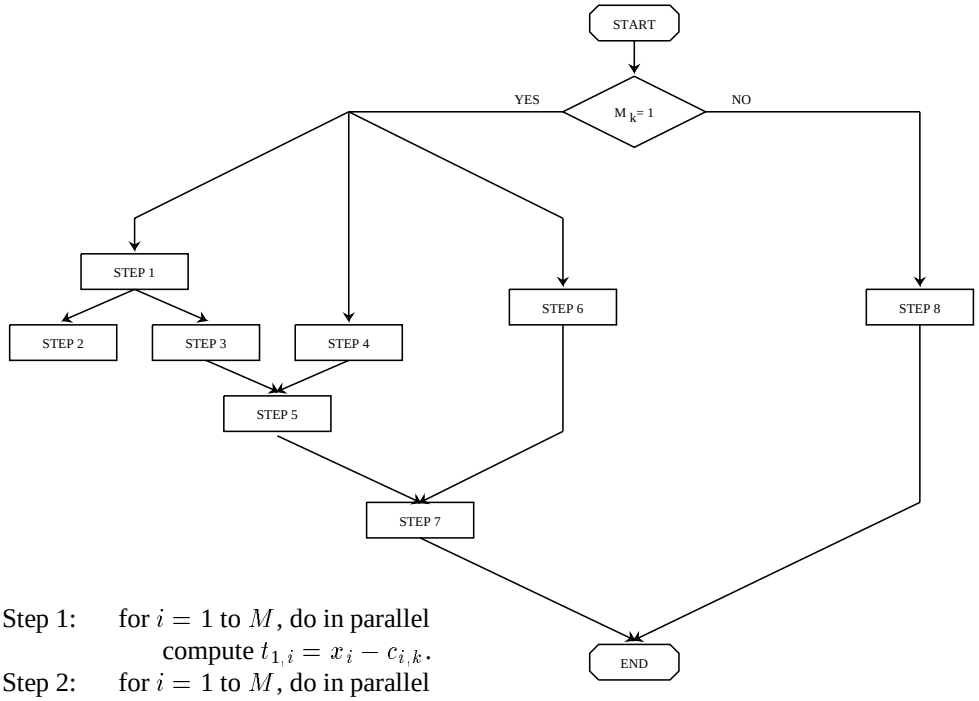
Step 3: This step needs 1 multiplier and is completed in 1 multiplication time.

Hardware requirement: $2M-1$ adders and $M+2$ multipliers.

Computation time: $1 + \lceil \log_2 M \rceil$ addition times plus 2 multiplication times.

7.1.2.2.2 A Sub-Unit for Updating $\vec{c}_k$ and $\hat{\nu}_k$

The function of this sub-unit is to adjust $\vec{c}_k$ and $\hat{\nu}_k$ according to the flowchart shown in figure 7.7.



Step 1: for $i = 1$ to M, do in parallel
compute $t_{1,i} = x_i - c_{i,k}$.
Step 2: for $i = 1$ to M, do in parallel
set $c_{i,k} = c_{i,k} + \eta_{km} d_i$.
Step 3: compute $t_2 = \sum_{i=1}^{M} d_i^2$.
Step 4: compute $t_3 = \beta \hat{\epsilon}_k$.
Step 5: compute $t_4 = t_2 t_3$.
Step 6: compute $t_5 = \alpha \hat{\nu}_k$.
Step 7: set $\hat{\nu}_k = t_4 + t_5$.
Step 8: set $\hat{\nu}_k = \alpha \hat{\nu}_k$.

Figure 7.7: Flow chart for updating $\vec{c}_k$ and $\hat{\nu}_k$ of the partitioning module.

Analysis

Step 1: This step needs M adders and is completed in 1 addition time.

Step 2: This step needs M adders and M multipliers, and is completed in 1 addition time plus 1 multiplication time.

Step 3: This step needs $M-1$ adders and M multipliers and is completed in $\lceil \log_2 M \rceil$ addition times plus 1 multiplication times.

Step 4: This step needs 1 multiplier and is completed in 1 multiplication time.

Step 5: This step needs 1 multiplier and is completed in 1 multiplication time.

Step 6: This step needs 1 multiplier and is completed in 1 multiplication time.

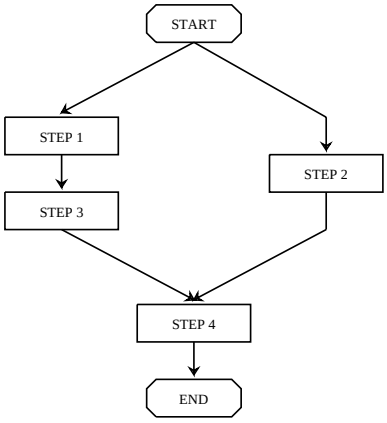
Step 7: This step needs 1 adders and is completed in 1 addition time.

Step 8: This step needs 1 multiplier and is completed in 1 multiplication time.

Hardware requirement:	$3M$ adders and $2M+4$ multipliers.
Computation time:	$2 + \lceil \log_2 M \rceil$ addition time plus 2 multiplication time if $M_k(\vec{x}) = 1$, 1 multiplication time if $M_k(\vec{x}) = 0$.

7.1.2.2.3 A Sub-Unit for Updating $\hat{\epsilon}$

The function of this sub-unit is described by the flowchart in figure 7.8.



Step 1: compute $t_1 = \sum_{i=1}^{N} \{f_{k,i}(\vec{x}) - g_{k,i}(\vec{x})\}^2$
Step 2: compute $t_2 = \alpha \hat{\epsilon}_k$.
Step 3: compute $t_3 = \beta t_1$.
Step 4: set $\hat{\epsilon}_k = t_2 + t_3$.

Figure 7.8: Flowchart for updating error $\hat{\epsilon}_k$.

Analysis

Step 1: The sub-unit recieves the differences $\{f_{k,i}(\vec{x}) - g_{k,i}(\vec{x})\}$ from the expert module indicated by M_k. It thus needs to wait 1 addition time for the expert module to compute these values. To calculate t_1, this sub-unit uses $N-1$ adders and N multipliers and completes this computation in $\lceil \log_2 N \rceil$ addition times plus 1 multiplication time.

Step 2: This step needs 1 multipliers and is completed in 1 multiplication time.

Step 3: This step needs 1 multipliers and is completed in 1 multiplication time.

Step 4: This step needs 1 adder and is completed in 1 addition time.

Hardware requirement: N adders and $N+2$ multipliers.

Computation time: $2 + \lceil \log_2 N \rceil$ addition times plus 2 multiplication times.

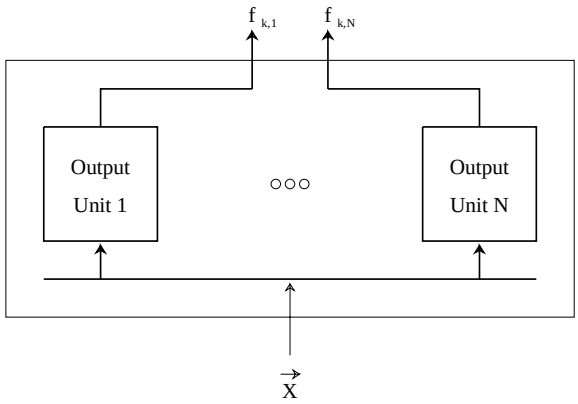


Figure 7.9: The block diagram for the parallel implementation of an expert module.

7.1.3 A Linear Expert Module

Figure 7.9 shows the block diagram of a linear expert module that represents a mapping from $\vec{x}$ to R^N. The system is composed of N output units, each associated with one output dimension.

7.1.3.1 An Output Unit

The task of output unit n is to compute function $f_{k,n}$ and update the corresponding parameters $\vec{w}_{k,n}$. This output unit is composed of 2 sub-units:

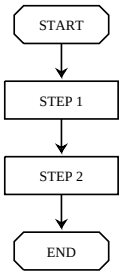
- a sub-unit for computing function $f_{k,n}$,
- and a sub-unit for updating $\vec{w}_{k,n}$.

7.1.3.1.1 A Sub-Unit for Computing Function $f_{k,n}$

The task of this sub-unit is to compute the linear function:

$$f_{k,n} = w_{k,n0} + \sum_{i=1}^{M} w_{k,ni} x_i, \tag{7.2}$$

where M is the dimension of an input. This task is described by the flowchart in figure 7.10.



Step 1: for $i = 1$ to M, do in parallel
compute $t_{1,i} = w_{k,ni}x_i$.
Step 2: compute $w_{k,n0} + \sum_{i=1}^{M} t_{1,i}$.

Figure 7.10: Flow chart for computing the output $f_{k,n}(\vec{x})$.

Analysis

Step 1: This step needs *M* multipliers and is completed in 1 multiplication time.

Step 2: This step needs *M* adders and is completed in $\lceil \log_2(M+1) \rceil$ addition times.

Hardware requirement: *M* adders and *M* multipliers.

Computation time: $\lceil \log_2(M+1) \rceil$ addition times plus 1 multiplication time.

7.1.3.1.2 A Sub-Unit for Updating $\vec{w}_{k,n}$

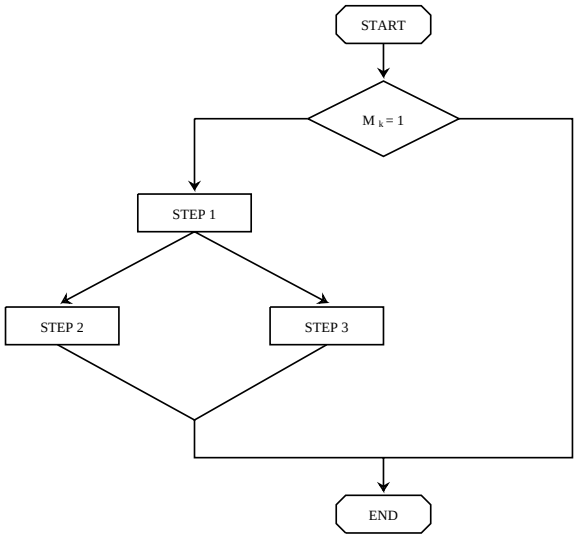
This sub-unit adjusts parameter $\vec{w}_{k,n}$ using the *LMS* algorithm, described by the flowchart in figure 7.11.

Analysis

Step 1: This step needs 1 adder and 1 multiplier, and is completed in 1 addition time plus 1 multiplication time.

Step 2: This step needs 1 adder and is completed in 1 addition time.

This step needs needs *M* adder and *M* multipliers, is completed in 1 addition time plus 1 multiplication time.



Step 1: compute
$t_1 = \eta_{lms}(g_{k,n}(\vec{x}) - f_{k,n}(\vec{x}))$
Step 2: set $w_{k,n0} = w_{k,n0} + t_1$.
Step 3: for $i = 1$ to M do in parallel
set $w_{k,ni} = w_{k,ni} + t_1$.

Figure 7.11: Flow chart for updating parameter $\vec{w}_{k,n}$.

Hardware requirement: $M+2$ adders and $M+1$ multipliers.

Computation time: 2 addition times plus 2 multiplication times.

7.1.4 A Post-Processor

In a classification task, a *Het* architecture needs a post-processor to determine the category of the input $\vec{x}$. A post-processor first finds the maximum value of the output $f_1(\vec{x})$, ... , $f_N(\vec{x})$, and then defines the category of the input to be the one corresponding to the maximum value. Its hardware requirement and the computation time are

Hardware requirement: $N-1$ comparators.

Computation time: $\lceil \log_2 N \rceil$ comparison times.

