Applying Levenberg-Marquardt Algorithm With Block-Diagonal Hessian Approximation To Recurrent Neural Network Training

by

Chi-cheong Szeto

supervised by

Prof. Lai-wan Chan

A dissertation submitted in partial fulfillment
of the requirement for the degree of

Master of Philosophy

in

The Department of Computer Science and Engineering

The Chinese University of Hong Kong

August, 1999

摘要

Levenberg－Marquardt方法是一種非線性最小平方算法，它能用以訓練使用平方和非線性代價函數的神經網絡，它憑代價函數的二階知識去達到快速的收斂速度。可是使用這方法訓練遞歸神經網絡需要大量的運算和儲藏空間，在成批方式計算中，這方法需要每時期O（N^4T）運算（或在順序方式計算中，需要每時間步O（N^6）運算）和O（N^4）儲藏空間，在這裡N代表完全遞歸隱藏單元的數目，而T代表訓練數據的數目。

我們在本論文提出採用分塊對角海賽矩陣的Levenberg－Marquardt方法，及應用它在遞歸神經網絡訓練。使用這方法訓練遞歸神經網絡需要每時期或每時間步較少的運算（在成批方式計算中，運算在O（N^3T）到O（N^4T）之間；在順序方式計算中，運算在O（N^4）到O（N^6）之間。）和較少的儲藏空間（儲藏空間在O（N^2）到O（N^4）之間）。此外屬於不同分塊的權更新能夠獨立計算，這特性使這方法能並行處理。

這算法有不同的變化，例如：屬於不同分塊的權能夠非同步或同步作出更新，我們分別叫這些更新方法做非同步和同步方法，非同步方法每次更新一個分塊的權，而同步方法每次更新全部分塊的權。此外分割海賽矩陣成分塊對角矩陣涉及選擇分塊的數目、各別分塊的大小和權的成組方法。我們研究和詳細分析以上變化對表現的影響。

Abstract

The Levenberg-Marquardt method is a nonlinear least squares algorithm and can be used to train neural networks with cost functions in the form of sum-of-squares nonlinear functions. It utilizes the second-order information of the cost function to achieve fast convergence speed. However, the arithmetic operations and storage required to train recurrent networks are large. Training a recurrent network with the Levenberg-Marquardt algorithm requires $O(N^4T)$ operations per epoch in batch mode calculation (or $O(N^6)$ operations per time step in sequential mode calculation) and $O(N^4)$ storage, where N is the number of fully recurrent hidden units and T is the number of training data.

In this thesis, we proposed applying the Levenberg-Marquardt method with the block-diagonal Hessian matrix to the recurrent neural network training. This method requires less operations per epoch / time step (ranging from $O(N^3T)$ to $O(N^4T)$ in batch mode calculation or ranging from $O(N^4)$ to $O(N^6)$ in sequential mode calculation) and less storage (ranging from $O(N^2)$ to $O(N^4)$). Moreover, weight updates of different blocks can be calculated independently. This property makes the parallel processing possible.

There are different variations of this algorithm. For example, weights of different blocks can be updated asynchronously or synchronously. We call these updating methods asynchronous and synchronous methods respectively. Asynchronous method updates weights of one block at a time while synchronous method updates weights of all blocks at a time. Moreover, partitioning the Hessian matrix into a block-diagonal matrix involves the choices of the number of blocks, sizes of the respective blocks and weight-grouping methods. Their effects on the performance were studied and analyzed in detail.

Acknowledgment

I would like to thank my supervisor, Prof. Chan for her efforts in helping me to finish the work.

Table of Contents

Chapter 1 Introduction

1.1 Time series prediction

Many things around us appear as collections of observations made sequentially in time. Examples are sales figures measured in successive weeks, share prices measured on successive days, company profits measured in successive years and measurements of the performance of a manufacturing process in successive hours. They range from economics to engineering. Marketing analysts need to predict future sales and economists want to predict economic cycles. Process control specialists need to predict furnace temperatures and investors want to predict the stock market. So, we have great interest in predicting the future values of these time ordered values termed *time series* to make corresponding decisions.

The prediction is often made based on the past information. Let $\mathbf{x}(t)$, t = 0, 1, ... be the time series where t is the time. The d past values of $\mathbf{x}$ ($\mathbf{x}(t)$, $\mathbf{x}(t-1)$, ..., $\mathbf{x}(t-d+1)$) form the d-dimensional *time delay space* or *lag space* for prediction. The number of prediction steps is called the *prediction horizon*. They are illustrated in Equation 1.1.

$$\hat{\mathbf{x}}(t+p) = F(\mathbf{x}(t), \mathbf{x}(t-1), ..., \mathbf{x}(t-d+1)) \qquad (1.1)$$

where $\hat{\mathbf{x}}(t+p)$ is the forecasted value

p is the prediction horizon, $p \geq 1$

d is the dimension of time delay space or lag space, $d \geq 1$

F is the forecasting model

In the next section, the basic forecasting models F used in the literature of neural network are reviewed.

1.2 Forecasting models

In this section, three basic types of forecasting models used in the literature of neural network are described. They are the *networks using time delays* [Dorffner96, Sharda90, Tang93, Ulbricht92, Weigend90], *networks using context units* [Elman90, Jordan86, Karunanithi92, Mori93, Wilson95] and *layered fully recurrent networks* [Li90, Pedersen95].

1.2.1 Networks using time delays

1.2.1.1 Model description

We consider a simple neural network using time delays: a feedforward network with one hidden layer shown in Figure 1-1. The inputs of this network are the input signal x(t) and the delayed signals x(t-1), x(t-2), ..., x(t-d+1) where d is the dimension of the time delay space. We use the time delay elements to obtain the delayed signals. The number of hidden units is determined by the complexity of mapping between the inputs and the outputs.

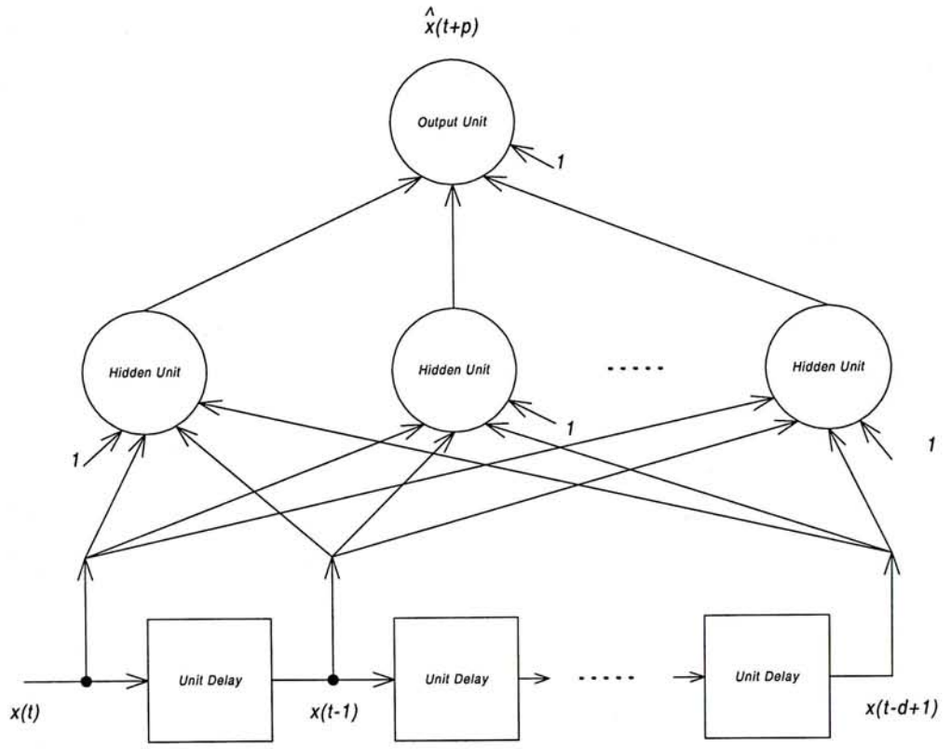


Figure 1-1 Network using time delays

This type of network was used in [Weigend90] to predict the sunspot number and was shown to outperform the threshold autoregressive (TAR) model. In [Sharda90], this model was tested on 75 time series from the M-Competition. This study showed that this model performed as well as an automatic Box-Jenkins modeling expert system, AUTOBOX. In [Tang93], this model was tested on 14 time series. These series included the international airline passenger data, company sale data and time series selected from the M-Competition. This model was shown

to be able to compete with or outperform the Box-Jenkins method when appropriate combinations of network and training parameters were used.

The time delays of the above network are placed between the input signal and the network. They can also be distributed over the whole network to retain information of the previous time steps. For example, time delays can be introduced on the connections between hidden and output units. [Ulbricht92, Dorffner96]

1.2.1.2 Limitation

The limitation of the neural network using time delays is that the number of delayed inputs is fixed and limited. If the number of delayed inputs is small, the amount of past information taken into account is small. However, if the number of delayed inputs is large, learning will be slow due to the increase in the number of weights. Moreover, a lot of training examples are required for successful learning and generalization. In Sections 1.2.2 and 1.2.3, this limitation is solved by feeding the processed past inputs back to the model. In effect, all the past inputs can be taken into account to make the prediction.

1.2.2 Networks using context units

1.2.2.1 Model description

Instead of the explicit use of delayed inputs to keep the past information, context units are used to feed the past information back to the model. For example, *Elman network* [Elman90] shown in Figure 1-2 uses context units to feed the past information from the hidden units back to the model. The hidden states are first copied to the states of the context units. Then, the context units are treated as additional inputs to the network. One important feature of the context units is that the dependence of context states on the weights is ignored during weight updating. This ignorance makes the training of network using context units similar to the training of feedforward network. This reduces the computation complexity significantly. In [Mori93], Elman network was used to do short-term load forecasting in power system. It was superior to the conventional feedforward three-layer neural network in forecasting accuracy.

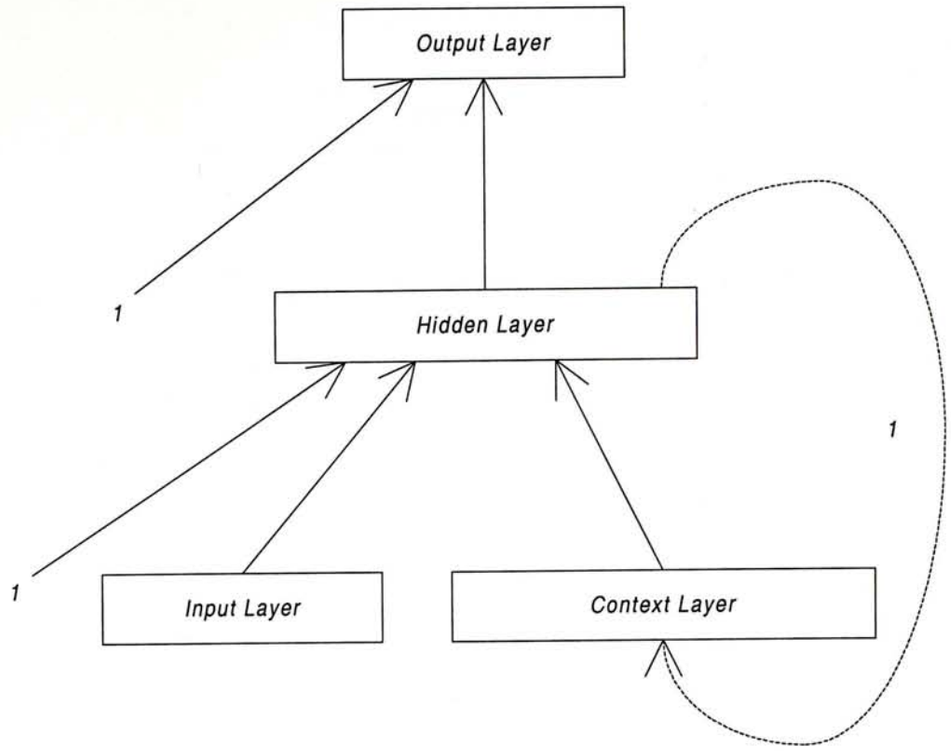


Figure 1-2 Elman network

Context layer can be placed in other locations. For example, Michael I. Jordan proposed a model called *Jordan network* [Jordan86] shown in Figure 1-3, which uses context units to feed the activations of the output units and context units of the previous time step back to the model. In [Karunanithi92], Jordan network was used to predict software reliability. It was tested on 14 different software projects and compared with the feedforward network and five well known software reliability growth prediction models. Jordan network was shown to be the best among the models.

Some networks use several context layers, which contain copies of activations of the previous time steps [Wilson95]. If the activations of the output units are copied to the context layers shown in Figure 1-4, the network is called *Jordan tower network.* If the activations of the hidden units are copied shown in Figure 1-5, the network is called *Elman tower network.* These networks are collectively called *tower-recurrent networks.* The networks with more context layers learned faster and found lower weight configurations with lower total error.

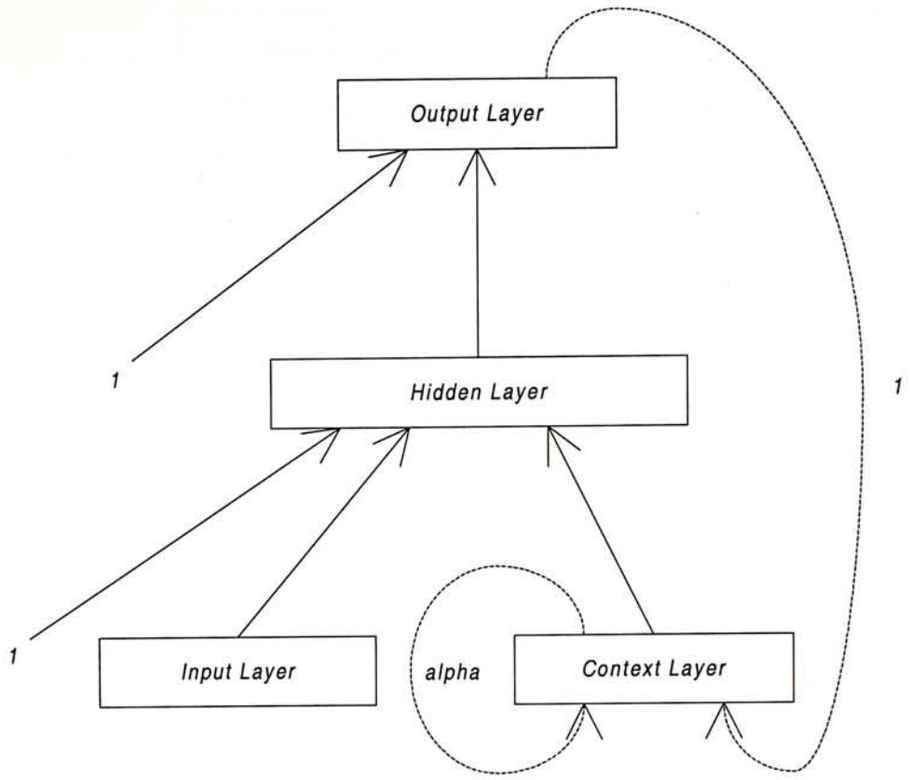


Figure 1-3 Jordan network

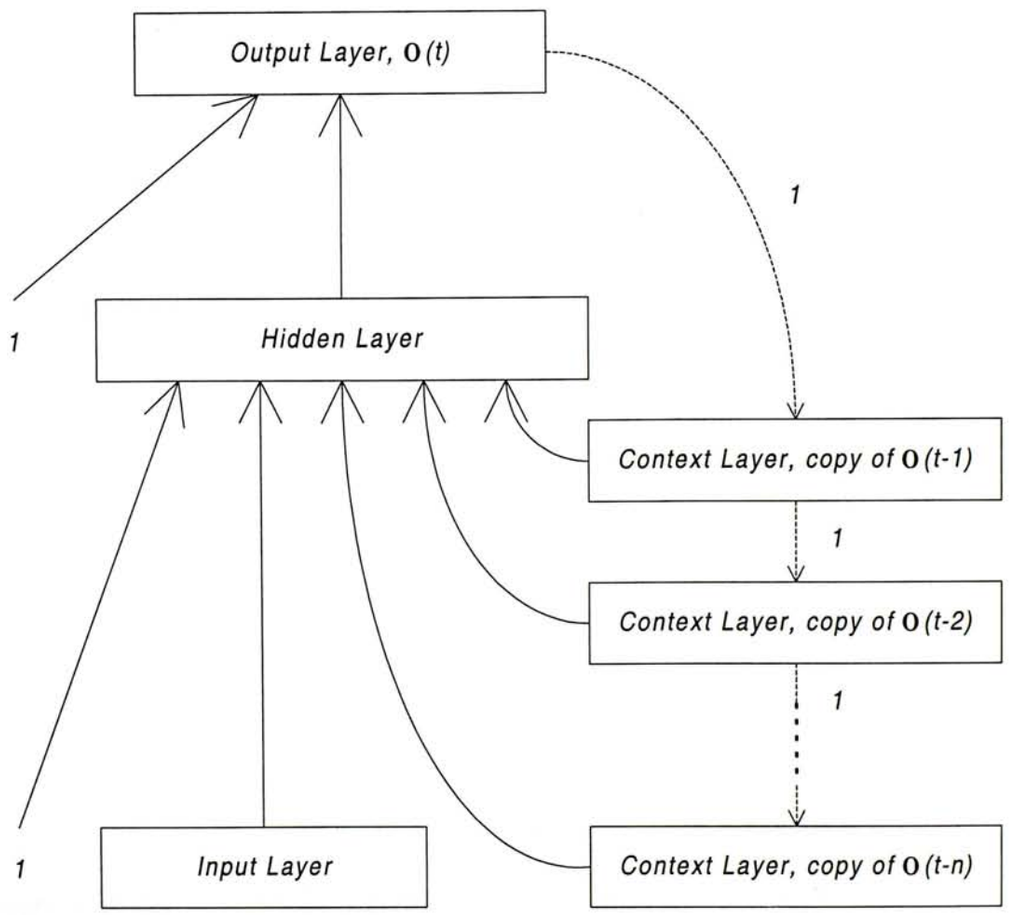


Figure 1-4 Jordan tower network

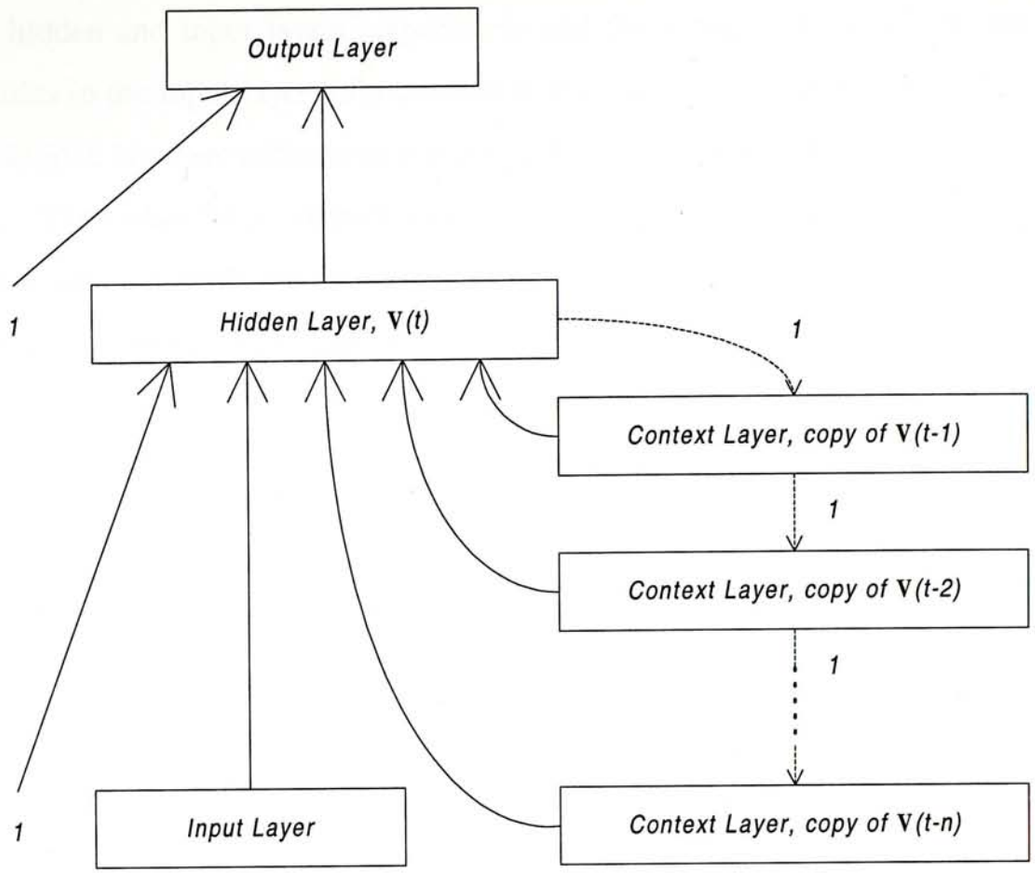


Figure 1-5 Elman tower network

1.2.2.2 Limitation

Since the dependence of context states on the weights is ignored during weight updating, errors due to the context states (previous hidden states) cannot be corrected. *Layered fully recurrent network* described in Section 1.2.3 does not have this limitation. It has the same feedback mechanism as that of the Elman network but context units are not used.

1.2.3 Layered fully recurrent networks

1.2.3.1 Model description

We consider a rather general scheme: a three-layer feedforward network with a fully-connected hidden layer termed layered fully recurrent network [Pedersen95]. This network includes the conventional feedforward network as a special case. Figure 1-6 shows the structure of the layered fully recurrent network. The arrows in the figure represent the weights and propagation directions. W_{ij}, W_{i0}, R_{jr}, w_{jk} and w_{j0} represent the hidden-to-output, output bias, recurrent, input-to-hidden and hidden bias weights respectively. The subscripts i, j and k are the indexes for the units in

the output, hidden and input layers respectively and the subscript 0 represents the bias. The units in the input layer fully connect to the units in the hidden layer. The units in the hidden layer are self-connected and fully connect to the other units in the same layer. They also fully connect to the units in the output layer. Usually, semilinear (in our case, tanh) transfer function is used in the hidden units and linear transfer function is used in the output units.