7.2 Radial Basis Function Architectures

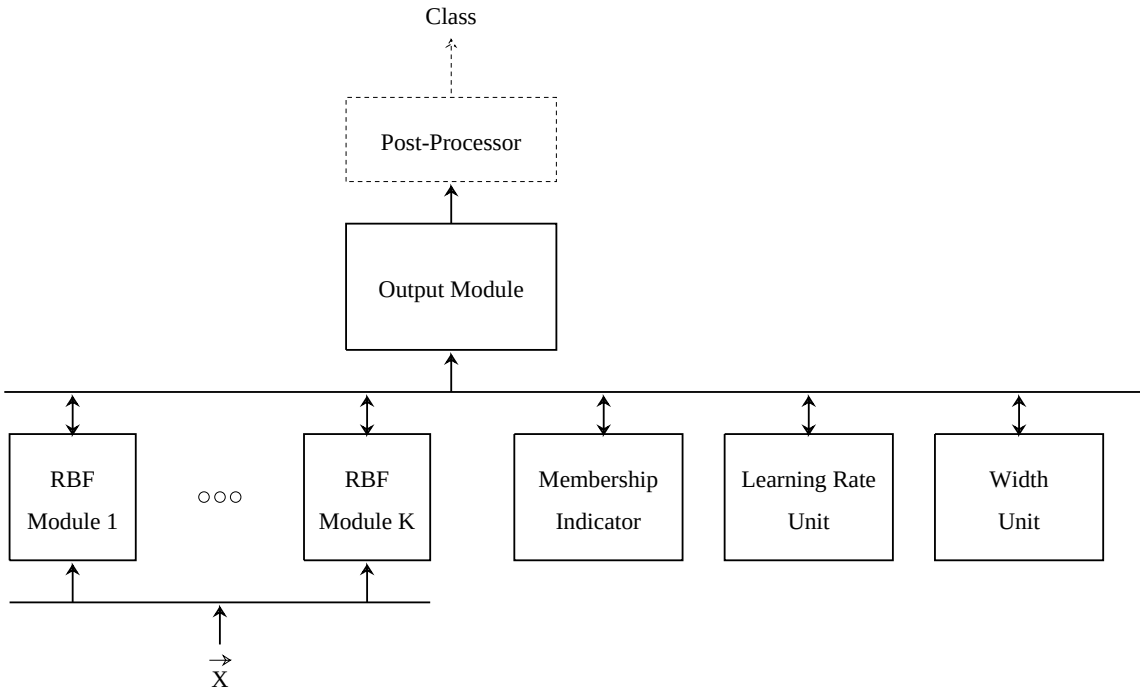


Figure 7.12: The block diagram of the parallel implementation of the radial basis function architecture.

Figure 7.12 shows the block diagram of the implementation of a radial basis function architecture which represents a mapping from R^M to R^N. This implementation, as illustrated by the figure, is composed of the following building blocks:

- *K* RBF modules, each for implementing a Gaussian radial basis function;
- an output module, for generating the output vector;
- a membership indicator, for determining the closest Gaussian center to the input;
- a learning rate unit, for calculating learning rate η_{km};
- a width unit, for calculating the widths of the radial basis functions;

- and a post-processor, for determining the category of the input in a classification task.

In the following, we provide the detail specifications of these modules, as well as the analysis of their hardware requirements and their execution times.

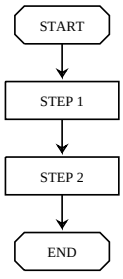
7.2.1 A RBF Module

A RBF module is composed of 3 units:

- a unit for computing the Euclidean devaition $\|\vec{x} - \vec{c}_k\|^2$,
- a unit for computing Gaussian function $\exp(-\|\vec{x} - \vec{c}_k\|^2/\sigma^2)$,
- and a unit for updating $\vec{c}_k$ and $\hat{v}$.

7.2.1.1 A Unit for Computing the Euclidean Deviation

The function of this unit is to compute the Euclidean deviation between $\vec{x}$ and $\vec{c}_k$, using the procedure described by the flowchart in figure 7.13.



Step 1: for $i = 1$ to M do in parallel
compute $(x_i - c_{i,k})^2$
Step 2: compute $\sum_{i=1}^{M}(x_i - c_{i,k})^2$.

where x_i and $c_{i,k}$ denote the i-th component of $\vec{x}$ and the Gaussian center $\vec{c}_k$ respectively.

Figure 7.13: Flow chart for computing the Euclidean deviation.

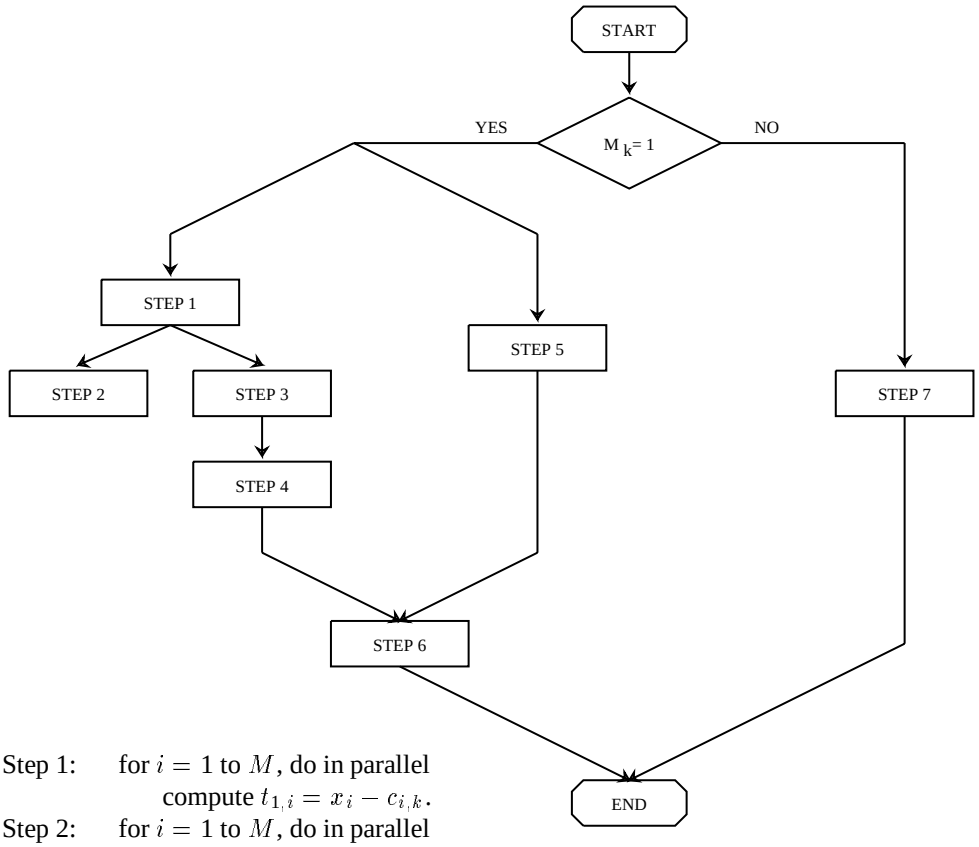
Hardware requirement: $2M-1$ adders and M multipliers.

Computation time: $1 + \lceil \log_2 M \rceil$ addition time plus 1 multiplication time.

7.2.1.2 A Unit for Computing Gaussian Function

This unit computes the Guassian function $\exp(-\|\vec{x} - \vec{c}_k\|^2/\sigma^2)$ using the Euclidean deviation generated by the unit for computing the Euclidean deviation. Its hardware is composed of 1 multiplier and 1 exponential circuit, and its computation is completed in 1 multiplication time plus 1 exponential time.

7.2.1.3 A Unit for Updating $\vec{c}_k$ and $\hat{v}_k$



Step 1: for $i = 1$ to M, do in parallel
compute $t_{1,i} = x_i - c_{i,k}$.
Step 2: for $i = 1$ to M, do in parallel
set $c_{i,k} = c_{i,k} + \eta_{km} d_i$.
Step 3: compute $t_2 = \sum_{i=1}^{M} d_i^2$.
Step 4: compute $t_3 = \beta t_2$.
Step 5: compute $t_4 = \alpha \hat{\nu}_k$.
Step 6: set $\hat{\nu}_k = t_3 + t_4$.
Step 7: set $\hat{\nu}_k = \alpha \hat{\nu}_k$.

Figure 7.14: Flow chart for updating $\vec{c}_k$ and $\hat{v}_k$ of the k-means algorithm.

Figure 7.14 shows the procedure used by this sub-unit to adjust $\vec{c}_k$ and $\hat{v}_k$ of the k-means algorithm. This procedure is similar to the one described in subsection 7.1.2.2.2. The hardware requirement and computation time of this procedure are:

Hardware requirement: $3M$ adders and $2M+3$ multipliers.

Computation time: $2 + \lceil \log_2 M \rceil$ addition time plus 2 multiplication time if $M_k(\vec{x}) = 1$,
1 multiplication time if $M_k(\vec{x}) = 0$.

7.2.2 An Output Module

Figure 7.15 depicts the schematic diagram of an output module. It illustrates the following building blocks:

- *N* output units, each corresponding to one output dimension;
- and a normalizing unit.

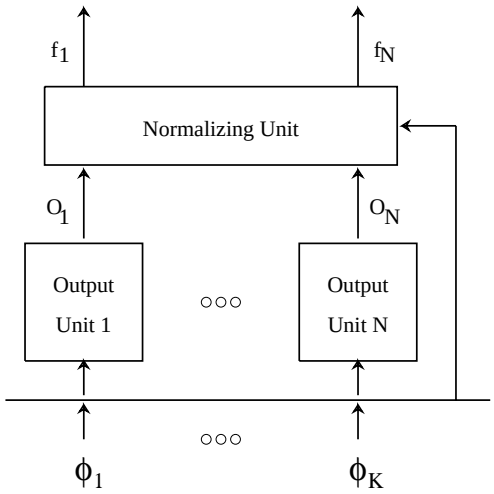


Figure 7.15: The block diagram of an output module.

7.2.2.1 An Output Unit

An output unit is composed of 2 sub-units:

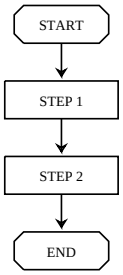
- a sub-unit for computing o_n,
- and a sub-unit for updating $\vec{w}_n$.

7.2.2.1.1 A Sub-Unit for Computing Output o_n

The task of this sub-unit is to compute the function:

$$o_n = \sum_{k=1}^{K} w_{k,n}\phi_k, \tag{7.3}$$

where ϕ_k is the output of the k-th RBF unit and $w_{k,n}$ is the corresponding amplitude coefficient. The procedure for computing the output o_n, which is essentially similar to the procedure described in subsection 7.1.3.1.1, is shown in figure 7.16.



Step 1: for $k = 1$ to K, do in parallel
compute $w_{k,n}\phi_k$.
Step 2: compute $\sum_{k=1}^{K} w_{k,n}\phi_k$.

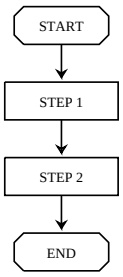
Figure 7.16: Flow chart for computing the output $f_{k,n}(\vec{x})$.

Hardware requirement: $K-1$ adders and K multipliers.

Computation time: $\lceil \log_2 K \rceil$ addition times plus 1 multiplication time.

7.2.2.1.2 A Sub-Unit for Updating $\vec{w}_n$

This sub-unit adjusts the amplitude coefficients $\vec{w}_n$ using the *LMS* algorithm. The procedure used in this sub-unit, which is essentially similar to the procedure described in subsection 7.1.3.1.2, is shown in figure 7.17.



Step 1: compute $t_1 = \eta_{lms}\{g_n(\vec{x}) - f_n(\vec{x})\}$
Step 2: for $k = 1$ to K do in parallel
set $w_{n,k} = w_{n,k} + phi_k t_1$.

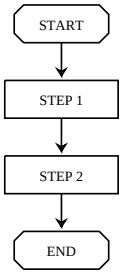
Figure 7.17: Flow chart for updating the parameters in the n-th output unit.

Hardware requirement: $K+1$ adders and $K+1$ multipliers.

Computation time: 2 addition times plus 2 multiplication times.

7.2.2.2 A Normalizing Unit

The task of a normalized unit is to divide the output o_n of each output unit with the sum of all ϕ_k. The computation in this task is described by the flowchart shown in figure 7.18.



Step 1: compute $t_1 = \sum_{k=1}^{K} \phi_k$.
Step 2: for $n = 1$ to N do in parallel
compute $f_n(\vec{x}) = o_n(\vec{x})$.

Figure 7.18: Flow chart for normalizing the output of the output module.

Analysis

Step 1: This step needs $K-1$ adders connected in a binary tree with $\lceil K/2 \rceil$ leaves. The process is completed in $\lceil \log_2 K \rceil$ addition times.

Step 2: This step needs *N* division circuits and is completed in 1 division time.

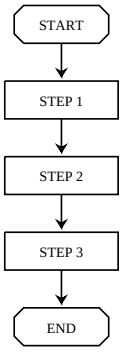
Since the hardware and time complexities of division and multiplication are essentially the same, we substitute the division operation with the multiplication operation.

Hardware requirement: $K-1$ adders and *N* multipliers.

Computation time: $\lceil \log_2 K \rceil$ addition times plus 1 multiplication time.

7.2.3 A Membership Indicator

A membership indicator determines the membership functions M_1,..., M_K based on the Euclidean deviations $\|\vec{x} - \vec{c}_k\|^2$, generated by the *K* RBF units. This determination is described by the flowchart in figure 7.19.