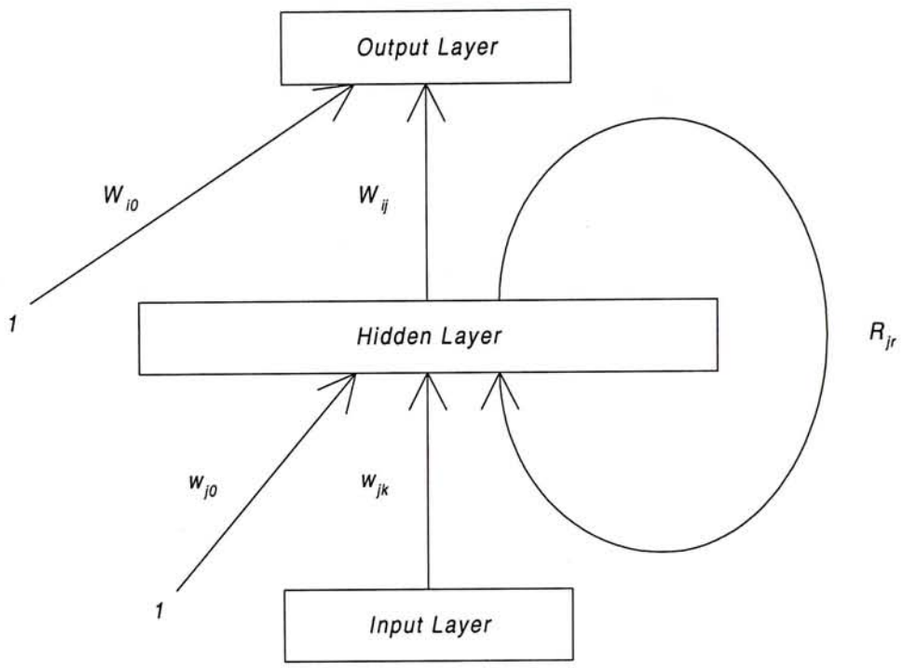


Figure 1-6 Layered fully recurrent neural network

At time t, the input unit k is clamped to the value $\xi_k(t)$. The hidden unit j thus receives a net input

$$h_j(t) = \sum_k w_{jk}\xi_k(t) + w_{j0} + \sum_r R_{jr}V_r(t-1) \tag{1.2}$$

and produces output

$$V_j(t) = g(h_j(t)) = g(\sum_k w_{jk}\xi_k(t) + w_{j0} + \sum_r R_{jr}V_r(t-1)) \tag{1.3}$$

The output unit i receives a net input

$$d_i(t) = \sum_j W_{ij}V_j(t) + W_{i0} \tag{1.4}$$

and produces the final output

$$O_i(t) = f(d_i(t)) = f(\sum_j W_{ij}V_j(t) + W_{i0}) \tag{1.5}$$

In [Li90], the layered fully recurrent network was used to predict the sunspot number. This network was shown to produce very good results, especially when doing long-range forecasting.

1.2.3.2 Our selection and motivation

The layered fully recurrent network described in this section is more flexible compared with the neural network forecasting models described in Sections 1.2.1 and 1.2.2 in modeling time series and was chosen in our study. Unlike the network using time delays, the layered fully recurrent network does not need to determine the dimension of time delay space in advance. Its feedback structure in effect takes all the past inputs into account. Unlike the network using context units, the dependence of previous hidden states on the weights is not ignored during weight updating. So, errors due to the previous hidden states can be corrected.

1.2.4 Other models

There are many different types of recurrent models. For example, [Shimohara88] discussed the possibilities of different feedback mechanisms (feedback sources and feedback destinations) in layer level. Feedback structures incorporated in synapse level were discussed in [Tsoi94]. These structures include local synapse feedback, local activation feedback and local output feedback. [Mozer93] considered the possibilities of different memory forms. These forms include delay, exponential and gamma. [Horne95] investigated some single layer recurrent networks using higher order connections. Examples are higher order network [Giles90], bilinear network and quadratic network.

1.3 Learning methods [Battiti92, Bishop95, Hertz91, Press88, Shepherd97]

After choosing the forecasting model, we then train the model to predict the future values of the time series. There are many ways of doing it, which are known as the *learning methods*. In this section, some of them are described.

First, we define a cost function to measure the performance of the forecasting model. The cost function is usually defined as the deviations between the target and the network output values. Sum-of-squares error function shown in Equation 1.6 is often used as the cost function E.

$$E = \sum_t E_t = \frac{1}{2}\sum_{ti} (\zeta_i(t) - O_i(t))^2 \qquad (1.6)$$

where $O_i(t)$ is the output value of the ith output unit at time t

$\zeta_i(t)$ is the target value for the ith output unit at time t

w is the weight vector comprising all the network weights

The objective of the learning process is to adjust the weight vector **w** of the model so as to minimize the cost function E. We have many different kinds of learning methods such as evolutionary algorithm [Angeline94, Fogel91], genetic algorithm [Koehn94], first order and second order methods [Battiti92, Bishop95, Hertz91, Press88, Shepherd97] to accomplish this task. In these examples, the first two methods aim at finding the global minimum while the latter two methods are guaranteed to locate a local minimum only. Among them, the first order and second order methods are more popular in the neural network training because of their fast learning speed. In the next section, we will describe more about these two methods.

1.3.1 First order and second order methods

The first order and second order methods are the numerical unconstrained minimization techniques [Dennis83]. These methods derive from the Taylor series expansion of the cost function E(**w**) in the neighborhood of an arbitrary point **w**.

$$E(\mathbf{w}+\mathbf{s}) = E(\mathbf{w}) + \mathbf{g}(\mathbf{w})^T\mathbf{s} + \tfrac{1}{2}\,\mathbf{s}^T\mathbf{H}(\mathbf{w})\mathbf{s} + \ldots \qquad (1.7)$$

where E is the cost function

w is the Wt×1 weight vector comprising all the network weights and the weights are indexed from 1 to Wt.

$$\mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_{Wt} \end{pmatrix} \qquad (1.8)$$

Wt is the number of network weights

g is the Wt×1 gradient vector (first partial derivatives of cost function E with respect to all network weights)

$$\mathbf{g} = \frac{\partial E}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_{Wt}} \end{pmatrix} \tag{1.9}$$

H is the Wt×Wt Hessian matrix (second partial derivatives of cost function E with respect to all network weights)

$$\mathbf{H} = \frac{\partial^2 E}{\partial \mathbf{w}^2} = \begin{pmatrix} \frac{\partial^2 E}{\partial w_1 \partial w_1} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_1 \partial w_{Wt}} \\ \frac{\partial^2 E}{\partial w_2 \partial w_1} & \frac{\partial^2 E}{\partial w_2 \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_2 \partial w_{Wt}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 E}{\partial w_{Wt} \partial w_1} & \frac{\partial^2 E}{\partial w_{Wt} \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_{Wt} \partial w_{Wt}} \end{pmatrix} \tag{1.10}$$

If the methods derive from the linear model,

$$E(\mathbf{w}+\mathbf{s}) \approx E(\mathbf{w}) + \mathbf{g}(\mathbf{w})^T\mathbf{s} \tag{1.11}$$

they are known as first order methods. One of the examples is *gradient descent method*. Its updating rule is

$$\Delta\mathbf{w} = -\mu\mathbf{g} \tag{1.12}$$

where μ is the step length parameter. This method has many variations, for example, using various adaptive learning rate strategies and using momentum.

If the methods derive from the quadratic model,

$$E(\mathbf{w}+\mathbf{s}) \approx E(\mathbf{w}) + \mathbf{g}(\mathbf{w})^T\mathbf{s} + \tfrac{1}{2}\,\mathbf{s}^T\mathbf{H}(\mathbf{w})\mathbf{s} \tag{1.13}$$

they are known as second order methods. One of the examples is the *Newton's method*. Its updating rule is

$$\Delta\mathbf{w} = -\mathbf{H}^{-1}\mathbf{g} \tag{1.14}$$

Other examples are the *Quasi-Newton method*, the *'memoryless' Quasi-Newton method* and the *conjugate gradient method.*

The mentioned examples in this section are designed to minimize the general nonlinear cost function. In the next section, we present a special case of

unconstrained minimization known as the *nonlinear least squares methods*, which are designed to minimize the sum-of-squares function only.

1.3.2 Nonlinear least squares methods

Nonlinear least squares methods are designed to minimize the sum-of-squares error function in Equation 1.6 and cost function in the form of sum-of-squares nonlinear functions. Examples of these methods are the *Gauss-Newton method*, the *damped Gauss-Newton method*, the *Levenberg-Marquardt method* and the *full Newton-type method.*

Let $e_i(t) = \zeta_i(t) - O_i(t)$ and $\mathbf{e}$ be a T·P×1 vector with elements $e_i(t)$

$$\mathbf{e} = \begin{pmatrix} \mathbf{e}(1) \\ \mathbf{e}(2) \\ \vdots \\ \mathbf{e}(T) \end{pmatrix} \text{ and } \mathbf{e}(t) = \begin{pmatrix} e_1(t) \\ e_2(t) \\ \vdots \\ e_P(t) \end{pmatrix} \tag{1.15}$$

Then the sum-of-squares error function in Equation 1.6 can be written as

$$E = \frac{1}{2}\mathbf{e}^T\mathbf{e} \tag{1.16}$$

By differentiating Equation 1.16, we obtain the gradient vector expression

$$\mathbf{g} = \nabla E = \mathbf{J}^T\mathbf{e} \tag{1.17}$$

By differentiating Equation 1.17 once more, we obtain the Hessian matrix expression

$$\mathbf{H} = \nabla^2 E = \mathbf{J}^T\mathbf{J} + \mathbf{S} \tag{1.18}$$

where $\mathbf{J}$ is the T·P×Wt Jacobian matrix with elements $\frac{\partial e_i(t)}{\partial w_j}$

$$\mathbf{J} = \frac{\partial \mathbf{e}}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial \mathbf{e}(1)}{\partial w_1} & \frac{\partial \mathbf{e}(1)}{\partial w_2} & \cdots & \frac{\partial \mathbf{e}(1)}{\partial w_{Wt}} \\ \frac{\partial \mathbf{e}(2)}{\partial w_1} & \frac{\partial \mathbf{e}(2)}{\partial w_2} & \cdots & \frac{\partial \mathbf{e}(2)}{\partial w_{Wt}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{e}(T)}{\partial w_1} & \frac{\partial \mathbf{e}(T)}{\partial w_2} & \cdots & \frac{\partial \mathbf{e}(T)}{\partial w_{Wt}} \end{pmatrix} \text{ and } \frac{\partial \mathbf{e}(t)}{\partial w_j} = \begin{pmatrix} \frac{\partial e_1(t)}{\partial w_j} \\ \frac{\partial e_2(t)}{\partial w_j} \\ \vdots \\ \frac{\partial e_P(t)}{\partial w_j} \end{pmatrix} \tag{1.19}$$

T is the number of training data

P is the number of network outputs

$$\mathbf{S} = \sum_{ti} e_i(t) \nabla^2 e_i(t) \tag{1.20}$$

Substituting Equations 1.17 and 1.18 into Equation 1.14, we have

$$\Delta \mathbf{w} = -\mathbf{H}^{-1}\mathbf{g} = -[\mathbf{J}^T\mathbf{J} + \mathbf{S}]^{-1}\mathbf{J}^T\mathbf{e} \tag{1.21}$$

The Gauss-Newton method assumes that S≈0. Its updating rule is

$$\Delta \mathbf{w} = -[\mathbf{J}^T\mathbf{J}]^{-1}\mathbf{J}^T\mathbf{e} \tag{1.22}$$

The Gauss-Newton method of Equation 1.22 is improved by combining it with a line-search algorithm. This method is called the damped Gauss-Newton method. Its updating rule is

$$\Delta \mathbf{w} = -\mu[\mathbf{J}^T\mathbf{J}]^{-1}\mathbf{J}^T\mathbf{e} \tag{1.23}$$

where μ is the step length parameter. This method is more reliable than the Gauss-Newton method.

The Levenberg-Marquardt method is a trust-region modification of the Gauss-Newton method. Its updating rule is as follows.

$$\Delta \mathbf{w} = -[\mathbf{J}^T\mathbf{J} + \lambda\mathbf{I}]^{-1}\mathbf{J}^T\mathbf{e} \tag{1.24}$$

where $\mathbf{I}$ is the Wt×Wt unit matrix

λ is the learning parameter

The Gauss-Newton method, the damped Gauss-Newton method and the Levenberg-Marquardt method described in Equations 1.22 to 1.24 ignore the term **S**. If **S** is small relative to $\mathbf{J}^T\mathbf{J}$, these methods are locally quadratically convergent. However, **S** is relative large for the problems that are very nonlinear and have comparatively large errors **e** at the solution. If so, the convergence speed of these methods becomes slow.

The full Newton-type method approximates **S** in Equation 1.20 by the quasi-Newton approximation. The convergence speed of this method will not deteriorate when this method is applied to problems that are very nonlinear and have relative large errors. However, this method is more complex.

In the following sections, the algorithm of the Levenberg-Marquardt method and our motivation for selecting it as our learning method will be described.

1.3.2.1 Levenberg-Marquardt method – our selection and motivation

As described in Section 1.3.2, the Levenberg-Marquardt method is one of the nonlinear least squares methods. Several factors make us to choose the Levenberg-Marquardt method as our training method. First, the Levenberg-Marquardt method is well defined when **J** in Equation 1.19 does not have full column rank (this happens if the number of training data is less than the number of weights). Second, when the Gauss-Newton update in Equation 1.22 is much too long, the Levenberg-Marquardt update in Equation 1.24 is often superior to the damped Gauss-Newton update in Equation 1.23 [Dennis83]. Last, the Levenberg-Marquardt method performs well in practice. In the study of [Shepherd97], many different first order and second order methods were compared. The Levenberg-Marquardt method was the fastest training method with the zero-residual N-parity problem and with the small-residual sin(x)cos(2x) problem, but the slowest second-order method with the larger-residual sin(x) problem. It also performs well in the N-parity problem when the scale of the problem increases. Moreover, [Hagan94] reported that the Levenberg-Marquardt method was very efficient compared with the backpropagation with variable learning rate and conjugate gradient backpropagation. These methods were tested on the sine wave, square wave, 2-D sinc function and 4-D function approximation problems.

In the following section, the algorithm of the Levenberg-Marquardt method will be described.

1.3.2.2 Levenberg-Marquardt method – algorithm

The Levenberg-Marquardt update is shown in Equation 1.24. In this equation, the parameter λ controls both the magnitude and direction of $\Delta\mathbf{w}$. As λ increases, the magnitude of $\Delta\mathbf{w}$ decreases and the direction of $\Delta\mathbf{w}$ changes gradually from the Gauss-Newton direction to the negative gradient direction.

The setting of the parameter λ during the minimization process is shown in Algorithm 1.1.

Algorithm 1.1 Levenberg-Marquardt method

1. $\lambda = 0.001$, $\beta = 10$, iteration = 0 and finished = false
2. WHILE finished = false
3. calculate $\Delta\mathbf{w} = -(\mathbf{J}^T\mathbf{J} + \lambda\,\mathbf{I})^{-1}\mathbf{J}^T\mathbf{e}$
4. calculate $E(\mathbf{w}+\Delta\mathbf{w})$
5. WHILE ($\lambda \leq$ maximum_λ) AND ($E(\mathbf{w}+\Delta\mathbf{w}) \geq E(\mathbf{w})$)
6. increase λ by a factor β
7. calculate $\Delta\mathbf{w}$
8. calculate $E(\mathbf{w}+\Delta\mathbf{w})$
9. END
10. IF $\lambda \leq$ maximum_λ
11. update $\mathbf{w}$ ($\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$)
12. decrease λ by a factor β
13. iteration $\leftarrow$ iteration + 1
14. END
15. calculate $\mathbf{J}^T\mathbf{e}$
16. IF (iteration > maximum_iteration) OR
    (minimum of validation error is reached = true) OR
    ($\mathbf{J}^T\mathbf{e}$ < minimum_gradient) OR ($\lambda >$ maximum_λ)
17. finished = true
18. END
19. END

In Step 1, λ is initialized to a small value such that the approximate Gauss-Newton step is tried first. If the cost function E increases after taking the Levenberg-Marquardt step, λ is increased by a factor β (Step 5) and the cost function E is recomputed (Step 6). This process is repeated until a decrease in the cost function E is obtained. If the cost function E decreases after taking the Levenberg-Marquardt step, the new weight vector is stored (Step 9) and λ is decreased by a factor β (Step 10). The whole algorithm is repeated until the termination criteria in Step 13 are satisfied.

1.3.3 Batch mode, semi-sequential mode and sequential mode of updating

The learning methods can be used in different modes, namely the *batch mode*, *semi-sequential mode* and *sequential mode of updating*. Batch mode learning methods update weights for all training data at a time. Semi-sequential mode learning methods update weights for part of the training data at a time. Sequential mode learning methods update weights for one datum at a time. Semi-sequential and sequential approaches are more efficient if the data is highly redundant. Moreover, they have the possibilities to escape from local minima since they are stochastic algorithms.

1.4 Jacobian matrix calculations in recurrent networks

The key step in the Levenberg-Marquardt algorithm is the computation of the Jacobian matrix, which is defined in Equation 1.19. In this section, two different computations of the Jacobian matrix will be described. These computations are similar to the calculations of gradient vector in the *Real Time Back-Propagation Through Time* (RTBPTT) algorithm [Williams90] and the *Real Time Recurrent Learning* (RTRL) [Williams89] algorithm. We call them the *RTBPTT-like* and *RTRL-like* calculations respectively and they are described in Sections 1.4.1 and 1.4.2 respectively.

1.4.1 RTBPTT-like Jacobian matrix calculation

The derivation of the RTBPTT-like Jacobian matrix calculation is as follows. First the temporal operation of the recurrent network is unfolded into a multi-layer feedforward network. Take the layered fully recurrent network in Figure 1-6 as an example. The t time step operation of the recurrent network is unfolded into a feedforward network with t hidden layers, which is shown in Figure 1-7. In this unfolded feedforward network, each weight has t number of copies. Take the recurrent weight R_{jr} as an example. Its t copies are labeled as $R_{jr}(1)$, $R_{jr}(2)$, $R_{jr}(3)$, …, $R_{jr}(t)$. The Jacobian element $\frac{\partial e_i(t)}{\partial R_{jr}}$ (first derivative of $e_i(t)$ with respect to the recurrent weight R_{jr}) is calculated by summing all the first derivatives of $e_i(t)$ with respect to the unfolded recurrent weights $R_{jr}(1)$, $R_{jr}(2)$, $R_{jr}(3)$, …, $R_{jr}(t)$. That is,

$$\frac{\partial e_i(t)}{\partial R_{jr}} = \sum_{\tau=1}^{t} \frac{\partial e_i(t)}{\partial R_{jr}(\tau)} \tag{1.25}$$

Other weights are calculated similarly. Based on this method, it can be shown that Jacobian element (derivative of $e_i(t)$) with respect to the hidden-to-output weight

$$\frac{\partial e_i(t)}{\partial W_{ij}} = \delta_i(t) V_j(t) \tag{1.26}$$

Jacobian element (derivative of $e_i(t)$) with respect to the output bias weight

$$\frac{\partial e_i(t)}{\partial W_{i0}} = \delta_i(t) \tag{1.27}$$

Jacobian element (derivative of $e_i(t)$) with respect to the recurrent weight

$$\frac{\partial e_i(t)}{\partial R_{jj'}} = \sum_{\tau=1}^{t} \delta_{ij}(\tau) V_{j'}(\tau - 1) \tag{1.28}$$

Jacobian element (derivative of $e_i(t)$) with respect to the input-to-hidden weight

$$\frac{\partial e_i(t)}{\partial w_{jk}} = \sum_{\tau=1}^{t} \delta_{ij}(\tau) \xi_k(\tau) \tag{1.29}$$

Jacobian element (derivative of $e_i(t)$) with respect to the hidden bias weight

$$\frac{\partial e_i(t)}{\partial w_{j0}} = \sum_{\tau=1}^{t} \delta_{ij}(\tau) \tag{1.30}$$

where

$$\delta_i(t) = -f'(d_i(t)) \tag{1.31}$$

$$\delta_{ij}(\tau) = \begin{cases} g'(h_j(\tau)) W_{ij} \delta_i(\tau) & \tau = t \\ g'(h_j(t)) \sum_{j'} R_{j'j} \delta_{ij'}(\tau + 1) & \tau < t \end{cases} \tag{1.32}$$

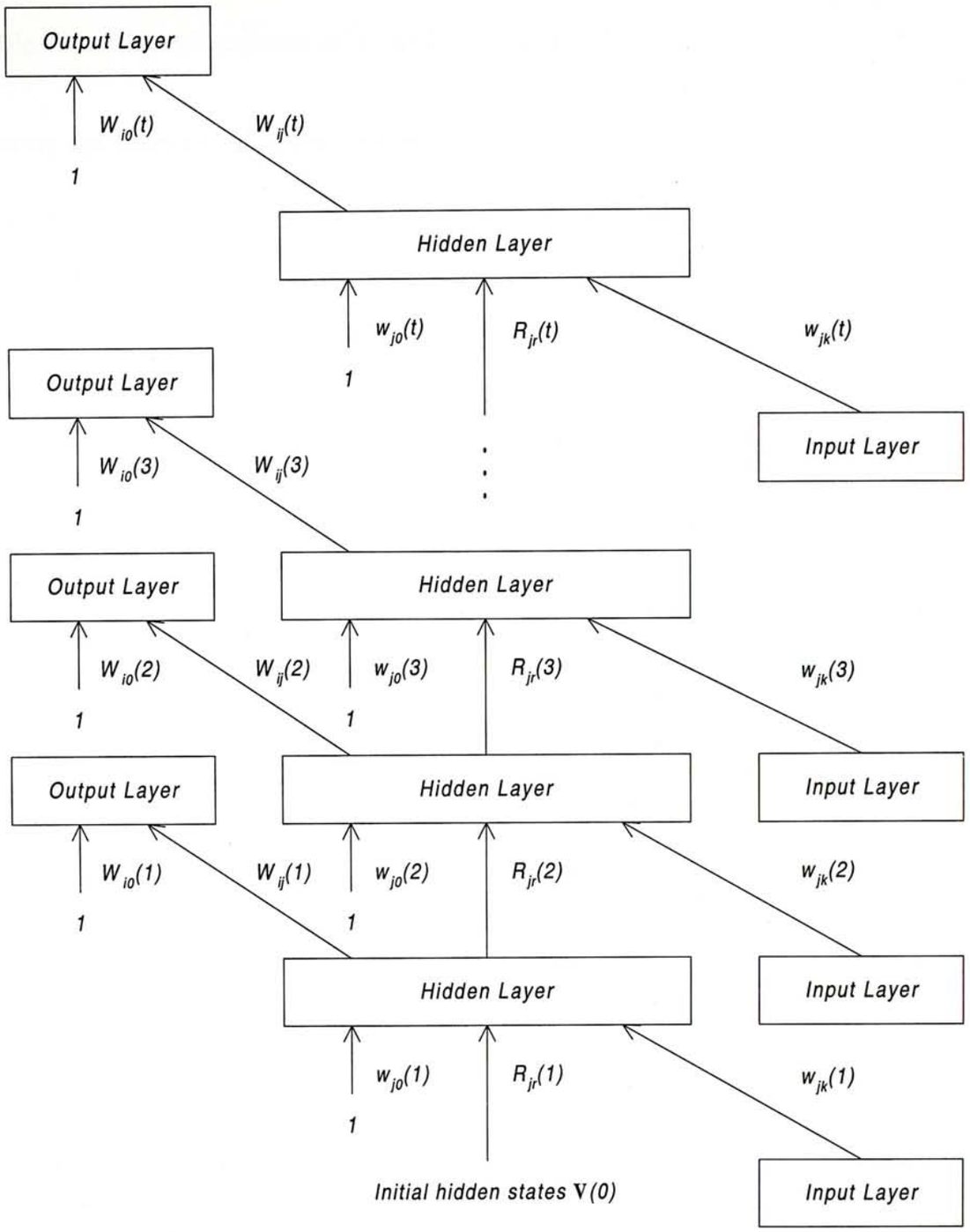


Figure 1-7 Unfolding the t time step operation of the recurrent network in Figure 1.6 into a feedforward network with t hidden layers.