Step 1: for $k = 1$ to K do in parallel
multiply $\hat{v}_k$ to $\|\vec{x} - \vec{c}_k\|^2$.
Step 2: find index k_w such that
$\hat{v}_{k_w}\|\vec{x} - \vec{c}_{k_w}\|^2 \leq \hat{v}_k\|\vec{x} - \vec{c}_k\|^2$,
for $1 \leq k \leq K$.
Step 3: set $M_{k_w}(\vec{x})$ to 1 and other $M_k(\vec{x})$ to 0.

Figure 7.19: Flow chart for normalizing the output of the output module.

Analysis

Step 1: This step needs K multipliers and is completed in 1 multiplication time.

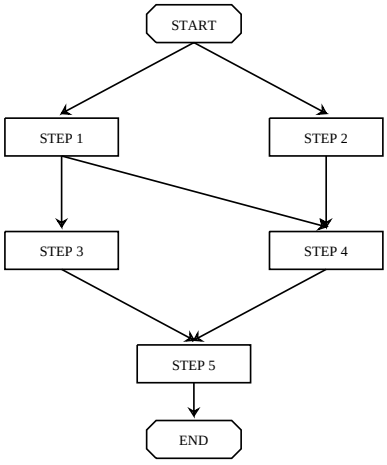
Step 2: This step needs $K-1$ comparators, and is completed in $\lceil \log_2 K \rceil$ comparison times.

Step 3: Since it takes comparatively short time to open or close the switch according to $M_k(\vec{x})$, we assume that there is no delay in this step.

Hardware requirement: $K-1$ comparators and K multipliers.

Computation time: $\lceil \log_2 K \rceil$ comparison times plus 1 multiplication time.

7.2.4 A Learning Rate Unit



Step 1: compute $t_1 = \sum_{k=1}^{K} \hat{\nu}_k$.
Step 2: compute $t_2 = \sum_{k=1}^{K} \hat{\nu}_k \ln \hat{\nu}_k$.
Step 3: compute $t_3 = \ln(t_1)$.
Step 4: compute $t_4 = t_2/t_1$.
Step 5: set $\eta_{km} = t_3 - t_4$.

Figure 7.20: Flow chart for computing the learning rate η_{km}.

The learning rate unit calculates the learning rate η_{km}:

$$\eta_{km} = ln(\sum_{k=1}^{K} \hat{v}_k) + \{\sum_{k=1}^{K} -\hat{v}_k ln\, \hat{v}_k\} / \{\sum_{k=1}^{K} \hat{v}_k\}, \tag{7.4}$$

using the procedure shown in figure 7.20. This procedure is similar to the one described in subsection 7.1.1.

Hardware requirement: $2K-1$ adders, $K+1$ multipliers, and $K+1$ log-circuit.

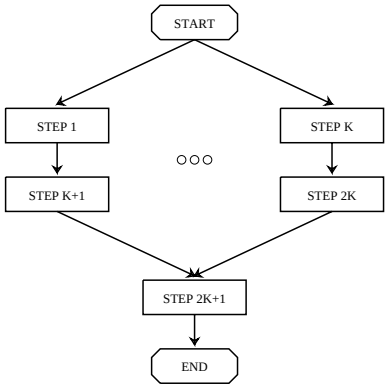
Computation time: $\lceil \log_2 K \rceil$ addition times plus 2 multiplication times plus 1 log-time.

7.2.5 A Width Unit

This unit is used to the width of the Gaussian function. The width σ of all the Gaussian functions are defined to

$$\{1/K \sum_{k=1}^{K} \|\vec{c}_k - \vec{c}_{k,nearest}\|^2\}^{1/2}. \tag{7.5}$$

The procedure for computing this width is illustrated by the flowchart in figure 7.21. **Analysis**



Step 1:	for $k = 1$ to K and $k \neq 1$ do in parallel compute $\|\vec{c}_1 - \vec{c}_k\|^2$
⋮	⋮
Step K:	for $k = 1$ to K and $k \neq K$ do in parallel compute $\|\vec{c}_K - \vec{c}_k\|^2$
Step K+1:	find $d_1^2 =$ the minimum of $\|\vec{c}_1 - \vec{c}_k\|^2$ for $1 \leq k \leq K$ and $k \neq 1$.
⋮	⋮
Step 2K:	find $d_K^2 =$ the minimum of $\|\vec{c}_K - \vec{c}_k\|^2$ for $1 \leq k \leq K$ and $k \neq K$.
Step 2K+1:	compute $\sigma^2 = (\sum_{k=1}^{K} d_k^2)/K$.

Figure 7.21: Flow chart for determining the width of the Gaussian radial basis function.

Step 1: This step needs $2KM-K-2M+1$ adders and $KM-M$ multipliers, and is completed in $1+\lceil \log_2 M \rceil$ addition times plus 1 multiplication time.

⋮ ⋮

Step K: This step needs $2KM-K-2M+1$ adders and $KM-M$ multipliers, and is completed in $1+\lceil \log_2 M \rceil$ addition times plus 1 multiplication time.

Step K+1: This step needs $K-2$ comparators, and is completed in $\lceil \log_2 (K-1) \rceil$ comparison times.

$\vdots$ $\vdots$

Step 2K: This step needs $K-2$ comparators, and is completed in $\lceil \log_2 (K-1) \rceil$ comparison times.

Step 2K+1: This step needs $K-1$ adders and 1 division circuit (multiplier), and is completed in $\lceil \log_2 K \rceil$ addition times plus 1 division (multiplication) time.

Hardware requirement: $2K^2M-K^2-2KM+2K-1$ adders, $K^2M-MK+1$ multipliers, and K^2-2K comparators.

Computation time: $\lceil \log_2 (K-1) \rceil$ comparison time plus $1+\lceil \log_2 K \rceil + \lceil \log_2 M \rceil$ addition times plus 2 multiplication time.

7.2.6 A Post-Processor

In a classification task, a *RBF* architecture needs a post-processor to determine the category of the input $\vec{x}$. A post-processor first finds the maximum value of the output $f_1(\vec{x})$, ... , $f_N(\vec{x})$, and then defines the category of the input to be the one corresponding to the maximum value. Its hardware requirement and the computation time are listed in the following summary:

Hardware requirement: $N-1$ comparators.

Computation time: $\lceil \log_2 N \rceil$ comparison times.

7.3 Back Propagation Architectures

This section describes the parallel implementations for two classes of back-propagation architectures: one for function approximation and the other for classification. The former is a general case of the architecture used in the Mackey-Glass problem described in section 5.3, and the latter is a general case of the architecture used in the hand-written character recognition problem described in section 5.4.

7.3.1 Function Approximation

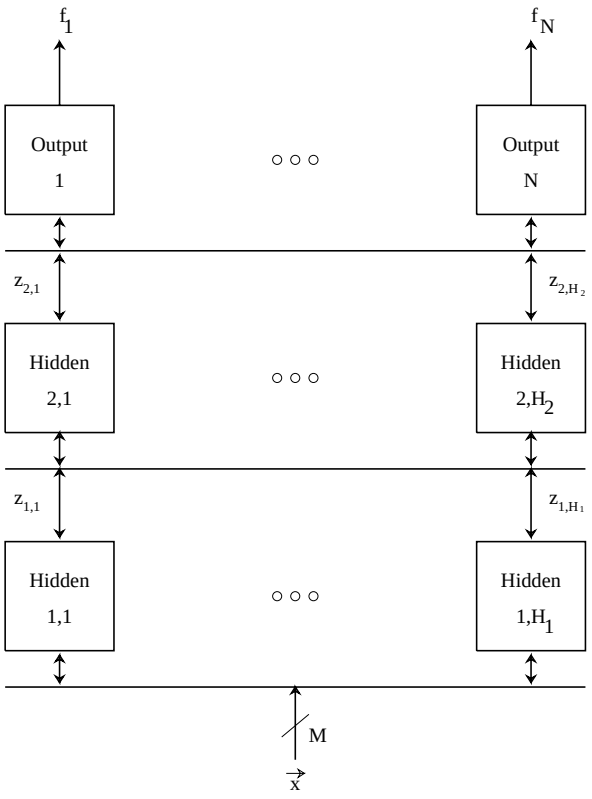


Figure 7.22: The block diagram of the parallel implementation of a back-propagation architecture with two hidden layers.

Figure 7.22 shows the implementation of an architecture for function approximation. This implementation is composed of

- H_1 first-hidden-layer units,
- H_2 second-hidden-layer units,
- and N linear output units.

The implementations of these units, including their analyses, are provided in the following three subsections.

7.3.1.1 A First-Hidden-Layer Unit

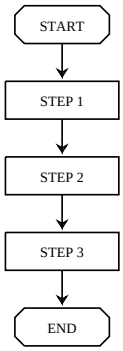
The implementation of this unit is composed of a sub-unit for computing its output $z_{1,i}$ and for updating its parameter $\vec{w}_{1,i}$.

7.3.1.1.1 A Sub-Unit for Computing $z_{1,i}$

This sub-unit receives an input $\vec{x} \in R^M$ and generates a scalar output:

$$z_{1,i}(\vec{x}) = s(w_{1,i0} + \sum_{j=1}^{M} w_{1,ij} x_j), \tag{7.6}$$

where s is a sigmoid function whose value $s(a)$ is defined to be $\{1 + \exp(-a)\}^{-1}$. The computation performed by this sub-unit is specified by the flowchart in figure 7.23.



Step 1: for $i = 1$ to M do in parallel
compute $w_{1,ij}\,x_j$.
Step 2: compute $a = w_{1,i0} + \sum_{i=1}^{M} w_{1,ij}\,x_j$
Step 3: compute sigmoid value $s(a)$.

Figure 7.23: Flow chart for computing the output of a first-hidden-layer unit.

Analysis

Step 1: This step needs *M* multipliers and is completed in 1 multiplication time.

Step 2: This step needs *M* adders and is completed in $\lceil \log_2(M+1) \rceil$ addition times.

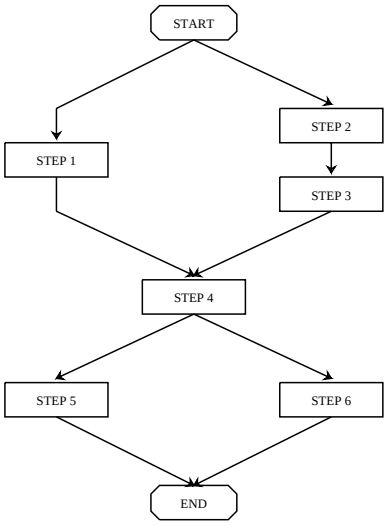
Step 3: This step needs 1 sigmoid-circuit and is completed in 1 sigmoid-time, defined as the time needed for computing a sigmoid function.

Hardware requirement: *M* adders, *M* multipliers, and 1 sigmoid-circuit.

Computation time: $\lceil \log_2(M+1) \rceil$ addition times plus 1 multiplication time plus 1 sigmoid-time.

7.3.1.1.2 A Sub-Unit for Updating $\vec{w}_{1,i}$

The task of this sub-unit is to update parameters $w_{1,i0}$,...,$w_{1,iM}$ based on $\epsilon_{2,1i}$,...,$\epsilon_{2,H_2 i}$ sent by the hidden units in the second layer. The update algorithm is given in the flowchart in figure 7.24.



Step 1: compute $t_1 = \sum_{j=1}^{H_2} \epsilon_{2,j\,i}$.
Step 2: compute $t_2 = 1 - z_{1,i}$.
Step 3: compute $t_3 = z_{1,i} t_2$.
Step 4: compute $\delta_{1,i} = t_1 t_3$.
Step 5: set $w_{1,i0} = w_{1,i0} + \delta_{1,i}$.
Step 6: for $j = 1$ to M do in parallel
set $w_{1,ij} = w_{1,ij} + \delta_{1,i} x_j$.

Figure 7.24: Flow chart for updating the parameters of a first-hidden-layer unit.

Analysis

Step 1: This step needs $H_2 - 1$ adders and is completed in $\lceil \log_2 H_2 \rceil$ addition times.

Step 2: This step needs 1 subtractor (adder) and is completed in 1 subtraction (addition) time.

Step 3: This step needs 1 multiplier and is completed in 1 multiplication time.

Step 4: This step needs 1 multiplier and is completed in 1 multiplication time.

Step 5: This step needs 1 adder and is completed in 1 addition time.

Step 6: This step needs M adders and M multipliers, and is completed in 1 addition time plus 1 multiplication time.

Hardware requirement: $H_2 + M + 1$ adders and $M + 2$ multipliers.

Computation time: 2 addition times plus 3 multiplication times, or $1 + \lceil \log_2 H_2 \rceil$ addition times plus 2 multiplication times, whichever is maximum.

7.3.1.2 A Second-Hidden-Layer Unit

The implementation of this unit is composed of a sub-unit for computing its output $z_{2,i}$ and for updating its parameter $\vec{w}_{2,i}$.

7.3.1.2.1 A Sub-Unit for Computing $z_{2,i}$

This sub-unit receives inputs $z_{1,1}$,..., z_{1,H_1} from the hidden units in the first layer and generates the output of the form:

$$z_{2,i}(\vec{x}) = s(w_{2,i0} + \sum_{i=1}^{H_1} w_{2,ij} z_{1,j}(\vec{x})). \tag{7.7}$$

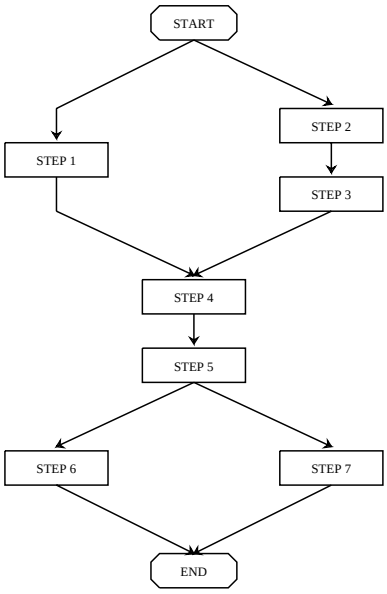
This output $z_{2,i}$ is computed with an algorithm similar to the one described in subsection 7.3.1.1.1. The hardware requirement and the execution time of this sub-unit are given below:

Hardware requirement: H_1 adder, H_1 multipliers, and 1 sigmoid-circuit.

Computation time: $\lceil \log_2(H_1+1) \rceil$ addition times plus 1 multiplication time plus 1 sigmoid-time.

7.3.1.2.2 A Sub-Unit for Updating $\vec{w}_{2,i}$

The task of this sub-unit is to update parameters $w_{2,i0}$,...,w_{2,iH_1} based on $\epsilon_{3,1i}$,...,$\epsilon_{3,Ni}$ sent by the output units. It also generates $\epsilon_{2,i1}$,...,ϵ_{2,iH_1} for updating the parameters in first-layer hidden units. The computation of this sub-unit is specified by the flowchart shown in figure 7.25. Note that this algorithm is similar to the algorithm in section 7.3.1.1.2 except we have added step 5.



Step 1: compute $t_1 = \sum_{j=1}^{N} \epsilon_{3,ji}$.
Step 2: compute $t_2 = 1 - z_{2,i}$.
Step 3: compute $t_3 = z_{2,i} t_2$.
Step 4: compute $\delta_{2,i} = t_1 t_3$.
Step 5: for $j = 1$ to H_1 do in parallel
compute $\epsilon_{2,ij} = w_{2,ij} \delta_{2,i}$.
Step 6: set $w_{2,i0} = w_{2,i0} + \delta_{2,i}$.
Step 7: for $j = 1$ to H_1 do in parallel
set $w_{2,ij} = w_{2,ij} + \delta_{2,i} z_{1,j}$.

Figure 7.25: Flow chart for updating the parameters of a second-hidden-layer unit.

Hardware requirement: H_1+N+1 adders and $2H_1+2$ multipliers.

Computation time: 2 addition times plus 4 multiplication times, or $1+\lceil \log_2 N \rceil$ addition times plus 3 multiplication times, whichever is maximum.

7.3.1.3 A Linear Output Unit

A linear output unit is composed of 2 sub-units:

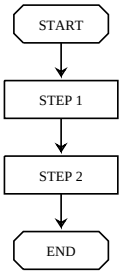
- a sub-unit for computing a linear output f_i,
- and a sub-unit for updating its parameter $\vec{w}_{3,i}$.

7.3.1.3.1 A Sub-Unit for Computing a Linear Output f_i

The task of this sub-unit is to compute the function:

$$f_i(\vec{x}) = w_{3,i0} + \sum_{j=1}^{H_2} w_{3,ij} z_{2,j}(\vec{x}). \tag{7.8}$$

This computation is described by the flowchart in figure 7.26.



Step 1: for $i = 1$ to H_2 do in parallel
compute $w_{3,ij}\, z_{2,j}$.
Step 2: compute $f_n(\vec{x}) = w_{3,n0} + \sum_{i=1}^{H_2} w_{3,ij}\, z_{2,j}$.

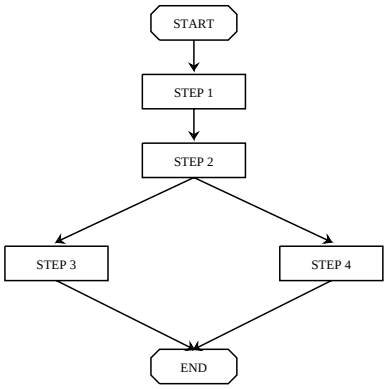
Figure 7.26: Flow chart for computing the output of a linear output unit.

Hardware requirement: H_2 adder and H_2 multipliers.

Computation time: $\lceil \log_2(H_2+1) \rceil$ addition times plus 1 multiplication time.

7.3.1.3.2 A Sub-Unit for Updating $\vec{w}_{3,i}$

The task of this sub-unit is to update parameters $w_{3,i0}$,..., w_{3,iH_2} and generates $\epsilon_{3,i1}$,...,ϵ_{3,iH_2} for updating the parameters of the hidden units in the second layer. The computation of this sub-unit is described in the figure 7.27.



Step 1: compute $\delta_{3,i} = \eta\{g_i(\vec{x}) - f_i(\vec{x})\}$.
Step 2: for $i = 1$ to H_2 do in parallel
compute $\epsilon_{3,ij} = w_{3,ij}\delta_{3,i}$.
Step 3: set $w_{3,i0} = w_{3,i0} + \delta_{3,i}$.
Step 4: for $i = 1$ to H_2 do in parallel
set $w_{3,ij} = w_{3,ij} + \delta_{3,i}z_{2,j}$.

Figure 7.27: Flow chart for updating the parameters of a linear output unit.

Analysis

Step 1: This step needs 1 adder and 1 multiplier, and is completed in 1 addition time plus 1 multiplication time.

Step 2: This step needs H_2 multipliers and is completed in 1 multiplication time.

Step 3: This step needs 1 adder and is completed in 1 addition time.

Step 4: This step needs H_2 adders and H_2 multipliers, and is completed in 1 addition time plus 1 multiplication time.

Hardware requirement: H_2+2 adder, $2H_2+1$ multipliers.

Computation time: 2 addition times plus 3 multiplication times.

7.3.2 Classification

For an architecture that classifies input $\vec{x} \in R^M$ into N categories, its parallel implementation, shown as a block diagram in figure 7.28, is composed of

- H hidden units,
- N sigmoidal output units,
- and a post-processor.

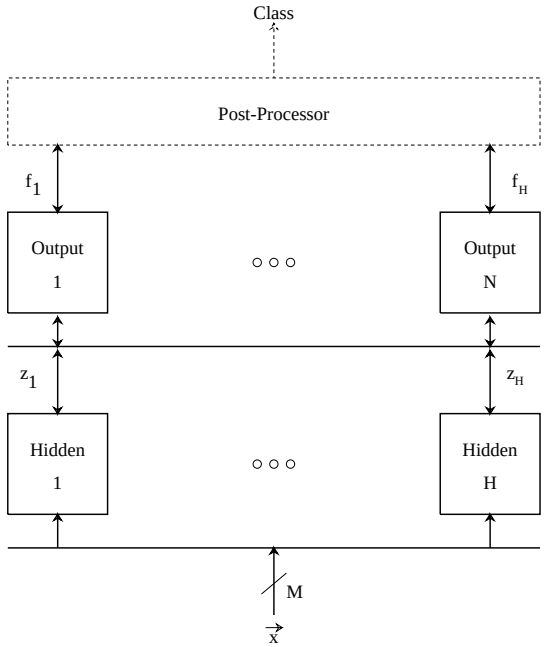


Figure 7.28: The block diagram of the parallel implementation of a back-propagation architecture for classification.

7.3.2.1 A Hidden Unit

The implementation of a hidden unit is identical to that of a first-layer hidden unit described in subsection 7.3.1.1, except for some notation. The hardware requirement and the execution time of a hidden unit are summarized below:

Hardware requirement:	$2M+N+1$ adders, $2M+2$ multipliers, and 1 sigmoid-circuit.
Time for computing output:	$\lceil \log_2(M+1) \rceil$ addition times plus 1 multiplication time plus 1 sigmoid-time.
Time for updating parameters:	2 addition times plus 3 multiplication times, or $1+\lceil \log_2 H_2 \rceil$ addition times plus 2 multiplication times, whichever is maximum.

7.3.2.2 A Sigmoidal Output Unit

A sigmoidal output unit is composed of 2 sub-units:

- a sub-unit for computing a sigmoidal output f_i,
- and a sub-unit for updating its parameter $\vec{w}_{2,i}$.

7.3.2.2.1 A Sub-Unit for Computing a Sigmoidal Output f_i

The task of this sub-unit is to compute the function:

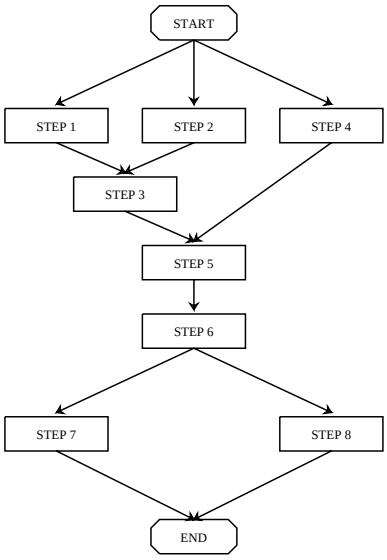
$$f_i(\vec{x}) = s(w_{2,i0} + \sum_{j=1}^{H} w_{2,ij} z_j(\vec{x})), \tag{7.9}$$

where z_j is the output of hidden unit j. The hardware requirement and the execution time for the implementation of this sub-unit are listed below:

Hardware requirement:	H adder, H multipliers, and 1 sigmoid-circuit.
Computation time:	$\lceil \log_2(H+1) \rceil$ addition times plus 1 multiplication time plus 1 sigmoid-time.

7.3.2.2.2 A Sub-Unit for Updating $\vec{w}_{2,i}$

The function of this sub-unit is to update parameters $w_{2,i0}$,..., $w_{2,iH}$, using the algorithm shown in figure 7.29.



Step 1: compute $t_1 = g_i(\vec{x}) - f_i(\vec{x})$.
Step 2: compute $t_2 = 1 - f_i(\vec{x})$.
Step 3: compute $t_3 = t_1 t_2$.
Step 4: compute $t_4 = \eta f_i(\vec{x})$.
Step 5: compute $\delta_{2,i} = t_3 t_4$.
Step 6: for $j = 1$ to H do in parallel
compute $\epsilon_{2,ij} = w_{2,ij}\delta_{2,i}$.
Step 7: set $w_{2,i0} = w_{2,i0} + \delta_{2,i}$.
Step 8: for $j = 1$ to H do in parallel
set $w_{2,ij} = w_{2,ij} + \delta_{2,i} z_j$.

Figure 7.29: Flow chart for updating the parameters of a sigmoidal output unit.

Hardware requirement: $H+3$ adder and $2H+3$ multipliers.

Computation time: 2 addition times plus 4 multiplication times.

7.3.2.3 A Post-Processor

In a classification task, a *BP* architecture needs a post-processor to determine the category of the input $\vec{x}$. A post-processor first finds the maximum value of the output $f_1(\vec{x})$, ... , $f_N(\vec{x})$, and then defines the category of the input to be the one corresponding to the maximum value.

Hardware requirement: $N-1$ comparators.

Computation time: $\lceil \log_2 N \rceil$ comparison times.

7.4 Hardware Cost for Parallel Implementations

In this section, we estimate the *hardware cost* for the parallel implementations of the *Het*, *RBF*, and *BP* architectures described in sections 7.1, 7.2, and 7.3. We define the *hardware cost* of each implementation by the *silicon area* of all the arithmetic operators in the implementation. The silicon area dedicated to the connections among arithmetic operators is not included partly because this area cannot be quantified correctly unless the implementation is actually laid out on silicon, and because approximating the hardware cost by the areas of the operators is accurate enough for the purpose of establishing the *relative* hardware costs of *Het*, *RBF* and *BP* architectures.