1.4.2 RTRL-like Jacobian matrix calculation

The RTRL-like method calculates the Jacobian elements without duplicating the units and weights. Its calculation of Jacobian elements at time t $\frac{\partial e_i(t)}{\partial w}$ (first derivative of $e_i(t)$ with respect to the network weight w) are expressed in terms of the derivatives of hidden states at time t-1, $\frac{\partial \mathbf{V}(t-1)}{\partial w}$. The derivatives of $\frac{\partial \mathbf{V}(t-1)}{\partial w}$ are

available from the calculations of Jacobian elements at time t-1, $\frac{\partial e_i(t-1)}{\partial w}$. When calculating the Jacobian elements at time 1, we assume that $\frac{\partial \mathbf{V}(0)}{\partial w} = 0$. The RTRL-like calculations of the Jacobian elements with respect to the recurrent, input-to-hidden and hidden bias weights are shown in Equations 1.33 to 1.35 respectively.

$$\frac{\partial e_i(t)}{\partial R_{j'r'}} = -f'(d_i(t))\sum_j W_{ij}\frac{\partial V_j(t)}{\partial R_{j'r'}} \tag{1.33}$$

where $\frac{\partial V_j(t)}{\partial R_{j'r'}} = g'(h_j(t))\left(V_{r'}(t-1)\delta_{jj'} + \sum_r R_{jr}\frac{\partial V_r(t-1)}{\partial R_{j'r'}} \right)$ and $\frac{\partial V_r(0)}{\partial R_{j'r'}} = 0$

$\delta_{jj'}$ denotes the Kronecker delta

$$\frac{\partial e_i(t)}{\partial w_{j'k'}} = -f'(d_i(t))\sum_j W_{ij}\frac{\partial V_j(t)}{\partial w_{j'k'}} \tag{1.34}$$

where $\frac{\partial V_j(t)}{\partial w_{j'k'}} = g'(h_j(t))\left(\xi_{k'}(t)\delta_{jj'} + \sum_r R_{jr}\frac{\partial V_r(t-1)}{\partial w_{j'k'}} \right)$ and $\frac{\partial V_r(0)}{\partial w_{j'k'}} = 0$

$$\frac{\partial e_i(t)}{\partial w_{j'0}} = -f'(d_i(t))\sum_j W_{ij}\frac{\partial V_j(t)}{\partial w_{j'0}} \tag{1.35}$$

where $\frac{\partial V_j(t)}{\partial w_{j'0}} = g'(h_j(t))\left(\delta_{jj'} + \sum_r R_{jr}\frac{\partial V_r(t-1)}{\partial w_{j'0}} \right)$ and $\frac{\partial V_r(0)}{\partial w_{j'0}} = 0$

The Jacobian elements (first derivatives of $e_i(t)$) with respect to the hidden-to-output and output bias weights do not depend on the derivatives of hidden states. The RTRL-like calculations of these Jacobian elements are the same as the RTBPTT-like calculations shown in Equations 1.26 and 1.27.

1.4.3 Comparison between RTBPTT-like and RTRL-like calculations

The number of operations of the RTBPTT-like calculation increases with time t. On average, the RTBPTT-like calculation requires $O(N^2T)$ operations per time step where N is the number of fully recurrent hidden units and T is the number of training data. The number of operations of the RTRL-like calculation is even across time t. The RTRL-like calculation requires $O(N^4)$ operations per time step. If the number of training data is small ($T<N^2$), RTBPTT-like calculation is more efficient. Otherwise, RTRL-like calculation is more efficient.

If the weights are updated in batch mode, the Jacobian matrices that we use the RTBPTT-like and RTRL-like methods to calculate are the same. However, if the weights are updated in sequential mode, the Jacobian matrices that we use the RTBPTT-like and RTRL-like methods to calculate diverge. The reason is that in the RTRL-like calculation, the calculation of Jacobian elements at time t depends on the calculation at time t-1. During this time, the weights are assumed to be constant. However, this assumption is not true if the weights are updated in sequential mode. The error accumulated will not be negligible if the number of training data is very large. In [Catfolis93], the author remedied this problem by re-initializing the derivatives $\frac{\partial \mathbf{V}(t)}{\partial w}$ after a certain number of time steps. The RTBPTT-like calculation does not have this problem because the calculation at time t does not depend on the calculations of the previous time steps.

1.5 Computation complexity reduction techniques in recurrent networks

Neural network training is an iterative process. It usually requires a number of iterations to arrive at the solution, which cost a lot of time. This hinders the application of neural network to large problems. So, a more efficient learning method is required. One way to accomplish this task is to reduce the computation complexity of the learning method. A number of techniques were devised for doing this. These techniques are divided into the architectural and algorithmic approaches, which are described in Sections 1.5.1 and 1.5.2 respectively.

1.5.1 Architectural approach

Architectural approaches reduce the computation complexity by altering the architecture of the network. Examples are the *recurrent connection reduction method* [Chan95, Bengio89, Frasconi92], *treating the feedback signals as additional inputs method* [Elman90, Jordan86, Wilson95, Zipser89] and *growing network method* [Fahlman91]. These methods are described in Sections 1.5.1.1 to 1.5.1.3 respectively.

1.5.1.1 Recurrent connection reduction method

One way to reduce the computation complexity dramatically is to reduce the number of recurrent connections at the expense of generality of network architecture. Examples are the *locally connected recurrent neural network* [Chan95] and the *local feedback multilayered network* [Bengio89, Frasconi92]. They are described as follows.

Locally connected recurrent neural network

[Chan95] considered a class of recurrent connection reduction network topology called the locally connected recurrent neural network. The recurrent links of each hidden unit are connected to its neighborhood units only. Depending on the definition of neighborhood units, the network can be in ring, grid, cubic or other structures.

Local feedback multilayered network

Local feedback multilayered network described in [Bengio89] and [Frasconi92] is a special case of locally connected recurrent neural network. Only the self-recurrent connections are assumed.

1.5.1.2 Treating the feedback signals as additional inputs method

Another way to reduce the computation complexity is to treat the feedback signals as additional inputs so that the dependence of the feedback signals on the weights during weight updating is ignored. This technique was used in the networks using context units described in Section 1.2.2. Sub-grouping method [Zipser89] also applied this technique to some of the feedback signals. These examples are explained as follows.

Context units

Elman network [Elman90], Jordan network [Jordan86] and tower-recurrent networks [Wilson95] use context units to feed the past information back to the networks. The activations of the context units, that is the feedback signals, are treated as additional inputs to the networks.

Sub-grouping method [Zipser89]

Sub-grouping method reduces the computation complexity by treating some of the feedback signals as additional inputs. These feedback signals are chosen depending on the definition of the sub-networks. For example, a fully recurrent network with N hidden units and M inputs is divided into g fully recurrent sub-networks each with N/g hidden units (assuming that g is a factor of N). Each hidden unit in a sub-network will receive the activations of the N-N/g hidden units in the other subgroups as inputs in addition to the original M inputs. The effect of sub-grouping is to reduce the size of recurrently connected units from N to N/g.

1.5.1.3 Growing network method

In growing network method like the *recurrent cascade-correlation architecture* [Fahlman91], new hidden units with self-recurrent link are added one by one and are frozen once they are added to the network. Only the weights connected to the new hidden units are learned at a time. This reduces the computation complexity dramatically.

1.5.2 Algorithmic approach

Algorithmic approaches reduce the computation complexity without altering the architecture of the network. Examples are the *history cutoff method* [Williams90] and *changing the updating frequency from sequential mode to semi-sequential mode method* [Schmidhuber92]. They are described in Sections 1.5.2.1 and 1.5.2.2 respectively.

1.5.2.1 History cutoff method

Truncated Back-Propagation Through Time, BPTT(h) algorithm [Williams90]

Truncated Back-Propagation Through Time algorithm termed BPTT(h) algorithm uses a bounded-history approximation to the RTBPTT algorithm. The information is saved for a fixed number h of time steps and any information older than that is forgotten. Take the calculation of Jacobian element $\frac{\partial e_i(t)}{\partial R_{jr}}$ shown in Equation 1.25 as an example. In the BPTT(h) algorithm, the Jacobian element is calculated as follows.

$$\frac{\partial e_i(t)}{\partial R_{jr}} = \sum_{\tau=t-h+1}^{t} \frac{\partial e_i(t)}{\partial R_{jr}(\tau)} \tag{1.36}$$

1.5.2.2 Changing the updating frequency from sequential mode to semi-sequential mode method

Hybrid of BPTT and RTRL [Schmidhuber92]

The RTRL algorithm requires $O(N^4)$ operations per time step where N is the number of fully recurrent hidden units. Hybrid of BPTT and RTRL method reduces the computation complexity of the RTRL algorithm by changing the weight updating frequency from every time step to every h' time steps. This method computes the cumulative error gradient by means of the Back-Propagation Through Time (BPTT) -like calculation once every block of h' time steps. The RTRL-like calculation is used to encapsulate the history before the start of each block. If h' is chosen to be O(N), the average computational complexity per time step is reduced to $O(N^3)$.

1.6 Motivations for using block-diagonal Hessian matrix

In the last section, we reviewed a number of computation complexity reduction techniques used in the recurrent neural network training. In this thesis, we proposed using the block-diagonal Hessian matrix to reduce the computation complexity in the recurrent network training. The proposed method is algorithmic since it does not alter the architecture of the recurrent network. In the following, our motivations for using the proposed method are described.

Several factors make us to choose the method of using the block-diagonal Hessian matrix. First, the proposed method requires less operations and storage. For example, training a recurrent network with the Levenberg-Marquardt algorithm requires $O(N^4T)$ operations per epoch in batch mode calculation (or $O(N^6)$ operations per time step in sequential mode calculation) and $O(N^4)$ storage, where N is the number of fully recurrent hidden units and T is the number of training data. Using the proposed method requires less operations per epoch / time step (ranging from $O(N^3T)$ to $O(N^4T)$ in batch mode calculation or ranging from $O(N^4)$ to $O(N^6)$ in sequential mode calculation) and less storage (ranging from $O(N^2)$ to $O(N^4)$). Second, weight updates of different groups can be calculated independently

(explained in Chapter 2). This property makes the parallel processing possible. Finally, the proposed method performs well in the feedforward neural network training. For example, diagonal Hessian matrix, a special case of block-diagonal Hessian matrix, was used in [Becker88], [Fahlman88] and [Ricotti88]. Using diagonal Hessian matrix was shown to learn faster than the backpropagation with fixed learning rate and/or momentum on a random classification problem [Becker88], an encoder/decoder task [Fahlman88], the XOR problem [Fahlman88, Ricotti88] and a word stress determination application [Ricotti88]. Block-diagonal Hessian matrix was used in [Kollias88, 89] and [Wille97]. [Kollias88, 89] ignored the Hessian elements (second partial derivatives of the cost function E) with respect to the incoming weights of different neurons and [Wille97] suggested ignoring the Hessian elements (second partial derivatives of the cost function E) with respect to the weights of different layers. In [Kollias88, 89], the performance was shown to be better than performance of the backpropagation with fixed learning rate and momentum on the problem of XOR.

1.7 Objective

In this chapter, we proposed applying the Levenberg-Marquardt method with the block-diagonal Hessian matrix to the recurrent neural network training. In the remaining chapters, the objective is to assess the performance of the Levenberg-Marquardt method with the block-diagonal Hessian matrix in the recurrent neural network training. It also includes the identification of the factors of the proposed method and evaluation of the effects of these factors on the performance.

1.8 Organization of the thesis

In the next chapter, we will identify the factors of the proposed method, which includes the choices of block-diagonal Hessian matrices and weight updating methods. To assess the performance of the proposed method, we evaluated the method on three time series prediction problems. These problems are described in Chapter 3. The setup of the evaluation is also included in this chapter. The performance of the proposed method depends on the choices of the *updating methods*, *numbers and sizes of the blocks of the block-diagonal Hessian matrix* and *weight-grouping methods of the block-diagonal Hessian matrix*. (These factors are explained in Chapter 2.) The effects of these factors on the performance are studied in Chapters 4, 5 and 6 respectively. Finally, discussion and conclusion are given in Chapters 7 and 8 respectively.

Chapter 2 Learning with the block-diagonal Hessian matrix

2.1 Introduction

In the last chapter, we proposed applying the technique of block-diagonal Hessian approximation to the recurrent neural network training, which can reduce both the computation and storage complexities. We need to consider two factors when this technique is applied.

i. One is the choice of block-diagonal Hessian matrices. In Section 2.2, the general form and factors of the block-diagonal Hessian matrices will be described. In Section 2.3, we will describe four particular block-diagonal Hessian matrices, which will be studied in this thesis.

ii. The other factor is the choice of weight updating methods, which is described in Section 2.4.

2.2 General form and factors of block-diagonal Hessian matrices

2.2.1 General form of block-diagonal Hessian matrices

The Hessian matrix $\mathbf{H}$ of the cost function E has an entry $\frac{\partial^2 E}{\partial w_i \partial w_j}$, the second partial derivative, in the (i,j)-place where E is defined in Equation 1.6. It is a Wt×Wt matrix where Wt is the number of network weights. The *block-diagonal Hessian matrix* approximates to the Hessian matrix by letting the block-diagonal entries be equal to the corresponding entries in the Hessian matrix and the off-block-diagonal entries be equal to zeros. It is of the form

$$\mathbf{H}_{\text{block-diag.}} = \begin{pmatrix} \mathbf{H}_1 & 0 & 0 & 0 \\ 0 & \mathbf{H}_2 & 0 & 0 \\ 0 & 0 & \ldots & 0 \\ 0 & 0 & 0 & \mathbf{H}_B \end{pmatrix} \qquad (2.1)$$

where $\mathbf{H}_i$ is a $q_i \times q_i$ matrix for each i = 1, 2, … B.

$\sum_{i=1}^{B} q_i$ = the number of network weights, Wt.

Each sub-matrix $\mathbf{H}_i$ is called a *block*. Its number of rows or columns q_i is called the *block size*. The elements of the blocks are called the *non-zero Hessian elements*. In Equation 2.1, the total number of non-zero Hessian elements is $q_1^2+q_2^2+ \ldots +q_B^2$. Let us partition the weight vector $\mathbf{w}$ as $[\mathbf{w}_1^T\ \mathbf{w}_2^T \ldots \mathbf{w}_B^T]^T$ where $\mathbf{w}_i$ is a $q_i \times 1$ vector, such that $\mathbf{H}_i$ can be written as $\frac{\partial^2 E}{\partial \mathbf{w}_i \partial \mathbf{w}_i}$. We call $\mathbf{w}_i$ the *weight vector of the ith block* and its elements the *weights of the ith block.*

In particular, the Levenberg-Marquardt Hessian approximation described in Section 1.3.2 is considered.

$$\mathbf{H}_{full} = \mathbf{J}^T\mathbf{J} \tag{2.2}$$

where $\mathbf{J}$ is the Jacobian matrix defined in Equation 1.19

Since Equation 2.2 calculates all the Hessian elements, we call it *full Hessian matrix* to distinguish it from the block-diagonal Hessian matrix, which calculates Hessian elements in the block-diagonal region only.

Let us partition $\mathbf{J}$ as $[\mathbf{J}_1\ \mathbf{J}_2 \ldots \mathbf{J}_B]$ where $\mathbf{J}_i$ is a $T \cdot P \times q_i$ matrix.

$$\mathbf{J}_i = \frac{\partial \mathbf{e}}{\partial \mathbf{w}_i} \tag{2.3}$$

where $\mathbf{e}$ is the error vector defined in Equation 1.15

T is the number of training data

P is the number of network outputs

The ith block $\mathbf{H}_i$ in Equation 2.1 can then be written as

$$\mathbf{H}_i = \mathbf{J}_i^T\mathbf{J}_i \tag{2.4}$$

and the general form of the block-diagonal approximation to Equation 2.2 is

$$\mathbf{H}_{block-diag.} = \begin{pmatrix} \mathbf{J}_1^T\mathbf{J}_1 & 0 & 0 & 0 \\ 0 & \mathbf{J}_2^T\mathbf{J}_2 & 0 & 0 \\ 0 & 0 & \ldots & 0 \\ 0 & 0 & 0 & \mathbf{J}_B^T\mathbf{J}_B \end{pmatrix} \tag{2.5}$$

where

$$\mathbf{J}_i^T\mathbf{J}_j = 0,\ i \neq j \tag{2.6}$$

2.2.2 Factors of block-diagonal Hessian matrices

Equations 2.1 and 2.5 show the general form of block-diagonal Hessian matrices. To specify a particular block-diagonal matrix, we need to describe the following two factors

i. number and sizes of the blocks and
ii. *weight-grouping method*

They are explained as follows.

i. Number and sizes of the blocks

The first factor is the number and sizes of the blocks. It is denoted by a set $\{q_1, q_2, \ldots, q_B\}$ where q_i represents the size of the ith block. The number of elements in the set is equal to the number of blocks, B. The restrictions on this factor are as follows.

i. The number of blocks, B ≤ the number of network weights, Wt.
ii. The sum of block sizes, $q_1+q_2+ \ldots +q_B$ = the number of network weights, Wt.

In the special cases, the block-diagonal matrix becomes full Hessian matrix if B = 1 and becomes diagonal Hessian matrix if B = Wt.

ii. Weight-grouping methods

The second factor is the arrangement of the positions of elements in the weight vector **w**. This factor affects the positions of elements in the Hessian matrix and hence affects the approximation of the Hessian matrix.

Block-diagonal Hessian matrices with the following arrangements have the same Hessian elements approximating to zeros.

i. Different arrangements of positions of elements in $\mathbf{w}_i$ where $\mathbf{w}_i$ is the weight vector of the ith block for i = 1, 2, …, B.
ii. Different arrangements of positions of $\mathbf{w}_j$'s in the weight vector **w** where $\mathbf{w}_j$'s are the weight vectors of the blocks with equal number of elements.

The differences among these arrangements are excluded from the second factor.

In other words, the second factor specifies which weights are grouped in the weight vector of each block. So we call this factor the *weight-grouping method.*

The effects of the first and second factors on the performance of our proposed method are described in Chapters 5 and 6 respectively.

2.3 Four particular block-diagonal Hessian matrices

As described in the last section, block-diagonal Hessian matrices depend on the number of blocks, sizes of the blocks and weight-grouping methods, which have many varieties. This means that the number of different block-diagonal Hessian matrices approximating a Hessian matrix is very large. Among these block-diagonal Hessian matrices, we will evaluate four particular block-diagonal Hessian matrices which were either expected to have good performance or used in the previous studies [Kollias88, 89 and Wille97]. These matrices are described in Sections 2.3.1 to 2.3.4 and named as the *correlation*, *one-unit*, *sub-network* and *layer block-diagonal Hessian matrices* respectively. These matrices have one thing in common, which is described as follows.

In the layered fully recurrent network shown in Figure 1.6, we assume that linear and nonlinear transfer functions are used in the output and hidden units respectively. The cost function E is linear in the output-layer weights $\mathbf{w}_{\text{linear}}$ and nonlinear in the input-layer and recurrent-layer weights $\mathbf{w}_{\text{nonlinear}}$. Minimizing the cost function E defined in Equation 1.6 with respect to the weights $\mathbf{w}_{\text{linear}}$ and $\mathbf{w}_{\text{nonlinear}}$ can be viewed as the mixed linear-nonlinear least-squares problem. One of the solutions to this problem is that we minimize the cost function E by solving a nonlinear least-squares problem in the nonlinear variables $\mathbf{w}_{\text{nonlinear}}$. For any given values of $\mathbf{w}_{\text{nonlinear}}$, we calculate the corresponding optimal values of $\mathbf{w}_{\text{linear}}$ by solving a linear least-squares problem. This approach usually solves the mixed linear-nonlinear least-squares problems in less time and fewer function evaluations than the full nonlinear optimization of all the variables [Dennis83]. In neural network, Webb and Lowe (1988) show that, for some problems, this approach can yield better solutions, or can require less computational effort [Bishop95].