To compute the hardware costs for the parallel implementations of *Het*, *RBF* and *BP* architectures, we determine the number of arithmetic operators required in these implementations in subsections 7.4.1, 7.4.2, and 7.4.3. These computational requirements are then translated into the area costs in subsection 7.4.4

7.4.1 Heterogeneous Architecture

The implementation of the *Het* architecture for a function approximation task, as described in section 7.1, is composed of

- a learning rate unit, consisting of $2K$ adders $K+1$ multipliers, and $K+1$ log-circuits;
- a partitioning module, consisting of $5KM+KN-K$ adders, $3KM+KN+8K$ multipliers, and $K-1$ comparators;
- K expert modules, each consisting of $2MN+2N$ adders, and $2MN+N$ multipliers.
- and a multiplexer, whose complexity is assumed to be equivalent to 1 adder.

According to these specifications, the total number of operators in the implementation are

- $K-1$ comparators,
- $2KMN+5KM+3KN+K+1$ adders,
- $2KMN+3KM+2KN+9K+1$ multipliers,
- and $K+1$ log-circuits.

For the Mackey-Glass problem (where M=4, N=1, and K=8), the number of arithmetic operators in the parallel implementations of *Het* is equivalent to 7 comparators, 257 adders, 249 multipliers, and 9 log-circuits.

For the letter recognition problem (where M=100, N=26, and K=8), the number of arithmetic operators in the parallel implementations of *Het* is equivalent to 32 comparators, 46,233 adders, 44,489 multipliers, and 9 log-circuits. This results also includes a post-processor, consisting of 25 comparators.

7.4.2 Radial Basis Function Architecture

According to the specifications defined in section 7.2, the parallel implementation of the radial basis function *RBF* architecture is composed of

- K RBF modules, each consisting of $5M-1$ adders, $3M+4$ multipliers and 1 exponent-circuit.
- an output module, consisting of $2KN+K-1$ adders and $2KN+2N$ multipliers;
- a membership indicator, consisting of $K-1$ comparators and K multipliers;
- a learning rate unit, consisting of $2K-1$ adders, $K+1$ multipliers, and $K+1$ log-circuits;

- a width unit, consisting of $2K^2M-K^2-2KM+2K-1$ adders, $K^2M-KM+1$ multipliers, and K^2-2K comparators;

These specifications indicates that the total number of operators in the implementation are

- K^2-K-1 comparators,
- $2K^2M-K^2+3KM+2KN+4K-3$ adders,
- $K^2M+2KM+2KN+6K+2N+2$ multipliers,
- $K+1$ log-circuits,
- and K exponential-circuits.

For the Mackey-Glass problem (where M=4, N=1, and K=64), the number of arithmetic operators in the parallel implementations of *RBF* is equivalent to 4,031 comparators, 29,821 adders, 17,412 multipliers, 65 log-circuits, and 64 exponential-circuits.

For the letter recognition problem (where M=100, N=26, and K=128), the number of arithmetic operators in the parallel implementations of *RBF* is equivalent to 16,280 comparators, 3,305,981 adders, 1,671,350 multipliers, 129 log-circuits, and 128 exponential-circuits. These numbers also include a post-processor, consisting of 25 comparators.

7.4.3 Back-Propagation Architecture

For the back-propagation architecture described in subsection 7.3.1, which is used for function approximation, its implementation is composed of

- H_1 first-hidden-layer units, each consisting of H_2+2M+1 adders, $2M+2$ multipliers, and 1 sigmoid-circuit;

- H_2 second-hidden-layer units, each consisting of $2H_1+N+1$ adders, $3H_1+2$ multipliers, and 1 sigmoid-circuit;
- and N linear output units, each consisting of $2H_2+2$ adders and $3H_2+1$ multipliers.

In summary, the total number of operators in this implementation are

- $3H_1H_2+2H_1M+3H_2N+H_1+H_2+2N$ adders,
- $3H_1H_2+2H_1M+3H_2N+2H_1+2H_2+N$ multipliers,
- and H_1+H_2 sigmoid-circuits.

For the parallel implementation of the *BP* architecture used in the Mackey-Glass problem (where M=4, N=1, and H_1=H_2=20), its operator requirements are 1,462 adders, 1,501 multipliers, and 40 sigmoid-circuits.

For a back-propagation architecture described in subsection 7.3.2, which is used for classification, its implementation is composed of

- H hidden units, each consisting of $2M+N+1$ adders, $2M+2$ multipliers, and 1 sigmoid-circuit;
- N sigmoidal output units, each consisting of $2H+3$ adders, $3H+3$ multipliers, and 1 sigmoid-circuit.

The total number of operators in this implementation are

- $N-1$ comparators,
- $2HM+3HN+H+3N$ adders,
- $2HM+3HN+2H+3N$ multipliers,

- and $H+N$ sigmoid-circuits.

For the parallel implementation of the *BP* architecture used in the hand-written letter recognition problem (where M=100, N=26, and H=10), its operator requirements are 25 comparators, 2,868 adders, 2,878 multipliers, and 36 sigmoid-circuits.

7.4.4 Hardware Cost Comparison

Table 7.1 summarizes the number of arithmetic operators required in the parallel implementations of *Het*, *RBF*, and *BP*. To compute the hardware cost from these operator requirements, it is necessary that we know the silicon areas of the various operators in the implementation. In this investigation, similar to the case of serial implementations, we assume that each adder and comparator in the implementation are for two 32-bit numbers, and each multiplier is for two 16-bit number. We also assume that each non-linear function is implemented by a look-up table having 256 entries, each of 32 bits. Table 7.2 lists the implementation costs of the *Het*, *RBF* and *BP* architectures for the Mackey-Glass and letter recognition problem. We express the implementation costs in adder-area units, *aau*s, where 1 *aau* is equivalent to the area of one adder. In this hardware cost estimation, we approximate the silicon area of a comparator to be 0.5 *aau*s, the area of a multiplier to be 7 *aau*s, and the area of a look-up table to be 13 *aau*s. These approximations are based on the specifications of the SPERT chip [49, 50]. Table 7.2 shows the implementation costs of *Het*, *RBF*, and *BP*. It also shows the ratio of the hardware cost of each architecture to that of the *Het* architecture of the same problem. According to this table, the implementation cost of *Het* is lowest for the Mackey-Glass problem, and that of *BP* is lowest for the letter recognition problem. The implementation costs of the *RBF* architeture are highest for both problems.

Table 7.1: Number of Operators in the Parallel Implementations

Mackey-Glass Time Series Prediction				
Algorithm	Comparator	Adder	Multiplier	Nonlinear
Het	7	257	249	9
RBF	4,031	29,821	17,412	129
BP	0	1,462	1,501	40
Character Classification				
Algorithm	Comparator	Adder	Multiplier	Nonlinear
Het	32	46,223	44,489	9
RBF	16,280	3,305,981	1,671,478	257
BP	25	2,868	2,878	36

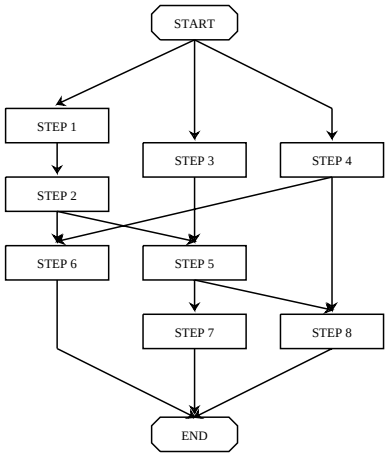
Table 7.2: Hardware Cost of the Parallel Implementations in adder area units.

Application	Algorithm	Area Cost	Ratio
Mackey-Glass	*Het*	2,070	1.00
	RBF	155,397	75.07
	BP	12489	6.03
Character	*Het*	357,789	1.00
	RBF	15,017,808	41.97
	BP	23,494	0.07

7.5 Computation Time for Parallel Implementations

In this section, we determine *computation times* of the *Het*, *RBF* and *BP* architectures for the parallel hardware specified in sections 7.1, 7.2, and 7.3. For this section, we compute for each architecture a *training-cycle time*, the time needed to to perform the arithmetic operations in one training cycle. We then compare the convergence rates of the three architectures with respected to the computational time, defined as the product of the training-cycle time and the number of pattern presentations. Even though our definition of a one-training-cycle does not include the time needed for transmitting data from one operator to another operator, it is accurate enough for establishing the *relative* speeds of the learning algorithms in the three aforementioned architectures.

7.5.1 Heterogeneous Architecture



Step 1: for $k = 1$ to K do in parallel
compute the error-weighted deviation between $\vec{x}$ and $\vec{c}_k$.
Step 2: determine the membership indicators.
Step 3: compute the outputs of all expert modules $\vec{f}_k$.
Step 4: determine the learning rates η_{km} and η_{lms}
Step 5: compute the output $\vec{f}(\vec{x}) = M_k(\vec{x})\vec{f}_k(\vec{x})$.
Step 6: update $\vec{c}_k$ and $\hat{\nu}_k$ for all $1 \leq k \leq K$.
Step 7: update $\hat{\epsilon}_k$ for $1 \leq k \leq K$.
Step 8: update $\vec{w}_k$ for $1 \leq k \leq K$.

Figure 7.30: Flow chart of the learning algorithm of the heterogeneous architecture.

Figure 7.30 shows the learning algorithm of the *Het* architecture for the hardware implementation specified in section 7.1.

Analysis

Step 1: This step is performed by the sub-unit for computing the error-weighted deviation and is completed in $1 + \lceil \log_2 M \rceil$ addition times plus 2 multiplication times.

Step 2: This step is performed by the membership indicator and is completed in $\lceil \log_2 K \rceil$ comparison times.

Step 3: This step is performed by the sub-units for computing $f_{k,i}$ in all the output units, and is completed in $\lceil \log_2 (M+1) \rceil$ addition times plus 1 multiplication time.

Step 4: This step is performed by the learning rate unit and is completed in $2 + \lceil \log_2 K \rceil$ addition times plus 2 multiplication times plus 1 log-time.

Step 5: This step is to transmit the output of the expert module corresponding to $M_k(\vec{x})$. Since it takes comparatively short time to close or open the switchs of the multiplexer, we assume that there is no delay in this step.

Step 6: This step is performed by the sub-units for updating $\vec{c}_k$ and $\hat{\nu}_k$, and is completed in $2 + \lceil \log_2 M \rceil$ addition time plus 3 multiplication times.

Step 7: This step is performed by the sub-units for updating $\hat{\epsilon}_k$ and is completed in $2 + \lceil \log_2 N \rceil$ addition times plus 2 multiplication times.

Step 8: This step is performed by the sub-units for updating $\vec{w}_{k,i}$ and is completed in 2 addition times plus 2 multiplication times.

End of Analysis

For the Mackey-Glass problem (where M=4, N=1, and K=8), this analysis indicates that the

critical path of the learning cycle of the *Het* architecture consists of step 4 and 6. The computation time of this critical path is equivalent to 9 addition times plus 4 multiplication times plus 1 log-time.

For the letter recognition problem (where M=100, N=26, and K=8), the above analysis indicates that the critical path consists of steps 1, 2 and 6. The computation time of such a path is equivalent to 3 comparison times plus 17 addition times plus 4 multiplication times. However, since the 26 outputs of the *Het* architecture are used as the estimates of the likelihoods of an input being various capital letters, we need to compare these 26 outputs in order to determine the class of the input. This determination needs $\lceil \log_2 26 \rceil = 5$ comparison times. Therefore, the total execution time of one training cycle is equivalent to 8 comparison times plus 17 addition times plus 4 multiplication times.

7.5.2 Radial Basis Function Architecture

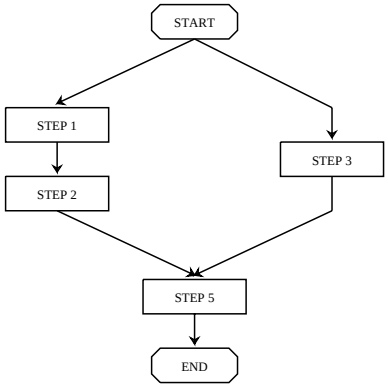
For the *RBF* architecture investigated in this dissertation, its learning procedure is divided into 3 stages:

- Locating the centers of the Gaussian functions using the k-means algorithm,
- Determining the widths of the Gaussian functions,
- Adjusting the heights of the Gaussian functions.