This approach requires that the linear weights $\mathbf{w}_{\text{linear}}$ and the nonlinear weights $\mathbf{w}_{\text{nonlinear}}$ can be optimized separately. This can be accomplished by applying the approximation shown in Equation 2.7.

$$\frac{\partial^2 E}{\partial \mathbf{w}_{linear} \partial \mathbf{w}_{nonlinear}} = 0 \qquad (2.7)$$

where $\mathbf{w}_{linear}$ is the weight vector containing the output-layer weights

$\mathbf{w}_{nonlinear}$ is the weight vector containing the input-layer and recurrent layer weights

In other words, the output-layer weights are grouped in the weight vector of a block. The approximation of Equation 2.7 is used in the correlation, one-unit, sub-network and layer block-diagonal Hessian matrices.

Take a recurrent network with M input, N hidden and P output units as an example. The P(N+1) output-layer weights are grouped in the weight vector of a block. This is illustrated in Figure 2-1 for the recurrent network with M = 2, N = 3 and P = 1. The bold connections are the weights that are grouped in the weight vector of a block.

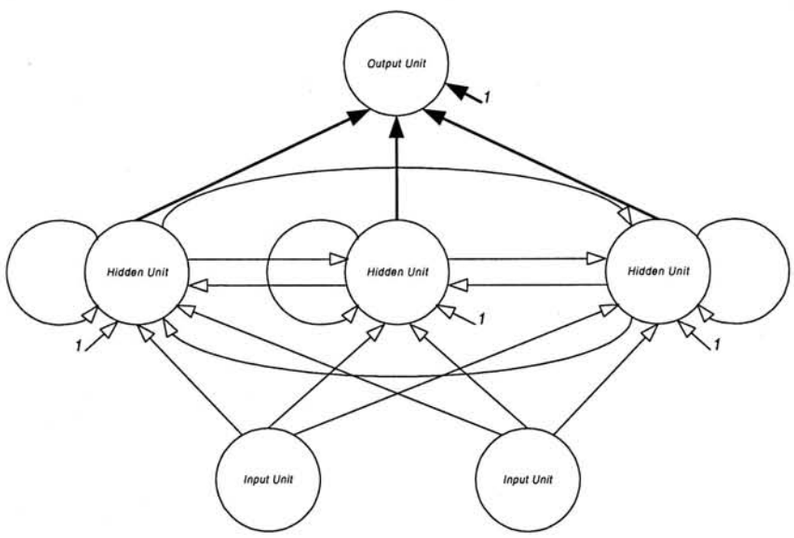


Figure 2-1 The bold hidden-to-output and output bias weights are grouped in the weight vector of a block.

2.3.1 Correlation block-diagonal Hessian matrix

One way of designing the block-diagonal Hessian matrix is to keep as much information of the full Hessian matrix as possible. In our case, we used the sum of absolute values of all the Hessian elements to measure the amount of information in a block-diagonal Hessian matrix. The formation of the correlation block-diagonal matrix is intended to achieve this goal. As described in Section 2.2, block-diagonal Hessian matrix is formed by ignoring the Hessian elements (second partial

derivatives of the cost function E) in the off-block-diagonal places. The correlation block-diagonal Hessian matrix is formed by ignoring the Hessian elements (second partial derivatives) with respect to the outgoing weights of different units. In other words, the outgoing weights of the same hidden unit or the same input unit are grouped in the weight vector of a block. This weight arrangement is called the *correlation weight-grouping method.*

The following shows the formation of the correlation block-diagonal Hessian matrix. We will first consider a simple case, a single output feedforward network. Then, we extend the result to the recurrent network.

Consider a single output feedforward network with one hidden layer. Since the feedforward network is a special case of recurrent network by letting the recurrent weights R be zeros, we use the same notation as that of the recurrent network described in Section 1.2.3. The Levenberg-Marquardt Hessian approximation to the Hessian element $H_{jk,j'k'} = \frac{\partial^2 E}{\partial w_{jk} \partial w_{j'k'}}$ described in Section 1.3.2 is shown in Equation 2.8 where w_{jk} and $w_{j'k'}$ are the weights locating between the input and hidden layer of the network. $e_i(t)$ is the error between the output and target values at time t defined in Equation 1.15.

$$H_{jk,j'k'} \approx \sum_t \frac{\partial e_i(t)}{\partial w_{jk}} \frac{\partial e_i(t)}{\partial w_{j'k'}} \tag{2.8}$$

$$= \sum_t (f'(d_i(t)) W_{ij} g'(h_j(t)) \xi_k(t)) (f'(d_i(t)) W_{ij'} g'(h_{j'}(t)) \xi_{k'}(t)) \tag{2.9}$$

$$= \sum_t \underbrace{f'(d_i(t))^2}_{\text{first term}} \underbrace{g'(h_j(t)) g'(h_{j'}(t))}_{\text{second term}} \underbrace{W_{ij} W_{ij'}}_{\text{third term}} \underbrace{\xi_k(t) \xi_{k'}(t)}_{\text{fourth term}} \tag{2.10}$$

Equation 2.9 is obtained by differentiating $e_i(t)$ with respect to w_{jk} and $w_{j'k'}$. After rearranging Equation 2.9, we obtain the expression of Hessian element shown in Equation 2.10. The first term $f'(d_i(t))^2$ is a squared function and hence positive for all t. The second term $g'(h_j(t))g'(h_{j'}(t))$ is also positive for all t since the derivative of sigmoid function is always positive. The sign of the third term $W_{ij}W_{ij'}$ is unchanged for all t. Only the last term $\xi_k(t)\xi_{k'}(t)$ changes its sign over time t. If the signs of $\xi_k(t)$ and $\xi_{k'}(t)$ are *correlated*, the magnitude of $H_{jk,j'k'}$ will be larger. That is, the signs of $\xi_k(t)$ and $\xi_{k'}(t)$ vary together or vary in opposite sign over time t. However,

if the variations of the signs of $\xi_k(t)$ and $\xi_{k'}(t)$ over time t are independent of each other, the magnitude of $H_{jk,j'k'}$ will be smaller.

If we choose k=k', the signs of $\xi_k(t)$ and $\xi_{k'}(t)$ will be correlated. This means that the magnitudes of the Hessian elements (second partial derivatives of the cost function E) with respect to the outgoing weights of the same input unit, that is k=k', are expected to be larger. As mentioned in the beginning of this section, the correlation block-diagonal Hessian matrix is formed by keeping those Hessian elements with large magnitude. So, the Hessian elements (second partial derivatives) with respect to the outgoing weights of the same input units are kept and the others are ignored. In other words, the outgoing weights of the same input unit are grouped in the weight vector of a block.

Take a feedforward network with M input, N hidden and one output units as an example. The N outgoing weights of the same input unit are grouped in the weight vector of a block. There are M+1 such weight vectors. This is illustrated in Figure 2-2 for the recurrent network with M = 2, N = 3 and one output unit. In each diagram, the bold connections are the weights that are grouped in the weight vector of a block.

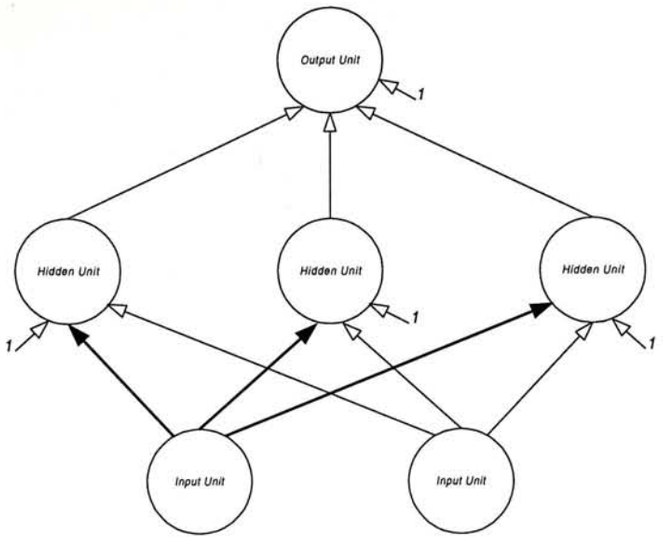
The limitation of the above analysis of Equation 2.10 is that we consider the signs of the terms in the expression of Hessian elements only, which is one of the factors that affects the magnitudes of Hessian elements. Furthermore, the analysis is applicable to feedforward weights and one output network only.

For the recurrent weights, we ignore the dependence of previous hidden states on the recurrent weights in the calculations of Hessian elements. Then, we obtain the expression of the Hessian elements (second partial derivatives of the cost function E) with respect to the recurrent weights R_{jr} and $R_{j'r'}$

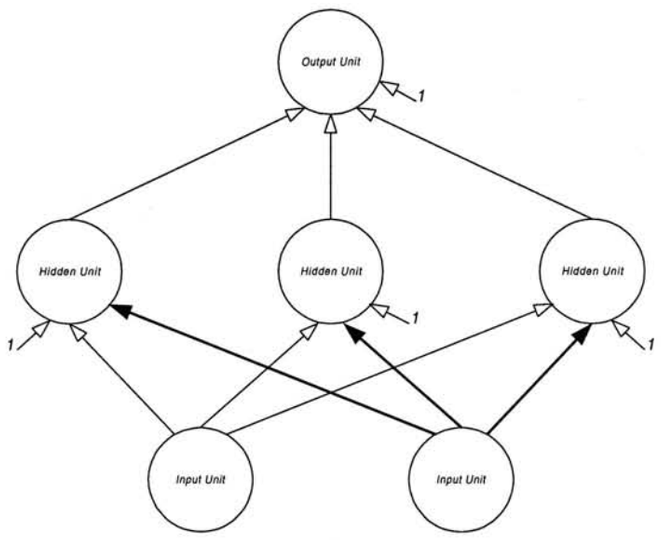
$$H_{jr,j'r'} = \frac{\partial^2 E}{\partial R_{jr} \partial R_{j'r'}} \approx \sum_t f'(d_i(t))^2 g'(h_j(t)) g'(h_{j'}(t)) W_{ij} W_{ij'} V_r(t-1) V_{r'}(t-1) \quad (2.11)$$

that is similar to Equation 2.10. We perform the analysis similar to that of Equation 2.10. We keep the Hessian elements (second partial derivatives) with respect to the outgoing weights of the same hidden units, that is r=r', and ignore the others. In other words, the outgoing weights of the same hidden unit are grouped in the weight vector of a block.

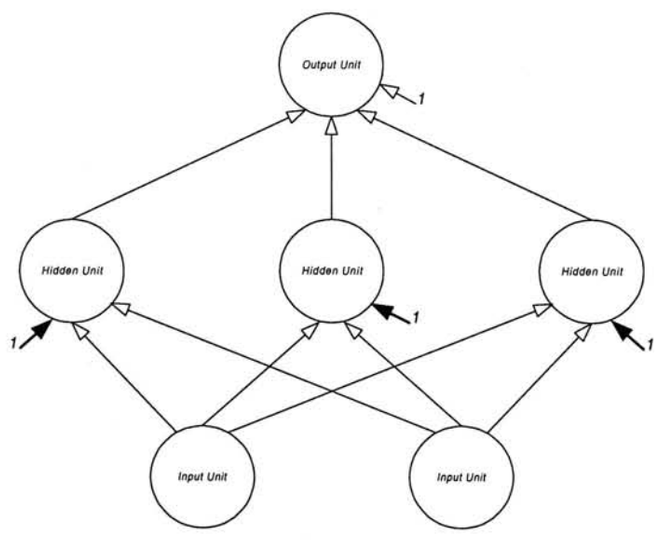
Take a recurrent network with N hidden units as an example. The N outgoing weights of the same hidden unit are grouped in the weight vector of a block. There are N such weight vectors. This is illustrated in Figure 2-3 for the recurrent network with N = 3. In each diagram, the bold connections are the weights that are grouped in the weight vector of a block.



(a)

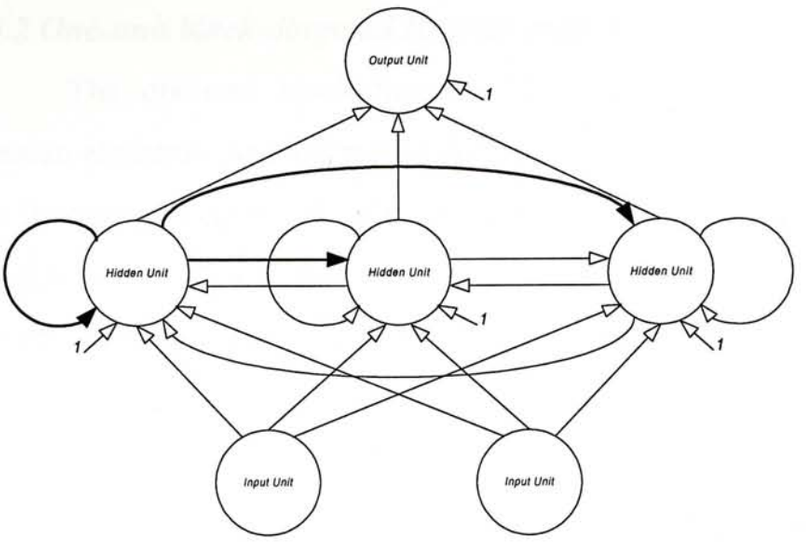


(b)

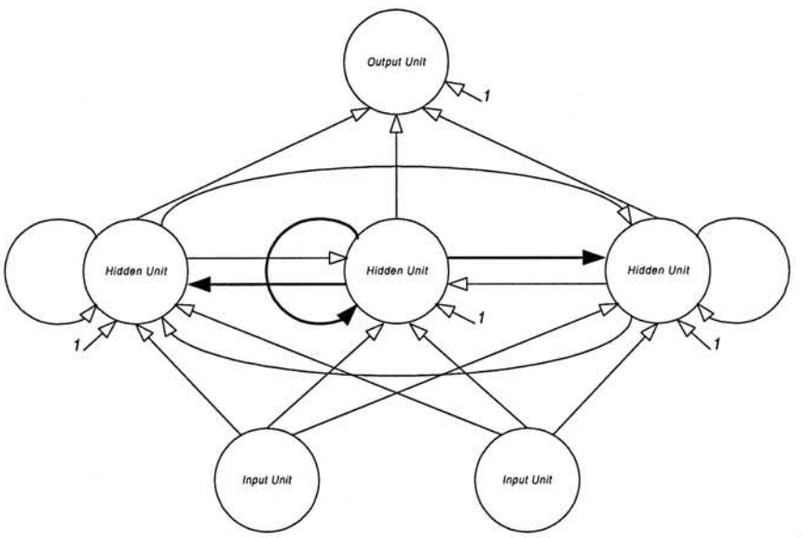


(c)

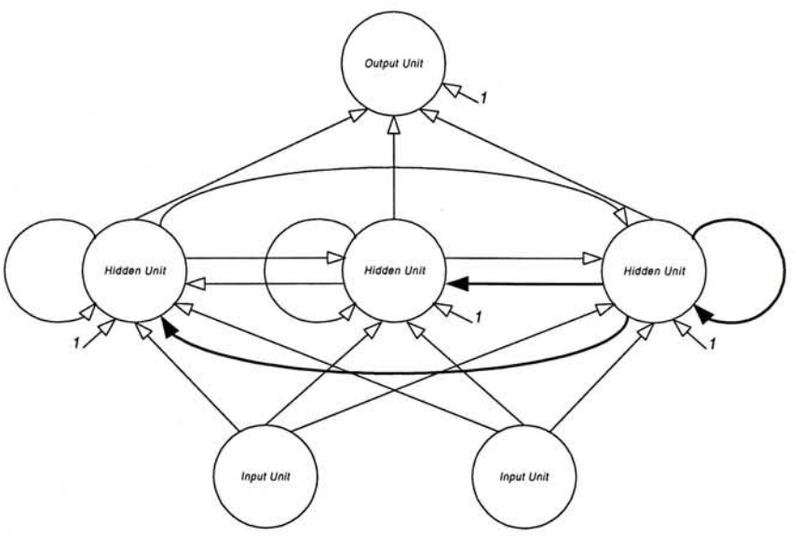
Figure 2-2 The weight arrangement of the correlation block-diagonal Hessian matrix. In each diagram, the bold weights are grouped in the weight vector of a block.



(a)



(b)



(c)

Figure 2-3 The weight arrangement of the correlation block-diagonal Hessian matrix. In each diagram, the bold weights are grouped in the weight vector of a block.

2.3.2 One-unit block-diagonal Hessian matrix

The one-unit block-diagonal Hessian matrix is formed by ignoring the Hessian elements (second partial derivatives of the cost function E) with respect to the incoming weights of different hidden units. In other words, the incoming weights of the same hidden unit are grouped in the weight vector of a block. This weight arrangement is called the *one-unit weight-grouping method.* This arrangement has been applied to the feedforward network training in [Kollias88] and [Kollias89].

Take a recurrent network with M input and N hidden units as an example. Each hidden unit is connected by M input units, one bias and N hidden units. The M+N+1 incoming weights of the same hidden unit are grouped in the weight vector of a block. There are N such weight vectors. This is illustrated in Figure 2-4 for the recurrent network with M = 2 and N = 3. In each diagram, the bold connections are the weights that are grouped in the weight vector of a block.

2.3.3 Sub-network block-diagonal Hessian matrix

The sub-network block-diagonal Hessian matrix is formed as follows. At first, the hidden units of a network are divided into several groups of hidden units and each group of hidden units is called a *sub-network.* The sub-network block-diagonal Hessian matrix is formed by ignoring the Hessian elements (second partial derivatives of the cost function E) with respect to the incoming weights of different sub-networks. In other words, the incoming weights of the same sub-network are grouped in the weight vector of a block. This weight arrangement is called the *sub-network weight-grouping method.*

Take a recurrent network with M input and N hidden units as an example. We assume that each sub-network consists of U hidden units. Then, the N hidden units are divided into N/U sub-networks (assuming that U is a factor of N). Each sub-network is connected by U×M input units, U biases and U×N hidden units. The U(M+N+1) incoming weights of the same sub-network are grouped in the weight vector of a block. There are N/U such weight vectors. This is illustrated in Figure 2-5 for the recurrent network with M = 2 and N = 3. U = 3 is chosen in this example. The bold connections are the weights that are grouped in the weight vector of a block.

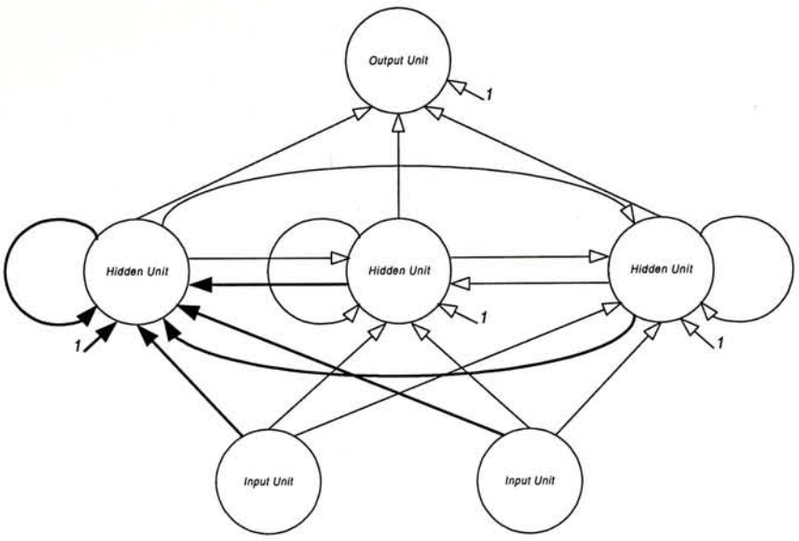
If U = 1, the sub-network block-diagonal matrix becomes the one-unit block-diagonal matrix described in Section 2.3.2. If U = x, we call this sub-network block-diagonal matrix the *x-unit block-diagonal Hessian matrix.* Its weight arrangement is called the *x-unit weight-grouping method.*

2.3.4 Layer block-diagonal Hessian matrix

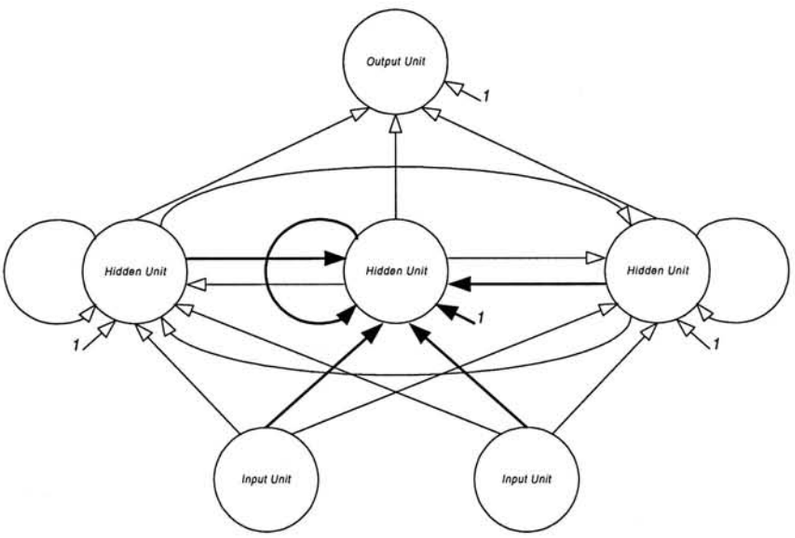
The layer block-diagonal Hessian matrix is formed by ignoring the Hessian elements (second partial derivatives of the cost function E) with respect to the weights of different layers. In other words, the weights of the same layer are grouped in the weight vector of a block. This weight arrangement is called the *layer weight-grouping method.* This arrangement was suggested for the feedforward networks in [Wille97].

Take a recurrent network with M input and N hidden units as an example. The (M+1)N input-to-hidden weights are grouped in the weight vector of a block and the N^2 recurrent weights are grouped in the weight vector of another block. This is illustrated in Figure 2-6 for the recurrent network with M = 2 and N = 3. In each diagram, the bold connections are the weights that are grouped in a block.