For the purpose of determining the computation time, we describe the algorithms in these three stages based on the parallel implementation specified in section 7.2. We then analyze each step in the algorithm for its computation time.

7.5.2.1 Determining the Centers of the Gaussian Functions

The k-means algorithm for determining the centers of the Gaussian functions are described by the flowchart in figure 7.31.



Step 1: for $k = 1$ to K do in parallel
compute the Euclidean deviation
between $\vec{x}$ and $\vec{c}_k$.
Step 2: determine the membership indicator M_k.
Step 3: determine learning rates $\eta_{k\,m}$.
Step 4: update $\vec{c}_k$ and $\hat{v}_k$ for $1 \leq k \leq K$.

Figure 7.31: Flowchart of the k-means algorithm for computing the centers of the Gaussian functions.

Analysis

Step 1: This step is performed by the sub-unit in the RBF modules, and is completed in $1+\lceil \log_2 M \rceil$ addition times plus 1 multiplication time.

Step 2: This step is performed by the membership indicator, and is completed in $\lceil \log_2 K \rceil$ comparison times plus 1 multiplication time.

Step 3: This step is performed by the learning rate unit and is completed in $\lceil \log_2 K \rceil$ addition times plus 2 multiplication times plus 1 log-time.

Step 4: This step is performed by the sub-unit for updating $\vec{c}_k$ and $\hat{v}_k$, and is completed in $2 + \lceil \log_2 M \rceil$ addition times plus 2 multiplication times.

End of Analysis

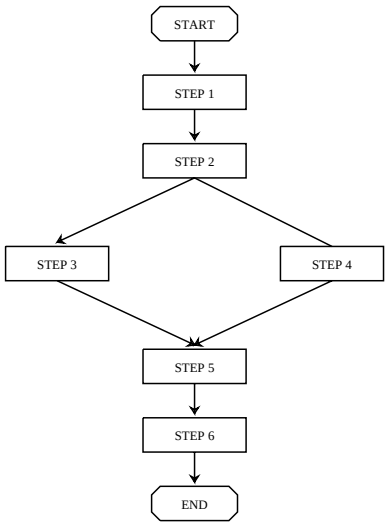
This analysis indicates that it takes $3+2\lceil\log_2 M\rceil$ addition times plus $\lceil\log_2 K\rceil$ comparison times plus 4 multiplication times, or $2+\lceil\log_2 K\rceil+\lceil\log_2 M\rceil$ addition times plus 4 multiplication times plus 1 log-time, whichever is longer, to complete one training cycle of the k-means algorithm.

7.5.2.2 Determining the Width of the Gaussian Functions

We use the width unit, described in section 7.2.5, to determine the width of the Gaussian functions. The computation time of this unit is equivalent to $\lceil\log_2(K-1)\rceil$ comparison times plus $1+\lceil\log_2 M\rceil+\lceil\log_2 K\rceil$ addition times plus 2 multiplication times.

7.5.2.3 Determining the Heights of the Gaussian Functions

The *least mean square (LMS)* algorithm is used to determine the heights of the Gaussian functions. This algorithm is described in the figure 7.32



Step 1: for $k = 1$ to K do in parallel
compute $r^2 = \|\vec{x} - \vec{c}_k\|^2$.
Step 2: for $k = 1$ to K do in parallel
compute $\phi_k = \exp(r^2/\sigma^2)$.
Step 3: compute $\sum_{k=1}^{K} \phi_k$.
Step 4: for $n = 1$ to N do in parallel
compute output o_n.
Step 5: normalize all outputs o_n by $\sum_{k=1}^{K} \phi_k$.
Step 6: update parameters $\vec{w}_n$.

Figure 7.32: Flowchart of the *LMS* algorithm for determining the heights of the Gaussian functions.

Analysis

Step 1: This step is performed by the sub-unit for computing the Euclidean deviation, and is completed in $1 + \lceil \log_2 M \rceil$ addition times plus 1 multiplication time.

Step 2: This step is performed by the sub-unit for computing a Gaussian function, and is completed in 1 multiplication time plus 1 exponent-time.

Step 3: This step is computed by the summing unit, and is completed in $\lceil \log_2 K \rceil$ addition times

Step 4: This step is performed by the sub-unit for generating output o_n, and is completed in $\lceil \log_2 K \rceil$ addition times and 1 multiplication time.

Step 5: This step is performed by the normalizing unit and is completed in 1 multiplication time.

Step 6: This step is performed by the sub-unit for updating $\vec{w}_n$, and is completed in 2 addition times and 2 multiplication times.

End of Analysis

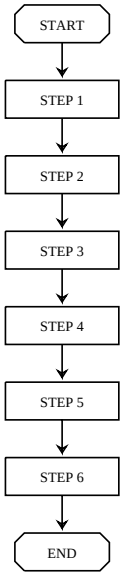
This analysis indicates that it takes $3 + \lceil \log_2 M \rceil + \lceil \log_2 K \rceil$ addition times plus 6 multiplication times plus 1 exponent-time to complete one training cycle of the *LMS* algorithm.

For the Mackey-Glass problem (where M=4, N=1, and K=64), it takes 6 comparison times plus 7 addition times plus 4 multiplication times to complete one cycle of the k-means algorithm. It takes 6 comparison times plus 9 addition times plus 2 multiplication time to determine the widths of the Gaussian functions, and it takes 11 addition times, 6 multiplication times and 1 exponent time to complete one cycle of *LMS*.

For the character recognition (where M=100, N=26, and K=128), it takes 7 comparison times, 17 addition times and 4 multiplication times to complete one training cycle of the k-means algorithm. It takes 7 comparison times plus 15 addition times plus 2 multiplication time to determine the widths of the Gaussian functions. For one training cycle of *LMS*, it takes 5 comparison times plus 17 addition times plus 6 multiplication times plus 1 exponent-time, where the post-processing time is already included.

7.5.3 Back-Propagation Architecture

7.5.3.1 Function Approximation



Step 1: for $i = 1$ to H_1 do in parallel
compute $z_{1,i}$.
Step 2: for $i = 1$ to H_2 do in parallel
compute $z_{2,i}$.
Step 3: for $i = 1$ to N do in parallel
compute f_i.
Step 4: for $i = 1$ to N do in parallel
update parameter $\vec{w}_{3,i}$.
Step 5: for $i = 1$ to H_2 do in parallel
update parameter $\vec{w}_{2,i}$.
Step 6: for $i = 1$ to H_1 do in parallel
update parameter $\vec{w}_{1,i}$.

Figure 7.33: Flow chart of the learning algorithm of the back-propagation architecture for function approximation.

Figure 7.33 shows the learning algorithm of the back-propagation architecture for function approximation. The algorithm is for a network with two hidden layers and a linear output layer which are specified in section 7.3.1.

Analysis

Step 1: This step is performed by the sub-units for generating output $z_{1,i}$ in the first-layer hidden units, and is completed in $\lceil \log_2(M+1) \rceil$ addition times plus 1 multiplication time plus 1 sigmoid-time.

Step 2: This step is performed by the sub-units for generating output $z_{2,i}$ in the second-layer hidden units, and is completed in $\lceil \log_2(H_1+1) \rceil$ addition times plus 1 multiplication time plus 1 sigmoid-time.

Step 3: This step is performed by the sub-units for generating linear output f_i in the linear output units, and is completed in $\lceil \log_2(H_2+1) \rceil$ addition times plus 1 multiplication time.

Step 4: This step is performed by the sub-units for updating parameter $\vec{w}_{3,i}$ in the linear output units, and is completed in 2 addition times plus 3 multiplication times.

Step 5: This step is performed by the sub-units for updating parameter $\vec{w}_{3,i}$ in the linear output units, and is completed in 2 addition times plus 4 multiplication times, or in $1+\lceil \log_2 N \rceil$ addition times plus 3 multiplication times, whichever is maximum.

Step 6: This step is performed by the sub-units for updating parameter $\vec{w}_{1,i}$ in the first-layer hidden units, and is completed in 2 addition times plus 3 multiplication times, or in $1+\lceil \log_2 H_2 \rceil$ addition times plus 2 multiplication times, whichever is maximum.

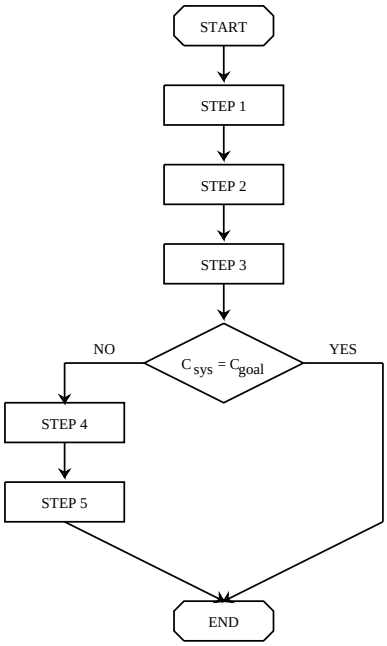
End of Analysis

For the Mackey-Glass problem (where M=4, H_1=H_2=20, and N=1), the time for completing one training cycle of the back-propagation architecture is equivalent to 23 addition times plus 12

multiplication times plus 2 sigmoid-times.

7.5.3.2 Classification

Figure 7.34 shows the learning algorithm of the back-propagation architecture for classification. The learning algorithm is for a network with one hidden layer and sigmoid outputs, described in subsection 7.3.2.



Let C_{sys} denote the category of the input pattern indicated by the back-propagation system.
Let C_{goal} denote the category of the input pattern indicated by the goal function.

Step 1: for $i = 1$ to H do in parallel
compute z_i.
Step 2: for $i = 1$ to N do in parallel
compute f_i.
Step 3: classify the category of input $\vec{x}$
such that $f_{C_{sys}}(\vec{x}) \geq f_i(\vec{x})$ for $1 \leq i \leq N$.
Step 4: for $i = 1$ to N do in parallel
update parameter $\vec{w}_{2,i}$.
Step 5: for $i = 1$ to H do in parallel
update parameter $\vec{w}_{1,i}$.

Figure 7.34: Flow chart of the learning algorithm of the back-propagation architecture for classification.

Analysis

Step 1: This step is performed by the sub-units for generating output z_i in the hidden units, and is completed in $\lceil \log_2(M+1) \rceil$ addition times plus 1 multiplication time plus 1 sigmoid-time.

Step 2: This step is performed by the sub-units for generating output f_i in the sigmoidal output units, and is completed in $\lceil \log_2(H+1) \rceil$ addition times plus 1 multiplication time plus 1

sigmoid-time.

Step 3: This step is completed in $\lceil \log_2 N \rceil$ comparison times.

Step 4: This step is performed by the sub-units for updating parameter $\vec{w}_{2,i}$ in the sigmoidal output units, and is completed in 2 addition times plus 4 multiplication times.

Step 5: This step is performed by the sub-units for updating parameter $\vec{w}_{1,i}$ in the hidden units, and is completed in 2 addition times plus 3 multiplication times, or in $1+\lceil \log_2 H_2 \rceil$ addition times plus 3 multiplication times, whichever is maximum.

End of Analysis

For the character recognition problem (where M=100, H=10, and N=26), the time needed to complete one training cycle of the back-propagation architecture is 5 comparison times plus 18 addition times plus 8 multiplication times plus 2 sigmoidal times.

7.5.4 Computation Time Comparison

Table 7.3 summarizes the training-cycle times of *Het*, *RBF* and *BP* architectures for the Mackey-Glass and letter recognition problems, derived in sections 7.5.1, 7.5.2, and 7.5.3. Table 7.4 lists the training-cycle times in terms of *addition time units (atu).* It also lists the ratio of the training-cycle time of each algorithm to that of *Het* for the same problem. The figures in this table are derived by translating the results listed in table 7.3 using the same assumptions as those in section 6.4. That is, we approximate the delay time of comparison to be 1 *atu*, the delay of multiplication to be 2 *atu*s, and the delay time for retrieval of a non-linear function value from a look-up table are 2 *atu*s. Comparing these training-cycle times reveals that the training-cycle time of *Het* is shorter than the other two.

Table 7.3: Times for Completing One Training Cycle Measured in Types of Operations.

Mackey-Glass Time Series Prediction				
Algorithm	Comparison	Addition	Multiplication	Nonlinear
Het	0	9	4	1
RBF(Center)	6	7	4	0
RBF(Width)	6	9	2	0
RBF(Height)	0	11	6	1
BP	0	23	12	2
Character Classification				
Algorithm	Comparison	Addition	Multiplication	Nonlinear
Het	8	17	4	0
RBF(Center)	7	17	4	0
RBF(Width)	7	15	2	0
RBF(Height)	5	17	6	1
BP	5	18	8	2

Table 7.4: Times for Completing One Training Cycle Measured in Addition Time Units.