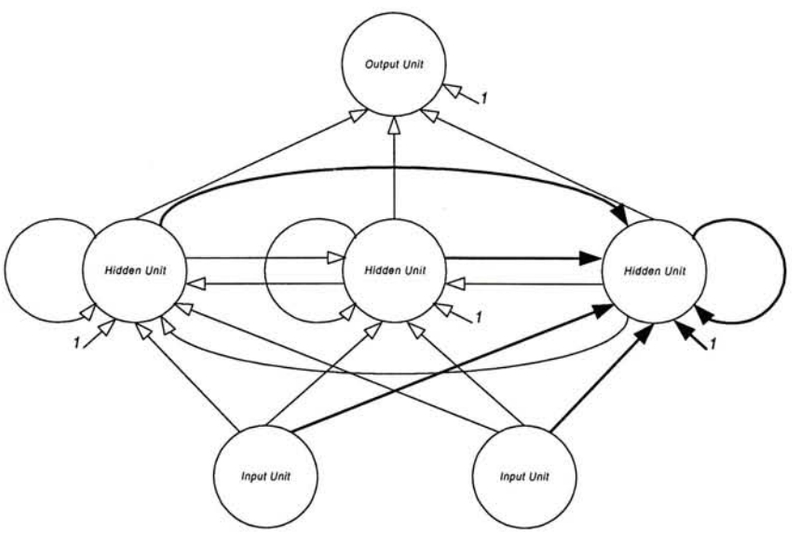
Table 2-1 summarizes the numbers and sizes of the blocks of the block-diagonal Hessian matrices described in Sections 2.3.1 to 2.3.4 and Table 2-2 summarizes the weight-grouping methods used in these block-diagonal Hessian matrices. Table 2-3 gives some examples for the recurrent networks with one input, six or nine hidden and one output units.



(a)



(b)



(c)

Figure 2-4 The weight arrangement of the one-unit block-diagonal Hessian matrix. In each diagram, the bold weights are grouped in the weight vector of a block.

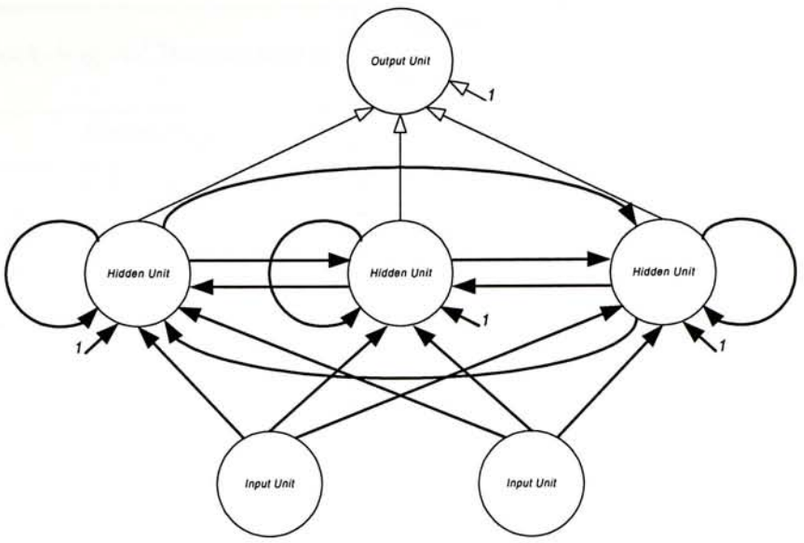
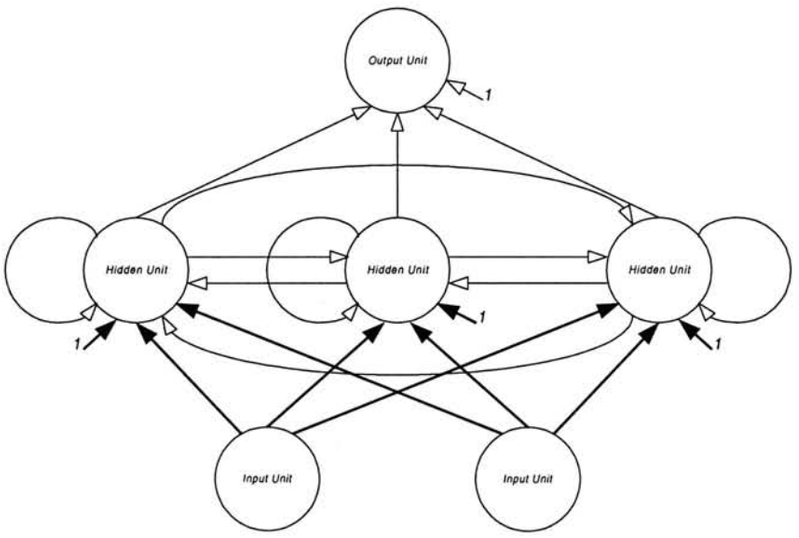
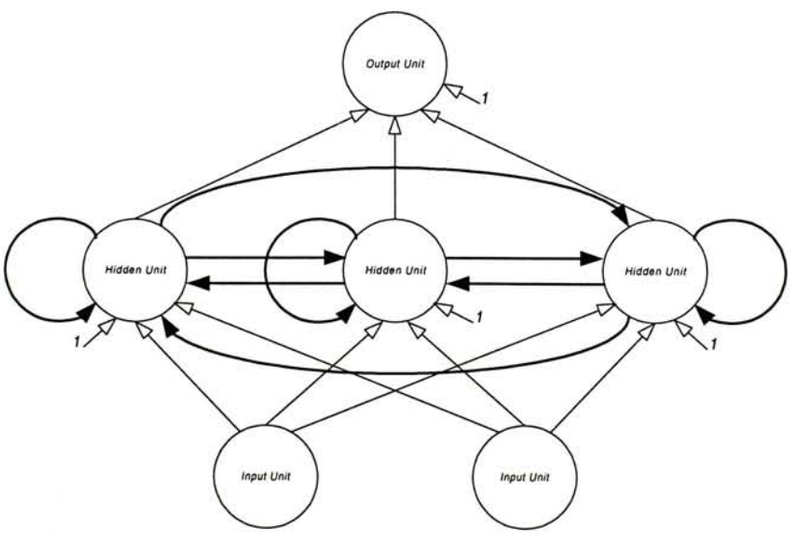


Figure 2-5 The weight arrangement of the three-unit block-diagonal Hessian matrix. The bold weights are grouped in the weight vector of a block



(a)



(b)

Figure 2-6 The weight arrangement of the layer block-diagonal Hessian matrix. In each diagram, the bold weights are grouped in the weight vector of a block

Block-diagonal Hessian matrix	No. of blocks, B	Block sizes, $\{q_1, q_2, \ldots, q_B\}$
Correlation	M+N+2	{N, N, …, N, P(N+1)}
Sub-network	N/U+1	{U(M+N+1), U(M+N+1), …, U(M+N+1), P(N+1)}
Sub-network (U=1) / One-unit	N+1	{M+N+1, M+N+1, …, M+N+1, P(N+1)}
Sub-network (U=3) / Three-unit	N/3+1	{3(M+N+1), 3(M+N+1), …, 3(M+N+1), P(N+1)}
Layer	3	$\{(M+1)N, N^2, P(N+1)\}$

Table 2-1 Numbers and sizes of blocks of the correlation, sub-network, one-unit, three-unit and layer block-diagonal Hessian matrices where M, N, P are the number of input, hidden and output units respectively.

Weight-grouping method	Which weights are grouped in the weight vector of a block
Correlation	output-layer weights; outgoing weights of the same hidden unit or the same input unit
Sub-network	output-layer weights; incoming weights of the same sub-network
Sub-network (U=1) / One-unit	output-layer weights; incoming weights of the same hidden unit
Sub-network (U=3) / Three-unit	output-layer weights; incoming weights of the same sub-network (U=3)
Layer	weights of the same layer

Table 2-2 Descriptions of the weight-grouping methods used in the correlation, sub-network, one-unit, three-unit and layer block-diagonal Hessian matrices.

Block-diagonal Hessian matrix	No. of Hidden units	No. of blocks	Block sizes
Correlation	6	9	{6,6,6,6,6,6,6,6,7}
	9	12	{9,9,9,9,9,9,9,9,9,9,9,10}
Sub-network (U=1) One-unit	6	7	{8,8,8,8,8,8,7}
	9	10	{11,11,11,11,11,11,11,11,11,10}
Sub-network (U=3) Three-unit	6	3	{24,24,7}
	9	4	{33,33,33,10}
Layer	6	3	{12,36,7}
	9	3	{18,81,10}

Table 2-3 Numbers and sizes of blocks of the correlation, one-unit, three-unit and layer block-diagonal Hessian matrices for the recurrent network with one input, six or nine hidden and one output units.

2.4 Updating methods

The updating equation of the Levenberg-Marquardt method in Equation 1.24 is written again as follows.

$$\Delta \mathbf{w} = -(\mathbf{J}^T\mathbf{J} + \lambda\, \mathbf{I})^{-1}\mathbf{J}^T\mathbf{e} \qquad (2.12)$$

If we apply the block-diagonal Hessian matrix in Equation 2.4 to Equation 2.12, Equation 2.12 becomes

$$\Delta \mathbf{w} = -(\mathbf{H}_{\text{block-diag.}} + \lambda\, \mathbf{I})^{-1}\mathbf{J}^T\mathbf{e} \qquad (2.13)$$

This system of updating equations can be decomposed into B smaller systems of updating equations.

$$\Delta \mathbf{w}_1 = -(\mathbf{J}_1^T\mathbf{J}_1 + \lambda_1\, \mathbf{I}_1)^{-1}\mathbf{J}_1^T\mathbf{e}$$

$$\Delta \mathbf{w}_2 = -(\mathbf{J}_2^T\mathbf{J}_2 + \lambda_2\, \mathbf{I}_2)^{-1}\mathbf{J}_2^T\mathbf{e}$$

$$\ldots$$

$$\Delta \mathbf{w}_B = -(\mathbf{J}_B^T\mathbf{J}_B + \lambda_B\, \mathbf{I}_B)^{-1}\mathbf{J}_B^T\mathbf{e} \qquad (2.14)$$

where $\Delta \mathbf{w}_i$ is a $q_i \times 1$ vector and $[\Delta \mathbf{w}_1^T\ \Delta \mathbf{w}_2^T\ \ldots\ \Delta \mathbf{w}_B^T]^T = \Delta \mathbf{w}$

$\mathbf{I}_i$ is a $q_i \times q_i$ unit matrix

λ_i is a learning parameter

B is the number of blocks

The calculation of the weight update vector $\Delta \mathbf{w}$ in Equations 2.14 requires less operations and storage than that in Equation 2.12. Moreover, $\Delta \mathbf{w}_1$, $\Delta \mathbf{w}_2$, ... $\Delta \mathbf{w}_B$ can be calculated separately in Equations 2.14 but they should be calculated at the same time in Equation 2.12. This property of Equation 2.14 leads to two different types of updating methods. They are the *asynchronous* and *synchronous updating methods*. The asynchronous method updates weights of one block $\mathbf{w}_i$ at a time while the synchronous method updates weights of all blocks $\mathbf{w}$ at a time. The descriptions of these two updating methods and their effects, together with the factors of block-diagonal Hessian matrices, on the training performance will be described in Chapter 4. Chapter 3 describes the data set and the setup of the experiments used in Chapter 4 and subsequent chapters.

Chapter 3 Data set and setup of experiments

3.1 Introduction

In this chapter, the problems that we chose to evaluate our proposed method are described. These problems consist of three time series prediction tasks and are described in Section 3.2. The details of this section include the source of data, data generation method, and partition of the training, validation and testing data.

The settings of our experiments are described in Section 3.3. They include the choices of recurrent neural network parameters and initialization methods. The method of dealing with over-fitting is described in Section 3.4.

3.2 Data set

Three time series are chosen as our data set. They are the *single sine, composite sine* and *sunspot data*, which are described in Sections 3.2.1 to 3.2.3 respectively. The first two series are synthetic data and the last one is real-life data. The data are used to do one-step ahead prediction. For example, when y(t) is presented to the network, the network gives us its estimate of y(t+1) where y(t), t = 0, 1, 2, ... is the time series.

3.2.1 Single sine

The first series shown in Equation 3.1 is a sinusoidal sequence of period $N_1 = 16$. We call it single sine.

$$y_{1a}(k) = \sin(\frac{2\pi k}{16}) \qquad (3.1)$$

where k = 0, 1, ..., 52

In real-life examples, the data usually contains some noise. To simulate this situation, we added some noise ε to Equation 3.1.

$$y_{1b}(k) = \sin(\frac{2\pi k}{16}) + \varepsilon \qquad (3.2)$$

where k = 0, 1, ..., 52

The noise ε is uniformly distributed over the interval (-α, α). Its probability density function Prob(ε) is

$$\text{Prob}(\varepsilon) = \begin{cases} \frac{1}{2\alpha} & \text{if } -\alpha < \varepsilon < \alpha \\ 0 & \text{otherwise} \end{cases} \qquad (3.3)$$

In our experiment, we set 24 on the ratio of the average power of the signal $y_{1a}(k)$ to the average power of the noise ε, which is known as the signal-to-noise ratio (SNR). The calculation of the interval (-α, α) of the noise ε is based on this ratio. It is as follows.

We first compute the average power of the signal $y_{1a}(k)$.

$$P_s = \frac{1}{N_1}\sum_{k=0}^{N_1-1} y_{1a}(k)^2 = 0.5$$

We then compute the average power of the noise ε.

$$P_n = E[\varepsilon^2] = \int_{-\infty}^{\infty} \varepsilon^2 \text{Prob}(\varepsilon) d\varepsilon = \frac{\alpha^2}{3}$$

Since $\text{SNR} = \frac{P_s}{P_n} = 24$, we obtain that

$$\frac{0.5}{\frac{1}{3}\alpha^2} = 24$$

$$\alpha = 0.25$$

We used Equation 3.2 to generate 52 data points (3.75 periods). The first 44 data points (2.75 periods) were used for training and the remaining 8 data points (0.5 period) were used for validation. Another 52 data points without added noise as shown in Equation 3.1 were used for testing.

3.2.2 Composite sine

The second series shown in Equation 3.4 is the sum of four sinusoidal sequences of different periods and amplitudes.

$$y_{2a}(k) = \frac{1}{\max(|y_2(k)|)} y_2(k) \qquad (3.4)$$

where $k = 0, 1, \ldots, 208$

$$y_2(k) = \sin(\frac{2\pi k}{64}) + \sin(\frac{2\pi k}{32}) + 4\cdot\sin(\frac{2\pi k}{\frac{64}{3}}) + 4\cdot\sin(\frac{2\pi k}{16})$$

$$\max(|y_2(k)|) = 9.0230$$

This series is re-scaled such that their values are within the range of [-1, 1]. Its period is $N_2 = 64$. We call this series composite sine.

Like what we did to the single sine data described in Section 3.2.1, we added some noise ε to the data as shown in Equation 3.5.

$$y_{2b}(k) = y_{2a}(k) + \varepsilon \qquad (3.5)$$

where $k = 0, 1, \ldots, 208$

The noise ε is uniformly distributed over the interval $(-\alpha, \alpha)$. Its probability density function Prob(ε) is shown in Equation 3.3. The interval $(-\alpha, \alpha)$ of the noise ε is determined by the pre-defined signal-to-noise ratio (SNR). In our case, SNR was set to 24 and the calculation of α was similar to that of the single sine data described in Section 3.2.1. The α calculated is 0.166.

We used Equation 3.5 to generate 208 data points (3.75 periods). The first 176 data points (2.75 periods) were used for training and the remaining 32 data points (0.5 period) were used for validation. Another 208 data points without added noise as shown in Equation 3.4 were used for testing.

3.2.3 Sunspot

The third series is a widely used real-life time series, yearly sunspot numbers. Sunspots are dark blotches on the sun. Normalized sunspot data from 1700 through 1920 was used for training and data from 1921 to 1955 was used for validation. Data from 1921 to 1955 was also used for testing to estimate the generalization performance.

Figure 3-1 shows the graphs of the above three time series. Table 3-1 summarizes the numbers of training data, validation data and testing data.

Data	No. of training data	No. of validation data	No. of testing data
Single sine	44	8	52
Composite sine	176	32	208
Sunspot	220	35	35

Table 3-1 Numbers of training, validation and testing data.

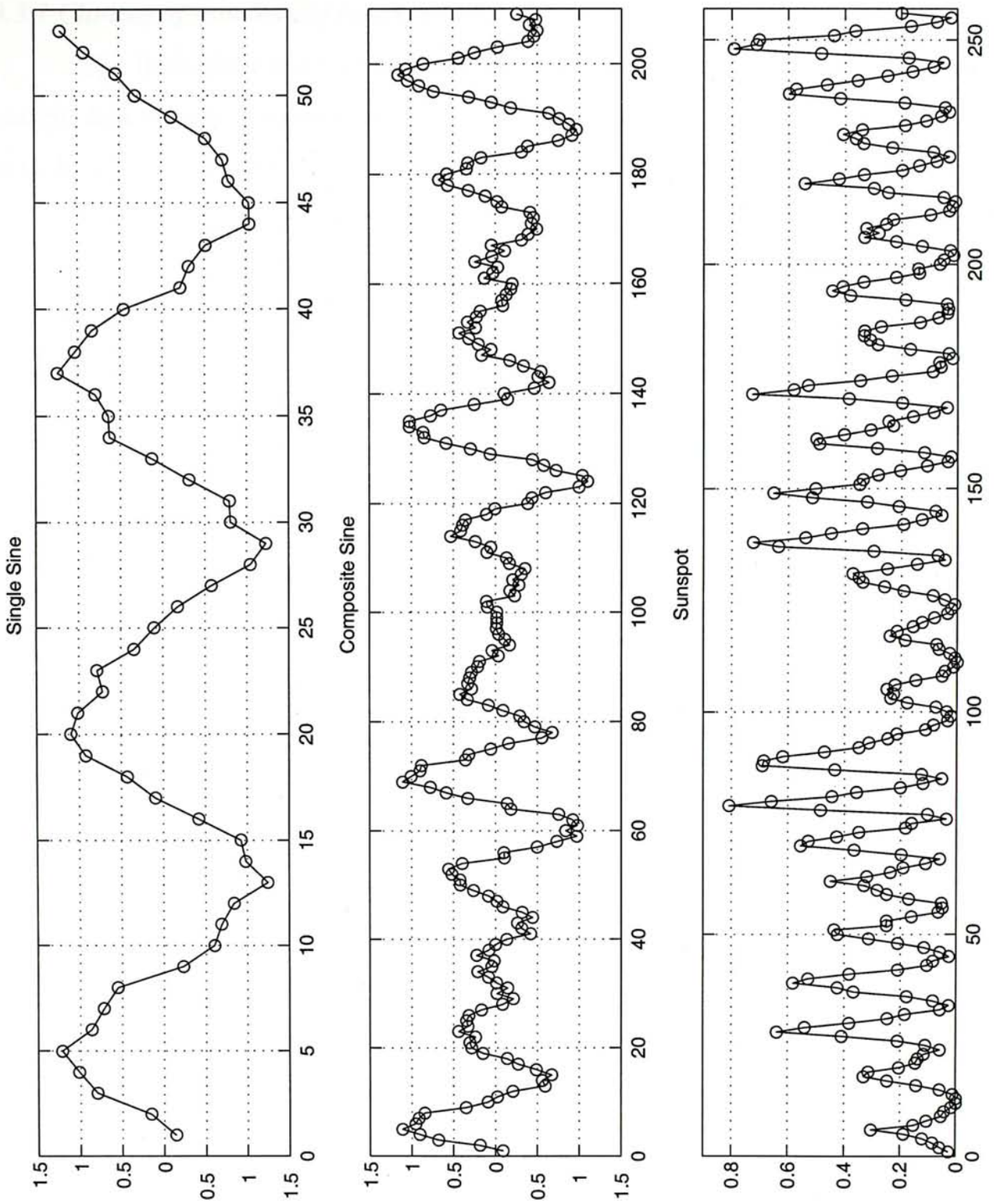


Figure 3-1 Graphs of data set

3.3 Choices of recurrent neural network parameters and initialization methods

Layered fully recurrent network described in Section 1.2.3 would be used in our experiments. The choices of its parameters, numbers of input, hidden and output units, will be described in Section 3.3.1. Moreover, we will describe how the hidden states and weights are initialized in Sections 3.3.2 and 3.3.3 respectively.

3.3.1 Choices of numbers of input, hidden and output units

The three time series chosen are univariate and at each time step, only the current data is input to the network. So, the number of input unit M is one. The task is to do next step prediction. So, the number of output unit P is also one.

Regarding the choice of the numbers of hidden units N, we used the Levenberg-Marquardt method described in Section 1.3.2.2 to train the networks with different numbers of hidden units (3, 6, and 9). We trained each network 20 times using different weight initializations. The one that gives the lowest average generalization error is chosen. Figures 3-2, 3-3 and 3-4 show the generalization errors of the networks trained to predict the single sine, composite sine and sunspot data respectively. These figures are the Box-whisker plots [Lawrence97]. The inter-quartile range (IQR) is shown with a box and the median is represented with a bar across the box. Whiskers extend from the ends of the box to the minimum and maximum values. Points greater than 1.5 IQR from the ends of the box is considered to be outliers and plotted separately with symbol '+'. In addition, the mean marked with 'x' and interpolated linearly by a solid line is superimposed on the plot. The minima of the generalization errors of the networks trained to predict the single and composite sine data are shown at the six hidden units. The minimum of the generalization error of the networks trained to predict the sunspot data is shown at the nine hidden units.

Table 3-2 shows the summary of number of hidden units, number of weights and data-to-weight ratio. We used the data-to-weight ratio to determine whether the network has too many free parameters.

3.3.2 Initial hidden states

The hidden states were initialized to 0. This means that no information is fed back to the network at the first time step.

3.3.3 Weight initialization method

The network weights were initialized to random numbers. They were uniformly distributed inside a small range of values [-0.1, 0.1]. Since a particular training run was sensitive to the initial conditions of the weights, we trained each network 20 times using different weight initializations.

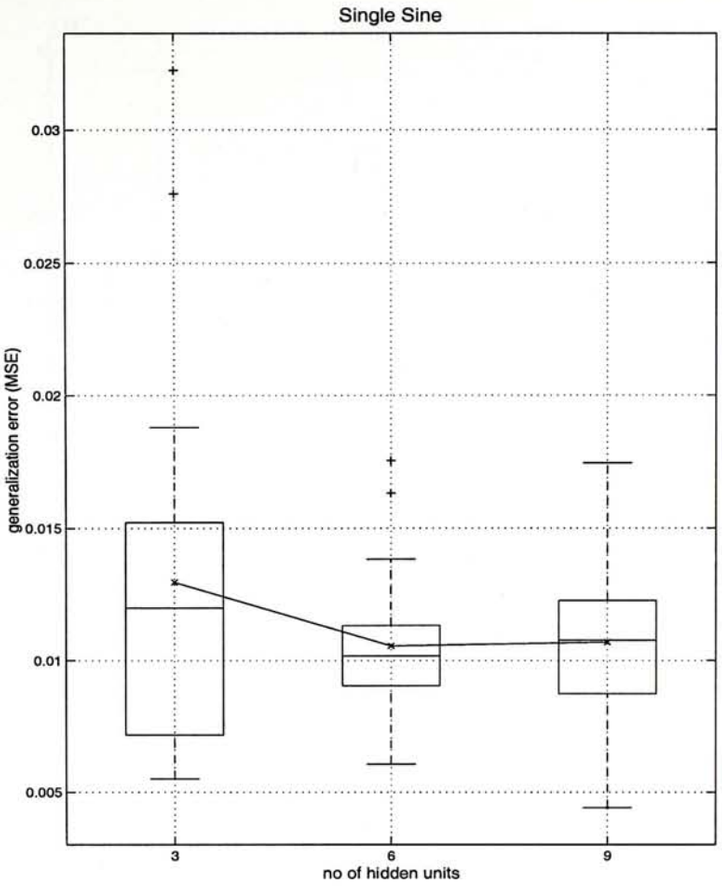


Figure 3-2 Generalization errors measured in terms of mean squared error against number of hidden units. The single sine training data was used in these experiments.

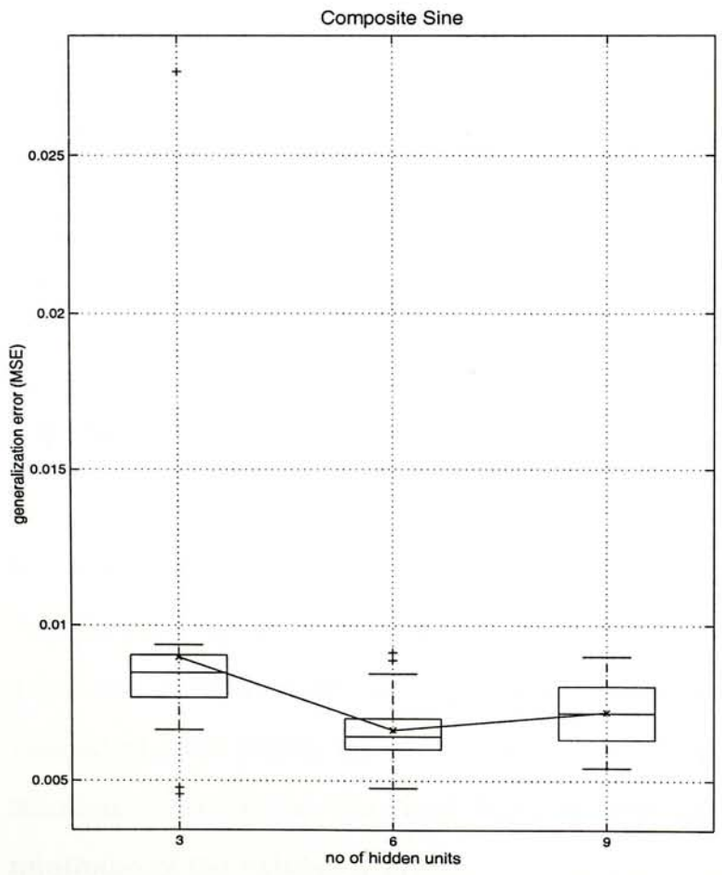


Figure 3-3 Generalization errors measured in terms of mean squared error against number of hidden units. The composite sine training data was used in these experiments.

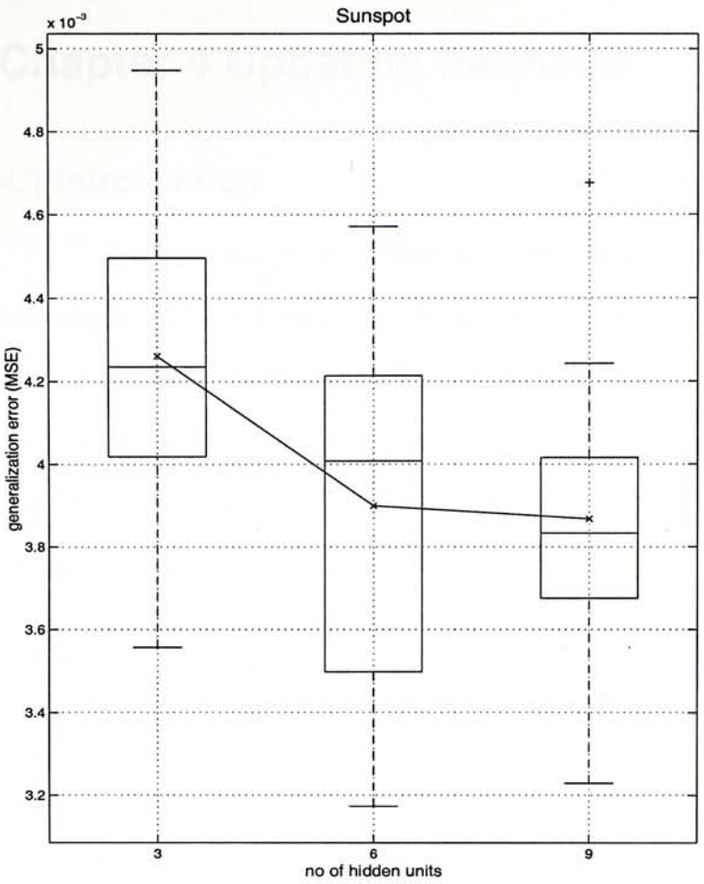


Figure 3-4 Generalization errors measured in terms of mean squared error against number of hidden units. The sunspot training data was used in these experiments.

Data	No. of hidden units	No. of weights	No. of training data : No. of weights
Single sine	6	55	1.25 : 1
Composite sine	6	55	3.2 : 1
Sunspot	9	109	2 : 1

Table 3-2 Number of hidden units, number of weights and data-to-weight ratio

3.4 Method of dealing with over-fitting

A network with too many free parameters will model the small details or noise of the time series and result in poor generalization. This phenomenon is called *over-fitting*. To avoid over-fitting, we used *early stopping* [Sarle95]. Validation error was computed on each iteration and the set of weights that minimized the validation error within the maximum amount of computation was chosen to be the solution. We define the *final training time* as the time required to reach the minimum of the validation error.

Chapter 4 Updating methods

4.1 Introduction

As discussed in Chapter 2, we need to consider two factors when the technique of block-diagonal Hessian approximation is applied. These two factors are

i. the choice of weight updating methods and
ii. the choice of block-diagonal Hessian matrices

In this chapter, we will study the first factor.

If the block-diagonal Hessian approximation is used, the original system of updating equations can be decomposed into B smaller systems of updating equations where B is the number of blocks. The weight updates of these B decomposed systems of updating equations can be calculated separately. So, the weights of different blocks can be updated asynchronously or synchronously. We call these updating methods asynchronous and synchronous updating methods respectively. Asynchronous method updates weights of one block at a time while synchronous method updates weights of all blocks at a time. In this chapter, implementation of these two updating methods is described and performance of these two updating methods is compared.

The subsequent sections are organized as follows. In Sections 4.2 and 4.3, we will evaluate the performance of the asynchronous updating method on the time series prediction problems described in Chapter 3. The block-diagonal Hessian matrices described in Chapter 2 would be used. Moreover, we would use the original Levenberg-Marquardt method with full Hessian matrix to provide a basis for comparison. Section 4.4 describes the corresponding result of using the synchronous updating method.

In Section 4.5, the training time performance of the asynchronous and synchronous updating methods is compared. The training time performance depends on two factors, which are the computation load per complete weight update and convergence speed. The effects of the updating methods on these two factors are also compared.

Finally, we compare our proposed methods with the traditional method: the gradient descent method with adaptive learning rate and momentum in Section 4.6.

4.2 Asynchronous updating method

The weight updating methods are divided into the asynchronous and synchronous updating methods. In this section, the asynchronous updating method, which updates weights of one block at a time, is studied. We will describe its implementation and evaluate its performance.

4.2.1 Algorithm

The implementation of the asynchronous updating method is shown in Algorithm 4.1.

Algorithm 4.1 Asynchronous updating method

1. $\lambda_1, \lambda_2, \ldots, \lambda_B = 0.001$, overflow_$\lambda$ = 0, β = 10, iteration = 0 and finished = false
2. i (index for weight vectors of different blocks) = 1
3. WHILE finished = false
4. calculate $\mathbf{J}_i^T\mathbf{e}$
5. IF $\mathbf{J}_i^T\mathbf{e} \geq$ minimum_gradient
6. IF $\lambda_i >$ maximum_λ
7. decrease λ_i by a factor β
8. overflow_$\lambda \leftarrow$ overflow_λ - 1
9. END
10. calculate $\Delta\mathbf{w}_i = -(\mathbf{J}_i^T\mathbf{J}_i + \lambda_i \mathbf{I}_i)^{-1}\mathbf{J}_i^T\mathbf{e}$
11. calculate $E(\mathbf{w}_i+\Delta\mathbf{w}_i)$
12. WHILE ($\lambda_i \leq$ maximum_λ) and ($E(\mathbf{w}_i+\Delta\mathbf{w}_i) \geq E(\mathbf{w}_i)$)
13. increase λ_i by a factor β
14. calculate $\Delta\mathbf{w}_i$
15. calculate $E(\mathbf{w}_i+\Delta\mathbf{w}_i)$
16. END
17. IF $\lambda_i \leq$ maximum_λ

18. update $\mathbf{w}_i$ ($\mathbf{w}_i \leftarrow \mathbf{w}_i + \Delta\mathbf{w}_i$)

19. decrease λ_i by a factor β

20. iteration $\leftarrow$ iteration + 1

21. ELSE

22. overflow_$\lambda \leftarrow$ overflow_λ + 1

23. END

24. END

25. update i (i $\leftarrow$ i + 1; IF i > B, i = 1, END)

26. calculate $\mathbf{J}^T\mathbf{e}$

27. IF (iteration > maximum_iteration) OR
(minimum of validation error is reached = true) OR
($\mathbf{J}^T\mathbf{e}$ < minimum_gradient) OR (overflow_λ = B)

28. finished = true

29. END

30. END

In this algorithm, only weights of one block at a time are updated (Step 18). The setting of λ_i is the same as the setting of λ of the original Levenberg-Marquardt method with full Hessian matrix described in Section 1.3.2.2.

4.2.2 Method of study

To have broader evaluation on the performance of the asynchronous updating method, we used the asynchronous updating method with different block-diagonal Hessian matrices. These block-diagonal Hessian matrices are the correlation, one-unit, three-unit and layer block-diagonal Hessian matrices described in Sections 2.3.1 to 2.3.4 respectively. To provide a basis for comparison, we used the original method with full Hessian matrix. To sum up, we also used the training methods with the following options.

i. asynchronous updating method and correlation block-diagonal Hessian matrix
ii. asynchronous updating method and one-unit block-diagonal Hessian matrix
iii. asynchronous updating method and three-unit block-diagonal Hessian matrix
iv. asynchronous updating method and layer block-diagonal Hessian matrix
v. original method and full Hessian matrix

These training methods were used to train the layered fully recurrent network described in Section 1.2.3 to predict the single sine, composite sine and sunspot data described in Sections 3.2.1 to 3.2.3 respectively.

We evaluated the performance of the asynchronous updating methods by comparing it with the performance of the original method with full Hessian matrix. The term performance refers to the time required to train the network (the final training time) and the generalization error of the trained network. The final training time and the generalization error are measured in terms of number of flops and mean squared error respectively.

4.2.3 Performance

The performance of different training methods is shown in Figure 4-1. Figures 4-1a and 4-1b show the final training time and the generalization errors respectively. The performance of the asynchronous updating methods used with the correlation, one-unit, three-unit and layer block-diagonal Hessian matrices is plotted in the first four columns respectively. The performance of the original method used with full Hessian matrix is plotted in the last column. This figure shows the performance of the training methods used to train the networks to predict the single sine data. The corresponding graphs of the performance of the training methods used to train the networks to predict the composite sine and sunspot data are shown in Figures 4-2 and 4-3 respectively.

We compare the performance of the asynchronous updating methods with that of the original method shown in Figures 4-1 to 4-3. The comparisons are made in terms of the ratios of their average performance. The ratios of their average final training time and average generalization errors are summarized in Tables 4-1 and 4-2 respectively.

Training data	Average final training time of the asynchronous updating method / Average final training time of the original method			
	correlation block-diagonal matrix	one-unit block-diagonal matrix	three-unit block-diagonal matrix	layer block-diagonal matrix
Single sine	0.98	0.87	0.78	0.79
Composite sine	1.86	1.65	1.38	1.27
Sunspot	1.28	0.23	0.25	0.32

Table 4-1 Ratios of the average final training time of the asynchronous updating method to that of the original method under different training data and block-diagonal Hessian matrices.

Training data	Average generalization error of the asynchronous updating method / Average generalization error of the original method			
	correlation block-diagonal matrix	one-unit block-diagonal matrix	three-unit block-diagonal matrix	layer block-diagonal matrix
Single sine	4.03	3.42	3.74	3.01
Composite sine	3.49	3.23	3.06	2.32
Sunspot	1.21	1.41	1.68	1.14

Table 4-2 Ratios of the average generalization error of the asynchronous updating method to that of the original method under different training data and block-diagonal Hessian matrices.

Table 4-1 shows that the training time performance of the asynchronous updating methods relative to that of the original method varied with the training data. It was good (shorter final training time), average or poor when the networks were trained to predict the sunspot, single sine or composite sine data respectively.

Table 4-2 shows that the generalization performance of the asynchronous updating methods was slightly poor when the networks were trained to predict the sunspot data but was very poor when the networks were trained to predict the single sine or composite sine data. The generalization errors were several times larger than that of the original method. This problem is investigated below.

4.2.4 Investigation on poor generalization

4.2.4.1 Hidden states

Hidden states show the internal operations of the network. Our first examination is the hidden states of the network in the final weight configuration. The hidden states of the networks trained to predict the single sine, composite sine

and sunspot data are shown in Figures 4-4, 4-5 and 4-6 respectively. The asynchronous updating method and the one-unit block-diagonal Hessian matrix were used to train these networks. In these figures, different rows represent different trials. There are 20 rows representing 20 different trials. Different columns represent different hidden units. We used networks with six hidden units to predict the single and composite sine data. So, there are six columns representing six different hidden units in Figures 4-4 and 4-5. We used networks with nine hidden units to predict the sunspot data. So, there are nine columns representing nine different hidden units in Figure 4-6. The hidden states in each subplot are plotted against time. Moreover, we draw a vertical line to separate the hidden states of the network operated on the training set and the validation set.

Figures 4-4 and 4-5 show that the states of most hidden units of the networks trained to predict the single and composite sine data were in the saturation region, where the hidden state levels off and approaches fixed limits, in most trials and time. Average absolute hidden state defined in Equation 4.1 was used to measure the degree of saturation.

$$\text{Average absolute hidden state} = \frac{1}{20NT}\sum_{tr=1}^{20}\sum_{j=1}^{N}\sum_{t=1}^{T}\left|V_j^{tr}(t)\right| \qquad (4.1)$$

where $V_j^{tr}(t)$ is the state of the jth hidden unit at time t in the trial tr,

t, j and tr are the indexes of the time step, hidden unit and trial,

N is the number of hidden units and

T is the number of training and validation data

The average absolute hidden states shown in Figures 4-4 and 4-5 are 0.79 and 0.73 respectively. They are large when compare with the average absolute hidden states of the networks trained by the original method, which are 0.42 and 0.36. The observations about the hidden states were the same when the other block-diagonal Hessian matrices (correlation, three-unit and layer block-diagonal Hessian matrices) were used instead of the one-unit block-diagonal Hessian matrix. The average absolute hidden states of using the other block-diagonal Hessian matrices are summarized in Table 4-3.

Training data	Average absolute hidden state				
	correlation block-diagonal Hessian matrix	one-unit block-diagonal Hessian matrix	three-unit block-diagonal Hessian matrix	layer block-diagonal Hessian matrix	full Hessian matrix
Single sine	0.91	0.79	0.88	0.86	0.42
Composite sine	0.84	0.73	0.80	0.74	0.36
Sunspot	0.73	0.57	0.68	0.60	0.38

Table 4-3 Average absolute hidden states of the asynchronous updating method under different training data and block-diagonal Hessian matrices.

Similarly, Figure 4-6 shows that the states of about half of the hidden units of the networks trained to predict the sunspot data were in the saturation region in most trials and time. The average absolute hidden state is 0.57. Again, it is large when compare with the average absolute hidden state of the networks trained by the original method, which is 0.38. The observations about the hidden states were the same when the other block-diagonal Hessian matrices were used instead of the one-unit block-diagonal Hessian matrix.

We observed that the poor generalization performance was related to the frequently saturated hidden states. Most hidden units of the networks trained to predict the single and composite sine data were frequently saturated and the generalization performance was very poor. About half of the hidden units of the networks trained to predict the sunspot data were frequently saturated. The problem of frequently saturated hidden states was less serious and the generalization performance was less poor.

The frequently saturated hidden states may be due to the large incoming weight magnitudes of the hidden units. So, the next examination is the incoming weight magnitudes of the hidden units in the final weight configuration.

4.2.4.2 Incoming weight magnitudes of the hidden units

The incoming weight magnitudes of the hidden units were measured in two ways.

i. One measurement is the maximum and minimum of the incoming weights
ii. The other is the 90th and 10th percentiles of the incoming weights

The maximum and minimum are based entirely on the extreme values. The 90th and 10th percentiles depend on the middle 80% of the incoming weights and are not affected by the extreme values. The difference between the 90th and 10th percentiles is used to measure the typical range of the incoming weights.

Figure 4-7 shows the magnitudes of the above measures. The placements of the training methods on the x-axes are the same as those shown in Figure 4-1. Figure 4-7a shows the magnitudes of maximum and minimum of the incoming weights of the hidden units, which are represented by symbols 'x' and 'o' respectively. Figure 4-7b shows the magnitudes of the 90th and 10th percentiles of the incoming weights of the hidden units, which are represented by symbols 'x' and 'o' respectively. These incoming weights are the ensemble of the incoming weights of 20 networks of different trials. These networks were trained to predict the single sine data. The corresponding graphs of the weight magnitudes of the networks trained to predict the composite sine and sunspot data are shown in Figures 4-8 and 4-9 respectively.

These figures show that the magnitudes of the maximum and minimum of the incoming weights of the networks that we used the asynchronous updating methods to train were from 3.5 to 40000 times larger than those of the networks that we used the original method to train. The magnitudes of the 90th and 10th percentiles of the incoming weights of the networks that we used the asynchronous updating methods to train were from 1.25 to 20 times larger than those of the networks that we used the original method to train. These indicated that at least 20% of the incoming weight magnitudes of the hidden units were from large to extremely large. The frequently saturated hidden states were probably due to these large incoming weight magnitudes of the hidden units.

The large incoming weight magnitudes of the hidden units can be explained as follows. Initially, the weights were very small and so did the network output. And we updated a small part of weights at a time to minimize the difference between the initial network output and the target value. That part of weights might become very large in order to produce significant network output when the other weights were still small.

4.2.4.3 Weight change against time

We investigated the problem further by observing the weight changes against time. The result showed that the large weight magnitudes were due to the large weight changes, which occur in the beginning of the learning. To solve this problem, we imposed weight change constraint on each weight. We call this the *asynchronous updating with constraint method*, which will further be described in Section 4.3.

Label	Descriptions		
	Hessian matrix	updating method	maximum allowed weight change
full	full Hessian matrix	N/A	N/A
c/a	correlation block-diagonal Hessian matrix	asynchronous	∞
c/a(1)		asynchronous with constraint	1
c/a(0.5)			0.5
c/a(0.1)			0.1
u1/a	one-unit block-diagonal Hessian matrix	asynchronous	∞
u1/a(1)		asynchronous with constraint	1
u1/a(0.5)			0.5
u1/a(0.1)			0.1
u3/a	three-unit block-diagonal Hessian matrix	asynchronous	∞
u3/a(1)		asynchronous with constraint	1
u3/a(0.5)			0.5
u3/a(0.1)			0.1
l/a	layer block-diagonal Hessian matrix	asynchronous	∞
l/a(1)		asynchronous with constraint	1
l/a(0.5)			0.5
l/a(0.1)			0.1

Table 4-4 Labels for the training methods using different Hessian matrices, asynchronous updating methods and values of maximum allowed weight change.