Application	Algorithm	Cycle Time	Ratio
Mackey-Glass	*Het*	19	1.00
	RBF(Center)	21	1.10
	RBF(Width)	19	1.00
	RBF(Height)	25	1.32
	BP	51	2.68
Character	*Het*	28	1.00
	RBF(Center)	32	1.14
	RBF(Width)	24	0.86
	RBF(Height)	36	1.28
	BP	43	1.54

Figures 7.35a and 7.35b show the *NMSE*s of the *Het*, *RBF*, and *BP* architectures with respect to the computation time for the Mackey-Glass and letter recognition problems. We measure the computation time in terms of the training-cycle time of the *Het* architecture. The computation times of *Het* and *BP* are defined to be the product of the number of pattern presentations and the training-cycle times of the corresponding learning algorithms. For *RBF*, its computation time is defined as:

$$\text{computation time} = N_p(T_{KM} + T_{Ht}) + T_{Wd}, \tag{7.10}$$

where N_p stands for the number of pattern presentations, and T_{KM}, T_{Ht}, and T_{Wd} stand for the training-cycle time of the k-means algorithm, the time for determining the width of the Gaussian function, and the training-cycle time of the *LMS* algorithm, respectively. These two figures reveal that *Het* has much better convergence rate than *RBF* and *BP*, when measured in terms of the computation time.

7.6 Complexity Comparison

In this section, we compare the complexities of the described parallel implementations of the *Het*, *RBF* and *BP* architectures for the Mackey-Glass problem and the hand-written letter recognition task. For each of these implementations, we define its complexity to be the product of the area cost and the computation time. Figures 7.36a and 7.36b show the *NMSE*s of the *Het*, *RBF*, and *BP* architectures with respect to the time-area complexity for the Mackey-Glass and letter recognition problems. These results reveal that the *Het* architecture requires less complexity than those required by the *RBF* and *BP* architectures for attaining the equivalent performance.

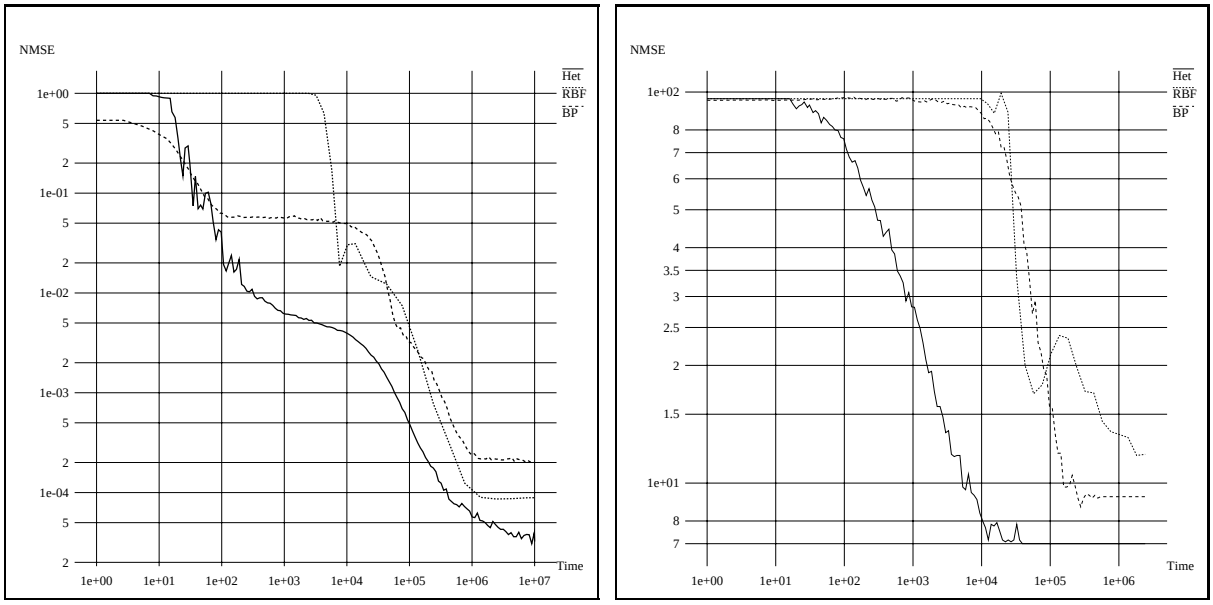


Figure 7.35: Execution time comparison of *Het*, *RBF*, and *BP* on (a) the Mackey-Glass problem, (b) and the hand-written capital letter recognition.

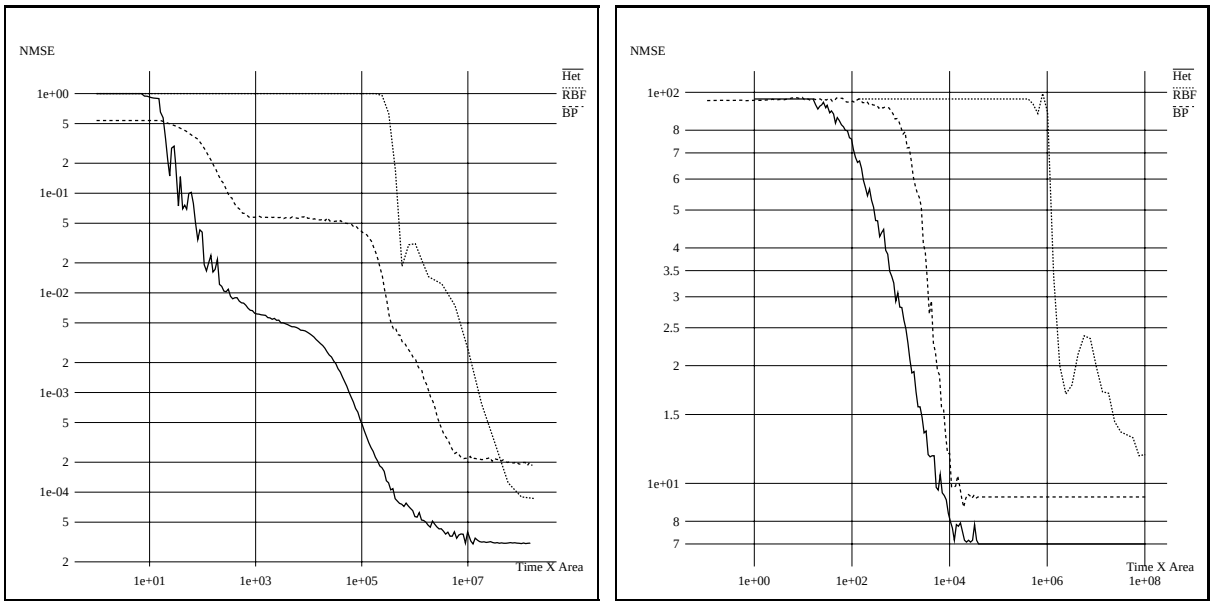


Figure 7.36: Time-Area complexity comparison for the *Het*, *RBF* and *BP* architectures on (a) the Mackey-Glass, (b) and the hand-written capital letter recognition problems.

7.7 Implication for Practical Hardware

As evident in sections 7.1, 7.2, and 7.3, it is unrealistic to implement artificial neural networks using the maximal parallel scheme in digital hardware since the hardware requirements are excessive. Because a purely serial implementation of large scale networks is often too slow, it is necessary to implement these architectures with a serial-parallel approach. In a serial-parallel implementation, a set of processors is programmed to perform the computations in parallel and information among different processors are transferred through a communication network.

Another possibility for implementing artificial neural networks is to use a mixed analog-digital technique. Even though the accuracy of an analog circuit is lower than its digital counterpart, the area of the analog circuit is in general more compact than that of the digital circuit, particularly, when the computational operation is one that is naturally performed by physical processes, such as the Kirchoff's current law to add, or the Ohm's law to multiply. In the heterogeneous architecture, the k-means algorithm performed in the gating module does not require high accuracy. It can thus be implemented with analog technology so that the implementation can be more compact. In fact, several analog or mixed analog-digital systems for performing the k-means algorithm have been implemented successfully [51, 52].

Chapter 8

Conclusion and Future Research

In this dissertation, we have investigated a class of heterogeneous architectures that are based on k-means partitioning. In these architectures, the *k-means algorithm* is used to partition the input domain into several non-overlapping sub-domains. The task defined on each sub-domain is then solved by an *expert module* trained in a supervised manner. The output of the architecture is defined to be the output of the expert module whose corresponding subdomain contains the input point.

Experiments have shown that the performance of these heterogeneous architectures depends strongly on the efficacy of the k-means algorithm in partitioning the input domain. In order to improve the overall performance of the heterogeneous architectures based on k-means partitioning, we have first enhanced the performance of the traditional k-means algorithm by integrating into its process two mechanisms: the first for biasing the partitioning process so that the algorithm can achieve an optimal partition, and the second for adjusting the learning rate dynamically, permitting the algorithm to converge very rapidly at first and later very closely towards an optimal solution. We

have also modified the deviation measure of this enhanced k-means algorithm so that the algorithm partitions the input domain based on both the input distribution, and information about the goal function and the capability of the expert modules. This new deviation measure allows the k-means algorithm to adjust the size of each individual sub-domain in the partition so that the representation resources in all the sub-domains are optimally used.

We have compared the heterogeneous architectures based on the error-weighted k-means partitioning against two traditional homogeneous architectures: the multi-layered perceptron trained by back-propagation and the radial basis function architecture. The performance of these three architectures are compared on the Mackey-Glass time series prediction and on a hand-written capital letter recognition task. We have found that the heterogeneous architecture exhibits better generalization than the radial basis function and back-propagation architectures. The convergence rate of the heterogeneous architecture, measured in terms of the number of training pattern presentations, is also faster than those of the back-propagation and radial basis function architectures. We have also evaluated the convergence rate with respect to the computational time for two specific cases: a *serial* implementation, where the operations in the architecture are performed in serial, and a *parallel* implementation, where the operations in the algorithm are performed with maximal concurrency. For both cases, the convergence rate of the heterogeneous architecture, measured in terms of the computation time, is faster than those of the back-propagation and radial basis function architectures. We have also found that in the parallel implementation, the convergence rate of the heterogeneous architecture when scaled with its hardware complexity is also lower than the other two architectures.

The heterogeneous architectures introduced in this dissertation can be further improved

in several aspects. In this study we assume that K, the number of the expert modules in the heterogeneous architecture, is pre-determined, i.e., by the available hardware. It appears worthwhile to study the trade-offs in varying K according to the characteristics of a given problem. Ideally one could come up with a cost heuristics that automatically adds or deletes expert modules to achieve an optimal performance/cost ratio [53]. Assuming that such a system is implemented on a machine with a finite amount of hardware, where there is a significant amount of multiplexing of "virtual" expert modules through the same physical processors, changing the number of expert modules would simply change the degree of multiplexing, but would not require any structural changes in the hardware architecture.

Currently, our heterogeneous architectures use binary-valued membership indicators. Since there is no overlap among sub-domains, the goal functions defined on the various sub-domains are approximated independently by different expert modules, resulting in a discontinuous output function. In order for a heterogeneous architecture to generate a continuous output function, smooth membership indicators which partition the input domain into overlapping sub-domains are needed [25]. To use smooth membership indicators, we would define the output of the architecture to be a combination of the expert outputs weighted by their corresponding smooth membership indicators. This would enable us to vary the output of the heterogeneous architecture smoothly across the boundaries of sub-domains.

The superior performance of the heterogeneous architecture is attributed to the fact that the assigned task is divided into sub-tasks, each solved by a different expert module. The architecture thus has some flexibility in defining the input domain of each expert module, allowing the granularity of the sub-task to match the characteristics of the basic function in the expert module. One

significant demonstration of this dissertation is that the heterogeneous architecture is more suitable for addressing large, complex problems than are traditional homogeneous architectures. This study illustrates that a large general purpose artificial neural network should be composed of a variety of different modules.

Bibliography

[1] A. R. Barron and R. L. Barron. Statistical learning networks: a unifying view. In *Symposium on the Interface: Statistics and Computing Science*, Reston, VA, April 1988.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propoagation. In *Parallel distributed processing: Explorations in the microstructure of cognition*, volume 1. Bradford Books, Cambridge, MA, 1986.

[3] D. Broomhead and D. Lowe. Multivariate functional interpolation and adaptive networks. *Complex System*, 2:321–355, 1988.

[4] S. Lee and R. Kill. Multilayer feedforward potential function networks. In *Proceedings of the 2nd IEEE International Conference on Neural Networks (ICNN-88)*, volume I, July 1988.

[5] M. Niranjan and F. Fallside. Neural networks and radial basis functions in classifying static speech patterns. Technical Report CUEDIF-INFENG17R22, Engineering Department, Cambridge University, 1988.

[6] J. Moody and C. J. Darken. Learning with localized receptive fields. In Touretzky, Hinton, and Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School*, San Mateo, CA, 1988.

[7] T. Poggio and F. Girosi. A theory of networks for approximation and learning. Technical Report A.I.Memo No. 1140, Massachusetts Institute of Technology, 1989.

[8] Y. S. Abu-Mostafa. Information theory, complexity, and neural networks. *IEEE Communications Magazine*, November 1989.

[9] G. Tesauro. Scaling relationships in back-propagation learning. *Complex System*, April 1987.

[10] S. Geman, E. Bienenstock, and R. Doursat. Neural networks and the bias/varaince dilemma. *Neural Computation*, 4:1–58, 1992.

[11] L. A. Akers and M. R. Walker. A limited-interconnect synthetic neural ic. In *Proceedings of the IEEE International Conference on Neural Networks (ICNN-88)*, July 1988.

[12] D. H. Hubel and T. N. Weisel. Receptive fields, binocular interaction and functional architecture in cat's visual cortex. *J. Physiol. (London)*, 160:106–154, January 1962.

[13] D. H. Hubel and T. N. Weisel. Receptive fields and functional architecture in two nonstriate visual area (18 and 19) of the cat. *J. Neurophysiol. (London)*, 28:229–289, 1965.

[14] D. H. Hubel and T. N. Weisel. Brain mechanism of vision. *Scientific American*, pages 130–146, September 1979.

[15] H. Ritter, T. Martinez, and K. Schulten. Topology conserving maps for learning visuo motor coordination. *Neural Networks*, 2(3):159–168, 1989.

[16] A. H. Waibel. Consonant recognition by modular construction of large phonemic time-delay neural networks. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, 1989.

[17] R. D. Jones, Y. C. Lee, C. W. Barnes, G. W. Flake, K. Lee, P. S. Lewis, and S. Qian. Function approximation and time series prediction with neural networks. Technical Report LA-UR 90-21, Los Alamos National Laboratory, Los Alamos, NM, 1990.

[18] J. B. Hamshire and A. H. Waibel. The meta-pi network, building distributed knowledge representations for robust pattern recognition. Technical Report CMU-CS-89-166, Carnegie-Mellon University, Pittsburg, PA, 1989.

[19] Y. Mori and K. Joe. A large-scale neural network which recognizes handwritten kanji cahracters. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*. Morgan Kaufmann, 1990.

[20] R. A. Jacobs. *Task decomposition through competition in a modular connectionist architecture*. PhD thesis, Department of Computer & Information Science, University of Massachusetts, Amherst, MA, September 1990.

[21] D. Fox, V. Heinze, K. Moller, S. Thrun, and G. Veenker. Learning by error-driven decomposition. In *Proceedings of the International Conference on Artitificial Neural Networks (ICANN-91)*. Elsevier Science Publishers, June 1991.

[22] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3(1), 1991.

[23] H. Kita, H. Masataki, and Y. Nishikawa. Nn/ii: Improved version of network for large-scale pattern recognition tasks. In *Proceedings of the International Joint Conference on Neural Networks, Singapore (IJCNN-91)*, November 1991.

[24] S. M. Omohundro. Geometric learning algorithm. Technical Report TR-89-041, International Computer Science Institute (ICSI), Berkeley, CA, July 1989.

[25] S. M. Omohundro. Bumptrees for efficient function, constraint, and classification learning. In *1990 IEEE Conference on Neural Information Processing Systems - Natural and Synthetic*, November 1990.

[26] S. P. Lloyd. Least squares quantization in pcm. *IEEE Transaction on Information Theory*, IT-28(2):129–137, 1982.

[27] J. MacQueen. Some methods for classification and analysis of multivariate observations. *Proc. 5th Berkeley Symp. Math. Stat. Prob.*, 281, 1967.

[28] T. Kohonen. Clustering, taxonomy, and topological maps of patterns. In M. Lang, editor, *Proceedings of the Sixth International Conference on Pattern Recognition*, Silver Spring, MD, 1982. IEEE Computer Society Press.

[29] J. Moody and C. J. Darken. Fast learning in network of locally tuned processing units. *Neural Computation*, 1:281–294, 1989.

[30] W. M. Huang and R. P. Lippmann. Neural net and traditional classifiers. In D. Anderson, editor, *Neural Information Processing System*, 1988.

[31] B. Widrow and M. E. Hoff. Adaptive switching circuit. In *Wescon Convention Record*, 1960.

[32] J. Tou and R. C. Gonzalez. *Pattern Recognition Principles*. Addison-Wesley Publishing Company, Inc, 1974.

[33] D. E. Rumelhart and D. Zipser. Feature discovery by competitive learning. In *Parallel distributed processing: Explorations in the microstructure of cognition*, volume 1. Bradford Books, Cambridge, MA, 1986.

[34] D. Desieno. Adding a conscience to competitive learning. In *Proceedings of the 2nd IEEE International Conference on Neural Networks (ICNN-88)*, volume I, July 1988.

[35] C. Darken and J. Moody. Fast adaptive k-means clustering: some empirical results. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN-90)*, June 1990.

[36] C. Darken and J. Moody. Learning schedules for stochastics optimization. In *1990 IEEE Conference on Neural Information Processing Systems - Natural and Synthetic*, November 1990.

[37] A. Gersho. Asymptotically optimal block quantization. *IEEE Transaction on Information Theory*, IT-25(4):373–380, 1979.

[38] P. Boucher and M. Goldberg. Color image compression by adaptive vector quantization. *Proc. IEEE ICASSP*, pages 29.6.1–29.9.4, March 1984.

[39] Y. Linde, A. Buzo, and R. M. Gray. A algorithm for vector quantizer design. *IEEE Transaction on Communications*, COM-28(4):84–95, January 1980.

[40] R. Hecht-Nielsen. Counterpropagation networks. *Applied Optics*, 26(3):4979–4984, December 1987.

[41] A. Saha and J. D. Keeler. Algorithm for better representation and faster learning in radial basis function networks. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*. Morgan Kaufmann, 1990.

[42] B. Kosko. Stochastic competitive learning. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN-90)*, June 1990.

[43] C. Chinrungrueng and C. H. Séquin. Optimal adaptive k-means algorithm with dynamic adjustment of learning rate. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN-91-Seattle)*, July 1991.

[44] M.C. Mackey and L. Glass. Oscillation and chaos in physiological control systems. *Science*, 197:287, 1977.

[45] A. S. Lapedes and R. Farber. How neural nets work. In D. Z. Anderson, editor, *Neural information processing systems*. American Institute of Physics, 1988.

[46] A. H. Kramer and A. Sangiovani-Vincentelli. Efficient parallel learning algorithms for neural networks. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, 1989.

[47] R. S. Crowder. Predicting the mackey-glass timeseries with cascade-corelation learning. In Touretzky, Hinton, and Sejnowski, editors, *Proceedings of the 1990 Connectionist Models Summer School*, San Mateo, CA, 1990.

[48] Yan Le Cun. Hlm: a multilayer learning network. In Touretzky, Hinton, and Sejnowski, editors, *Proceedings of the 1986 Connectionist Models Summer School*, San Mateo, CA, 1986.

[49] K. Asanović, B. Kingsbury, and N. Morgan. Spert specifications. Personal communication, December 1992.

[50] K. Asanović, J. Beck, B. Kingsbury, P. Kohn, N. Morgan, and J. Wawrzynek. Spert: A vliw/simd microprocessor for artificial neural network computations. Technical Report TR-91-072, International Computer Science Institute (ICSI), Berkeley, CA, December 1991.

[51] J. R. Mann and S. Gilbert. An analog self-organizing neural network chip. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, 1989.

[52] W. Fang, B. Sheu, and S. Chen. A real-time vlsi neuroprocessor for adaptive image compression based upon frequency-sensitive competitive learning. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN-91-Seattle)*, July 1991.

[53] J. Buhmann and H. Kuhnel. Complexity optimized vector quantization: a neural network approach. Personal communication, 1992.

Appendix A

Asymptotic Property of Partition That Minimizes the Total Variation-Weighted Variation

This appendix proves that when K, the number of regions in the partition, is large and p, the distribution of the pattern vectors $\vec{x}$, is smooth, the optimum partition and the optimum set of reference vectors that minimize the total variation-weighted variation *(TVV)* and those that minimize the total spatial variation *(TSV)* are the same. The definitions of the *TVV* and *TSV* are

$$TVV = \sum_{k=1}^{K} v_k \int_{\mathcal{I}_k} p(\vec{x}) \, \|\vec{x} - \vec{c}_k\|^2 \, d\vec{x} \quad \stackrel{\text{def}}{=} \quad \sum_{k=1}^{K} v_k^2, \tag{A.1}$$

$$TSV = \sum_{k=1}^{K} \int_{\mathcal{I}_k} p(\vec{x}) \, \|\vec{x} - \vec{c}_k\|^2 \, d\vec{x} \quad \stackrel{\text{def}}{=} \quad \sum_{k=1}^{K} v_k. \tag{A.2}$$

Without loss of generality, we define $\vec{c}_k$ to be the centroid of the vectors in the region $\mathcal{I}_k$. This definition is based on the optimality condition [39] that for a given partition, the optimum $\vec{\boldsymbol{c}}$ that minimizes all v_k are the Eulcidean centroids of the regions in the partition.

Suppose $\boldsymbol{I}^*$ is the optimum partition that minimizes *TSV*. We are going to show that this $\boldsymbol{I}^*$ also minimizes *TVV*. Let v_1^*, ... , v_K^* be the within-region variation of the optimum partition $\boldsymbol{I}^*$. Hence

$$\sum_{i=1}^{K} v_k \geq \sum_{i=1}^{K} v_k^* \tag{A.3}$$

For arbitrary values of v_1, ..., v_K;

$$\frac{1}{K}\sum_{k=1}^{K} v_k^2 \geq \left(\frac{1}{K}\sum_{k=1}^{K} v_k\right)^2, \tag{A.4}$$

where the equality holds if and only if $v_1 = \cdots = v_K$.

Substituting equation A.3 into equation A.4, we obtain

$$\frac{1}{K}\sum_{k=1}^{K} v_k^2 \geq \left(\frac{1}{K}\sum_{k=1}^{K} v_k^*\right)^2. \tag{A.5}$$

Gersho [37] showed that for asymptotically large K and a smooth underlying probability distribution, the within-region variations v_k^* of the optimum partition $\boldsymbol{I}^*$ must satisfy

$$v_1^* = \cdots = v_K^* = v^*. \tag{A.6}$$

Combining equation A.5 and A.6, we get

$$\frac{1}{K}\sum_{k=1}^{K} v_k^2 \geq (\ v^*)^2, \tag{A.7}$$

which can be rewritten as:

$$\sum_{k=1}^{K} v_k^2 \geq K(\ v^*)^2. \tag{A.8}$$

Since $K(\ v^*)^2$ is *TVV* of $\boldsymbol{I}^*$, equation A.8 indicates that $\boldsymbol{I}^*$ also minimizes *TVV*.

Let $\boldsymbol{I}^{**}$ denote the optimum partition that minimizes *TVV*. To complete the proof, it is neccessary to show that $\boldsymbol{I}^{**}$ must also minimize *TSV*. Suppose that $\boldsymbol{I}^*$ is the optimum partition of *TSV* and assume that it is different from $\boldsymbol{I}^{**}$. Then,

$$TSV(\ \boldsymbol{I}^{**}) > TSV(\ \boldsymbol{I}^*) \tag{A.9}$$

We have added the argument of $\boldsymbol{I}$ to *TSV* in order to indicate that *TSV* is a function depending on $\boldsymbol{I}$. As we have just proved that $\boldsymbol{I}^*$ must minimize *TVV* if it minimizes *TSV*, equation A.9 thus indicates that

$$TVV(\ \boldsymbol{I}^{**}) > TVV(\ \boldsymbol{I}^*). \tag{A.10}$$

This result contradicts our assumption that $\boldsymbol{I}^{**}$ is the optimum partition of *TVV*. This therefore indicates that $\boldsymbol{I}^{**}$ and $\boldsymbol{I}^*$ have to be the same.