Label	Descriptions	
	Hessian matrix	updating method
full	full Hessian matrix	N/A
c/s	correlation block-diagonal Hessian matrix	multiple λ's with line search synchronous
c/s(l)		single λ synchronous
c/s(ls)		multiple λ's synchronous
u1/s	one-unit block-diagonal Hessian matrix	multiple λ's with line search synchronous
u1/s(l)		single λ synchronous
u1/s(ls)		multiple λ's synchronous
u3/s	three-unit block-diagonal Hessian matrix	multiple λ's with line search synchronous
u3/s(l)		single λ synchronous
u3/s(ls)		multiple λ's synchronous
l/s	layer block-diagonal Hessian matrix	multiple λ's with line search synchronous
l/s(l)		single λ synchronous
l/s(ls)		multiple λ's synchronous

Table 4-5 Labels for the training methods using different Hessian matrices and implementation of synchronous updating methods

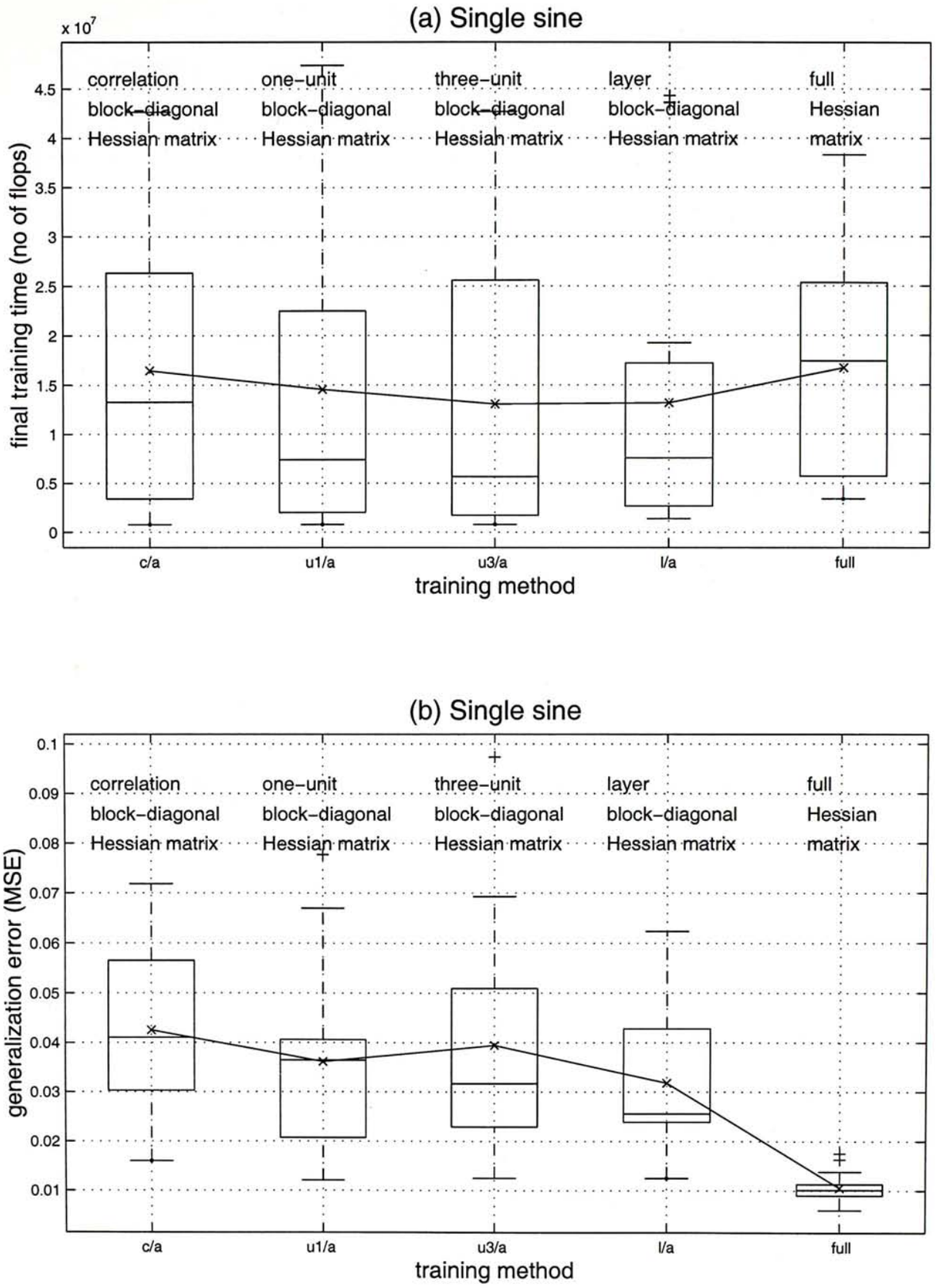


Figure 4-1 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using the asynchronous updating method and different block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 4-4. The single sine training data was used in these experiments.

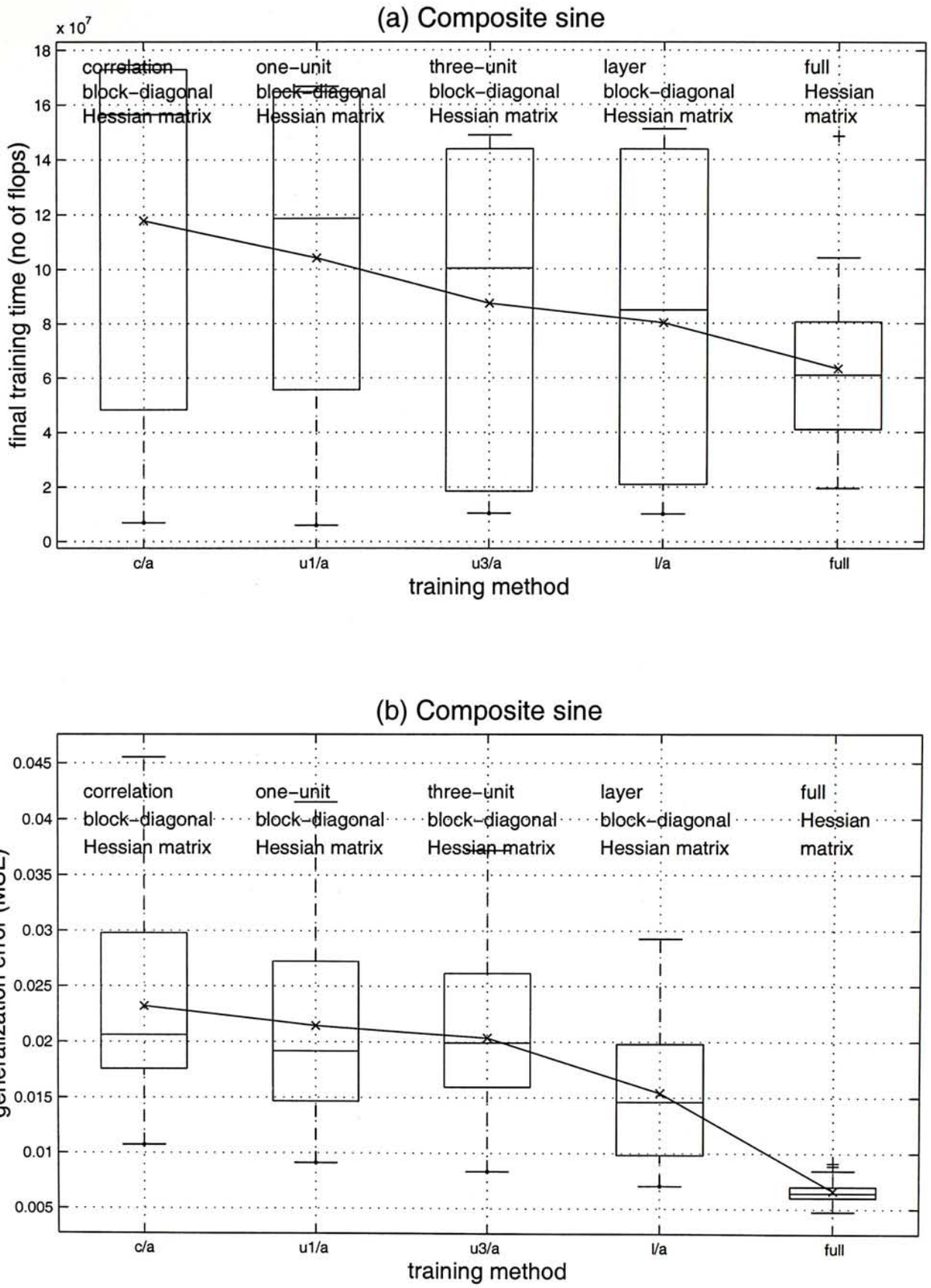


Figure 4-2 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using the asynchronous updating method and different block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 4-4. The composite sine training data was used in these experiments.

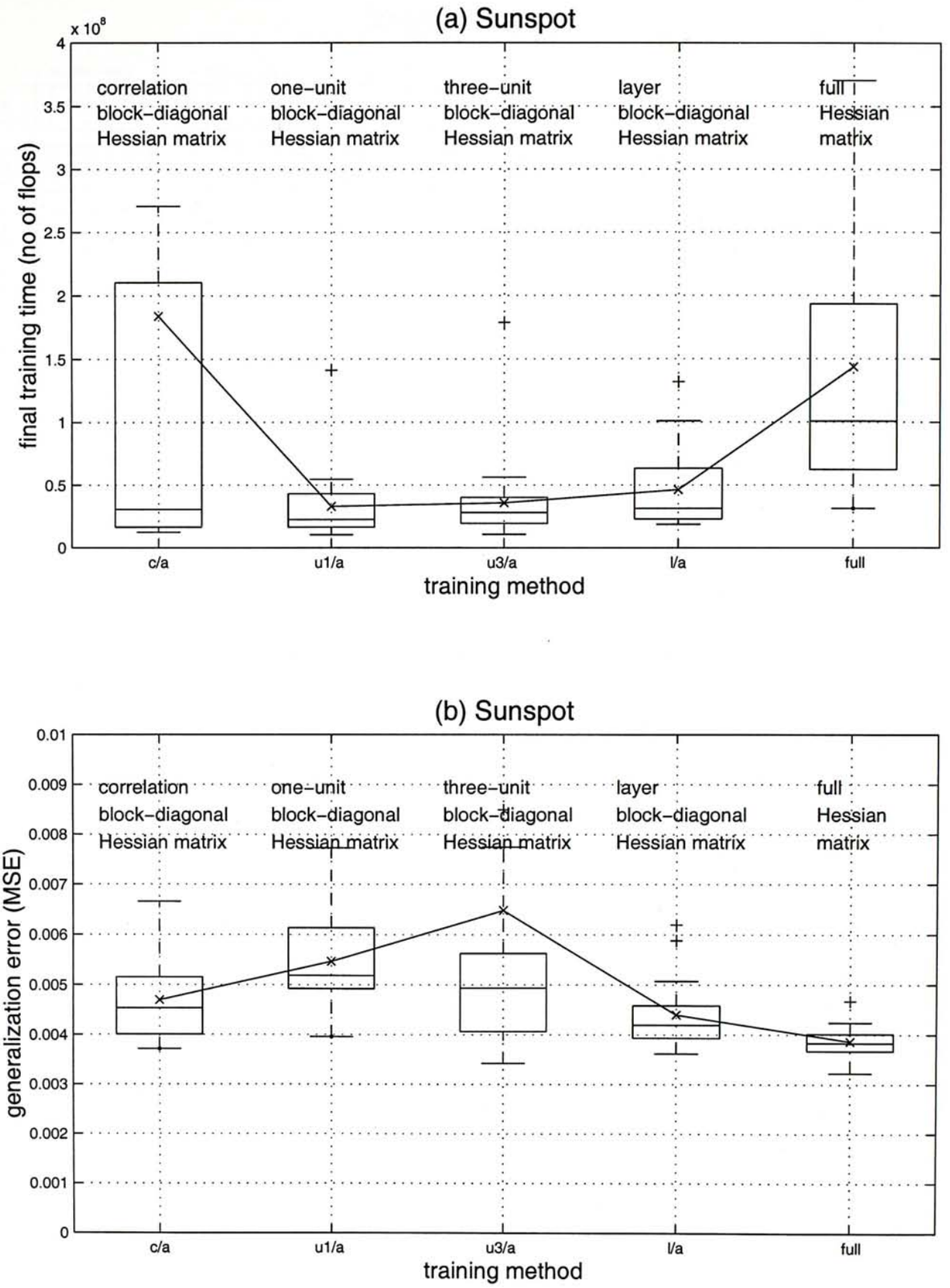


Figure 4-3 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using the asynchronous updating method and different block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 4-4. The sunspot training data was used in these experiments.

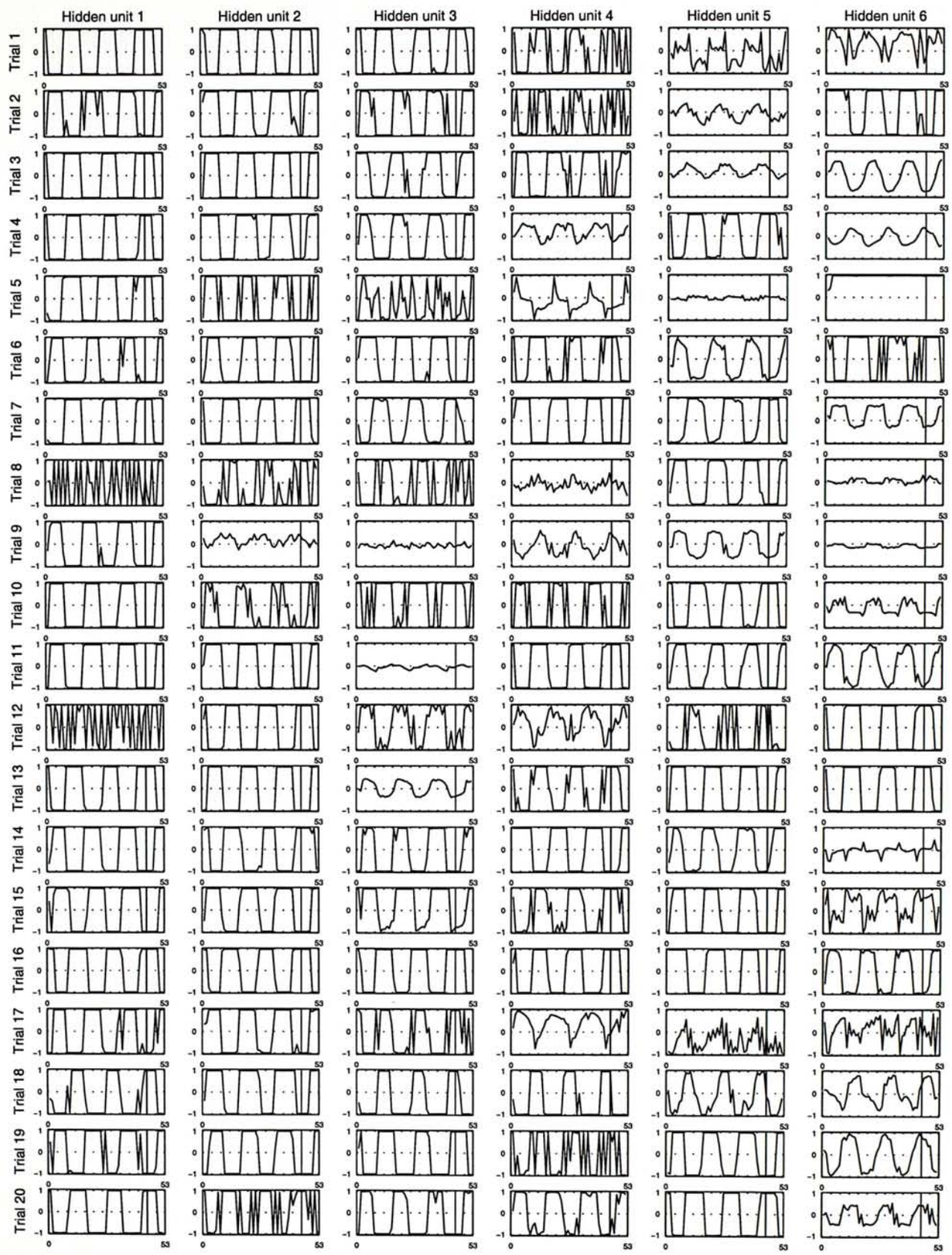


Figure 4-4 Hidden states of 20 networks of different trials. The asynchronous updating method and the one-unit block-diagonal Hessian matrix were used to train these networks. The single sine training data was used in these experiments.

Figure 4-5 Hidden states of 20 networks of different trials. The asynchronous updating method and the one-unit block-diagonal Hessian matrix were used to train these networks. The composite sine training data was used in these experiments.

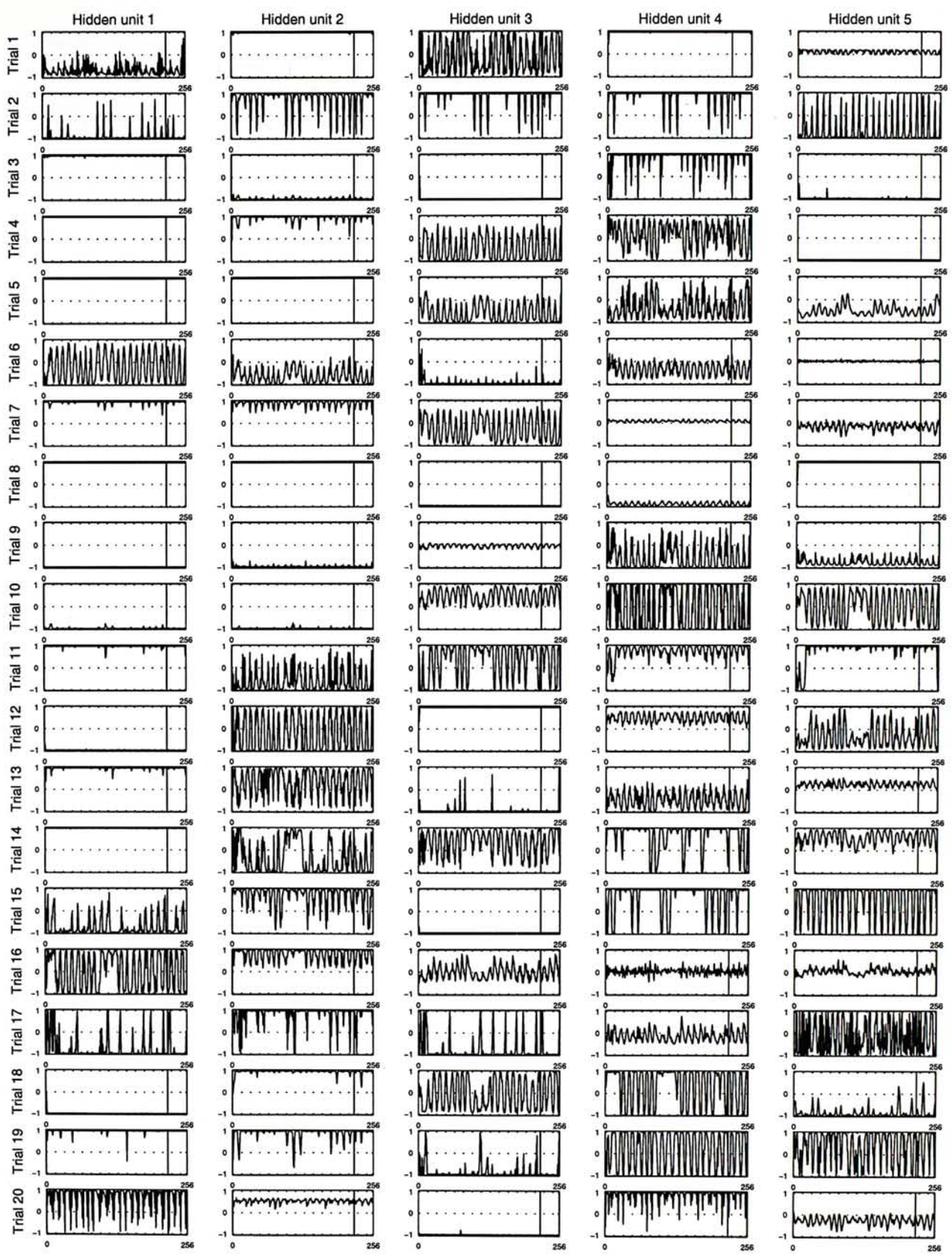


Figure 4-6 Hidden states of 20 networks of different trials. The asynchronous updating method and the one-unit block-diagonal Hessian matrix were used to train these networks. The sunspot training data was used in these experiments.

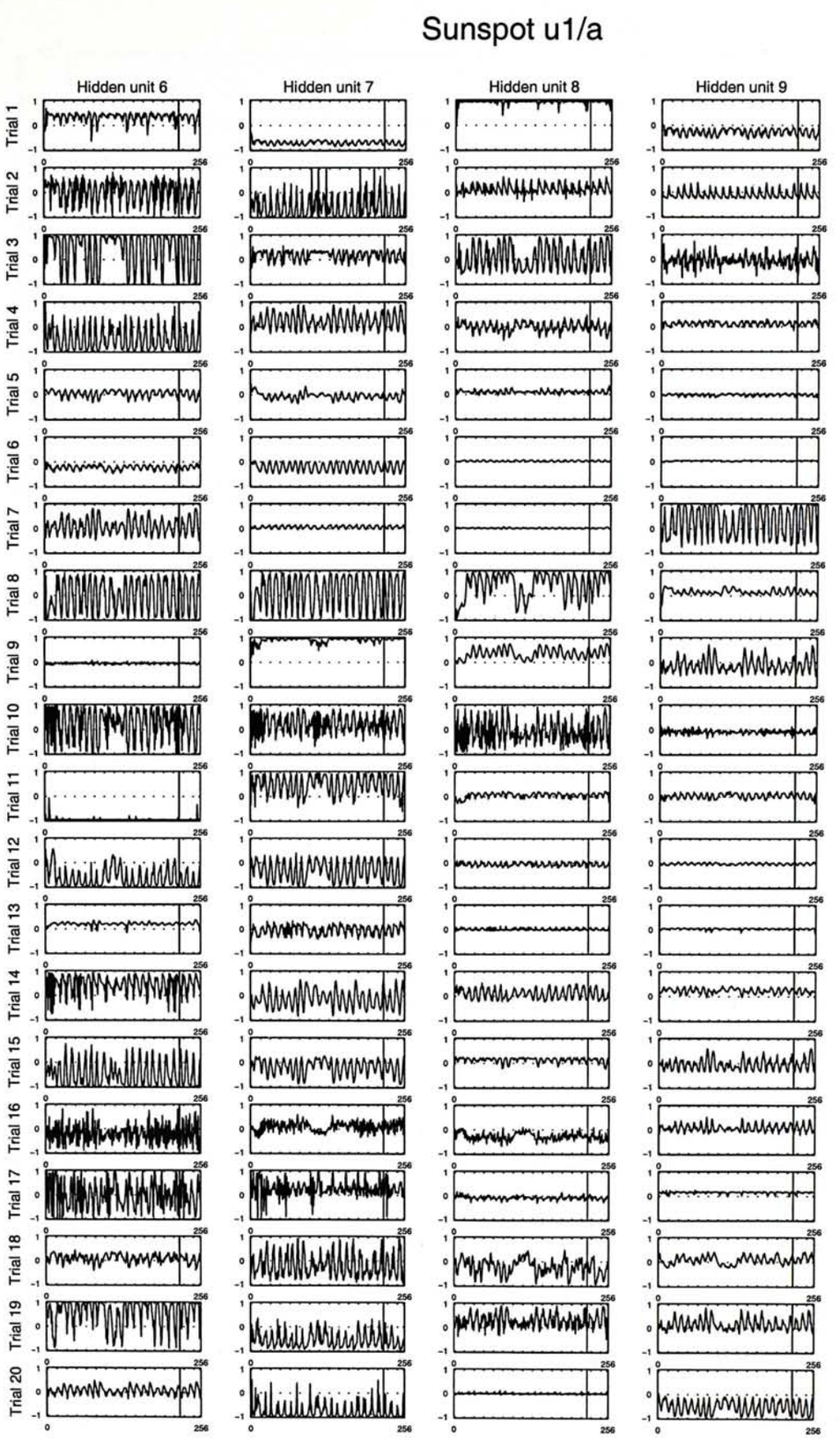
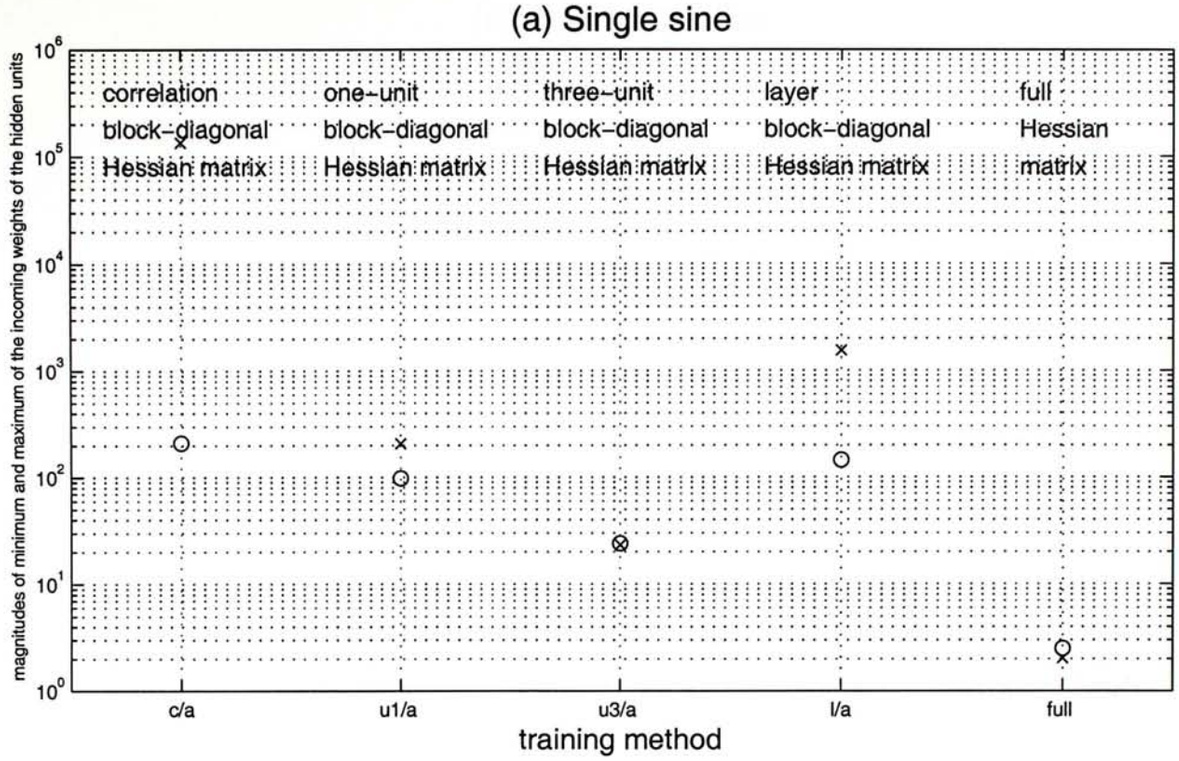


Figure 4-6 Hidden states of 20 networks of different trials. The asynchronous updating method and the one-unit block-diagonal Hessian matrix were used to train these networks. The sunspot training data was used in these experiments. (continued)



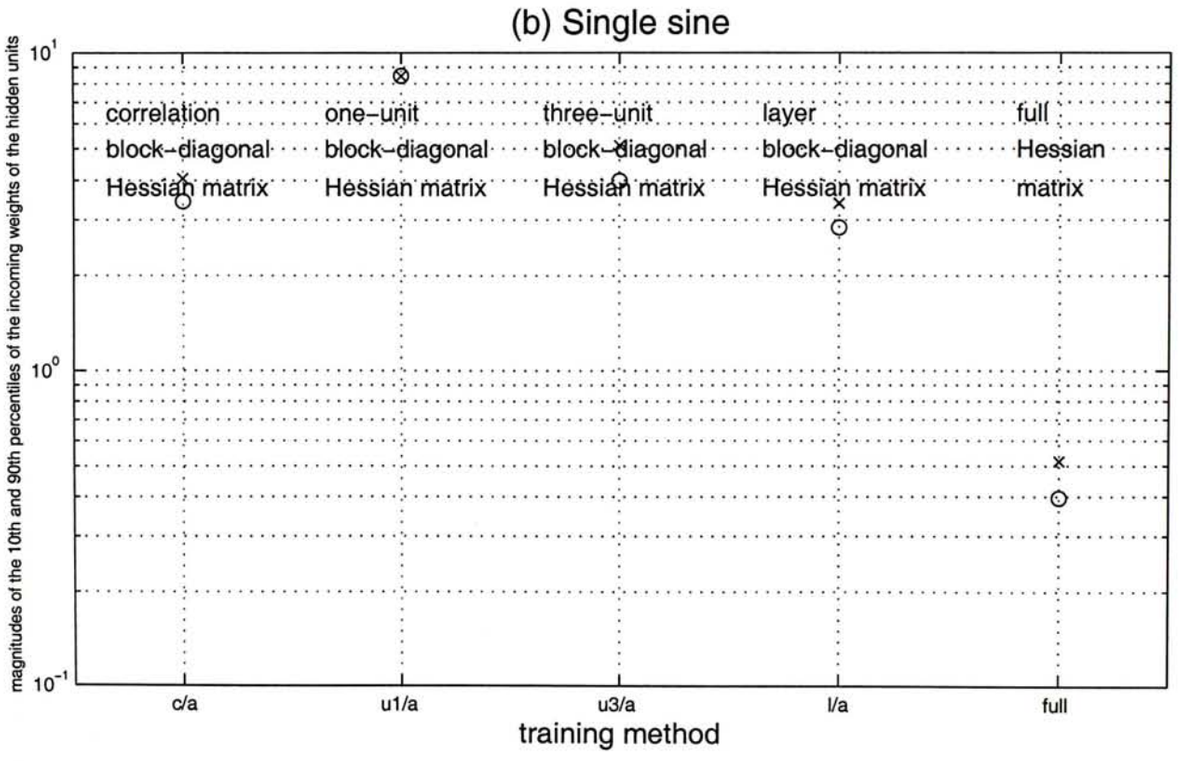
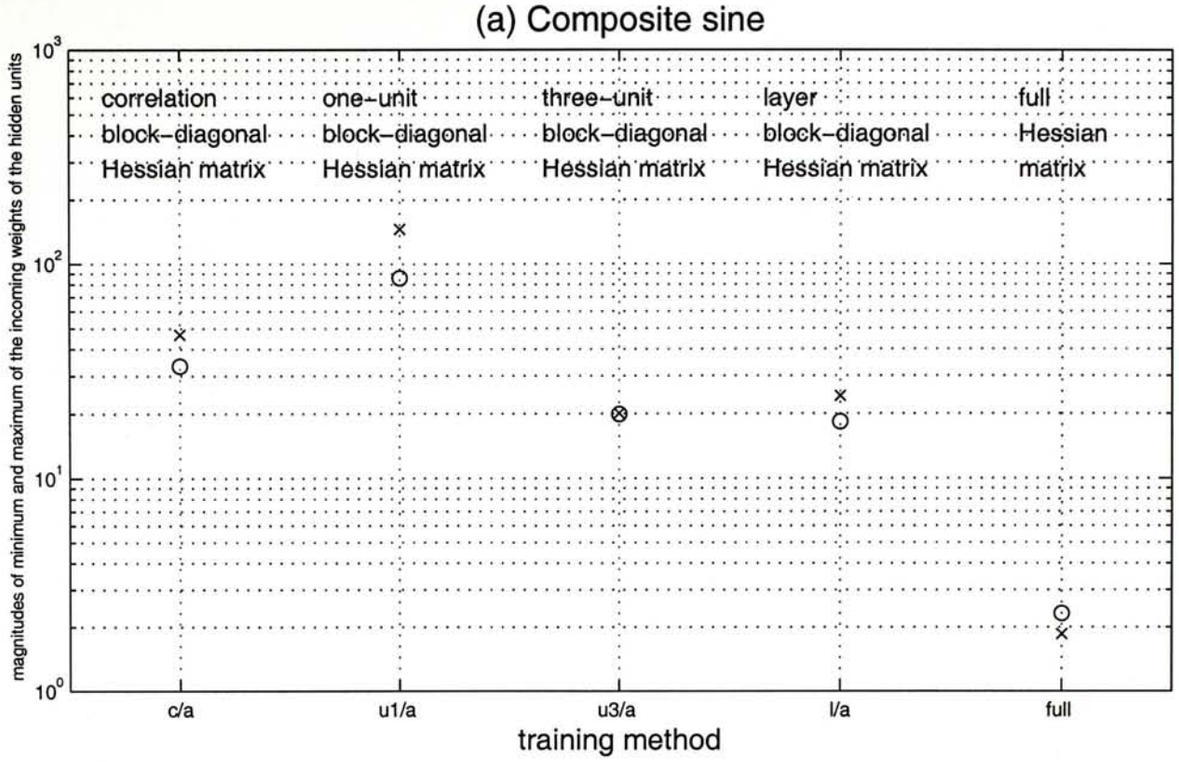


Figure 4-7 Magnitudes of (a) minimum 'o' and maximum 'x' and (b) the 10th percentile 'o' and 90th percentile 'x' of the incoming weights of the hidden units. The x-axes show the training methods using the asynchronous updating method and different block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 4-4. The single sine training data was used in these experiments.



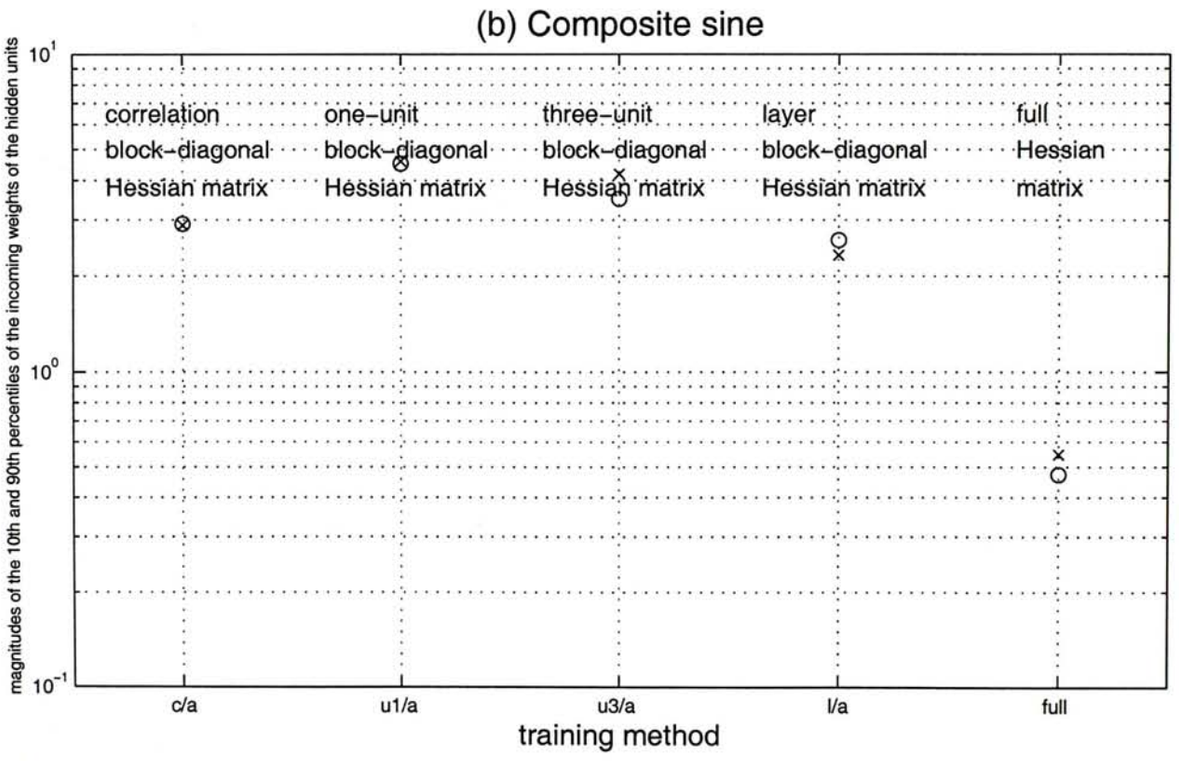
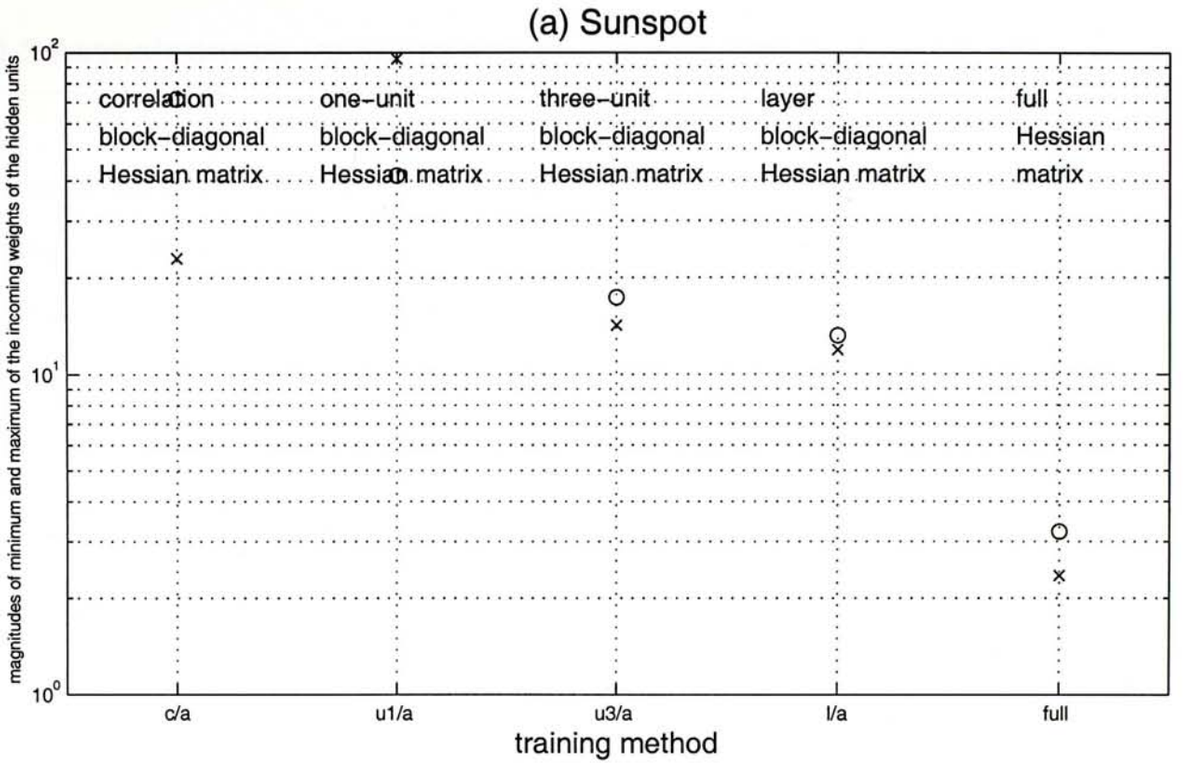


Figure 4-8 Magnitudes of (a) minimum 'o' and maximum 'x' and (b) the 10th percentile 'o' and 90th percentile 'x' of the incoming weights of the hidden units. The x-axes show the training methods using the asynchronous updating method and different block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 4-4. The composite sine training data was used in these experiments.



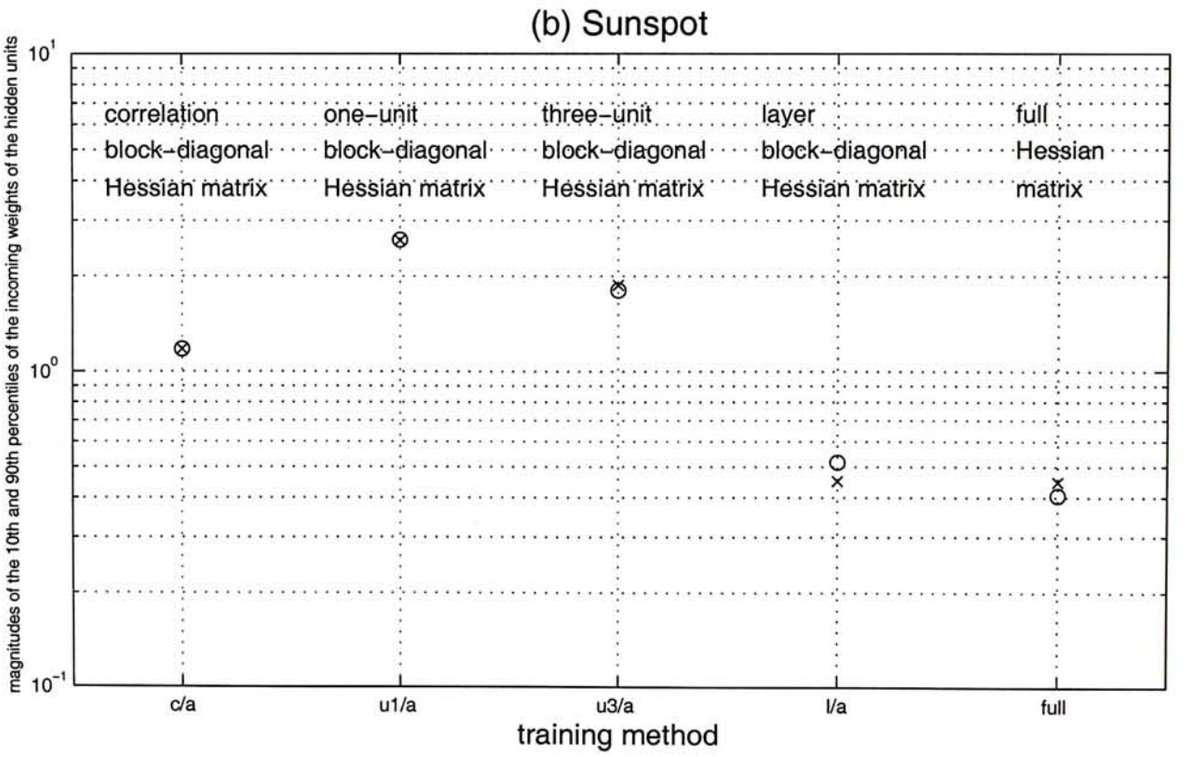


Figure 4-9 Magnitudes of (a) minimum 'o' and maximum 'x' and (b) the 10th percentile 'o' and 90th percentile 'x' of the incoming weights of the hidden units. The x-axes show the training methods using the asynchronous updating method and different block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 4-4. The sunspot training data was used in these experiments.

4.3 Asynchronous updating with constraint method

The implementation of the asynchronous updating method proposed in Section 4.2 has the problem of poor generalization generated by the large incoming weight magnitudes of the hidden units. In this section, another implementation of the asynchronous updating method which aims to remedy this problem is proposed. The evaluation of this implementation is the same as that described in Section 4.2.

4.3.1 Algorithm

Asynchronous updating with constraint method is derived from the asynchronous updating method described in Section 4.2.1. It adds some procedures to the asynchronous updating method to ensure that large weight changes do not happen. Its algorithm is as follows.

Algorithm 4.2 Asynchronous updating with constraint method

1. $\lambda_1, \lambda_2, \ldots, \lambda_B = 0.001$, overflow_$\lambda = 0$, $\beta = 10$, iteration = 0 and finished = false
2. set the value of maximum allowed weight change, maxwtchg
3. i (index for weight vectors of different blocks) = 1
4. WHILE finished = false
5. calculate $\mathbf{J}_i^T\mathbf{e}$
6. IF $\mathbf{J}_i^T\mathbf{e} \geq$ minimum_gradient
7. IF $\lambda_i >$ maximum_λ
8. decrease λ_i by a factor β
9. overflow_$\lambda \leftarrow$ overflow_λ - 1
10. END
11. calculate $\Delta\mathbf{w}_i = -(\mathbf{J}_i^T\mathbf{J}_i + \lambda_i \mathbf{I}_i)^{-1}\mathbf{J}_i^T\mathbf{e}$
12. calculate $E(\mathbf{w}_i+\Delta\mathbf{w}_i)$
13. WHILE ($\lambda_i \leq$ maximum_λ) and ($E(\mathbf{w}_i+\Delta\mathbf{w}_i) \geq E(\mathbf{w}_i)$)
14. increase λ_i by a factor β
15. calculate $\Delta\mathbf{w}_i$
16. calculate $E(\mathbf{w}_i+\Delta\mathbf{w}_i)$
17. END

18. IF $\lambda_i \leq$ maximum_λ

19. IF $\| \Delta \mathbf{w}_i \|_\infty^\dagger >$ maxwtchg

20. update $\mathbf{w}_i$ ($\mathbf{w}_i \leftarrow \mathbf{w}_i + \frac{\text{maxwtchg}}{\| \Delta \mathbf{w}_i \|_\infty} \Delta \mathbf{w}_i$)

21. ELSE

22. update $\mathbf{w}_i$ ($\mathbf{w}_i \leftarrow \mathbf{w}_i + \Delta \mathbf{w}_i$)

23. END

24. decrease λ_i by a factor β

25. iteration $\leftarrow$ iteration + 1

26. ELSE

27. overflow_$\lambda \leftarrow$ overflow_λ + 1

28. END

29. END

30. update i (i $\leftarrow$ i + 1; IF i > B, i = 1, END)

31. calculate $\mathbf{J}^T\mathbf{e}$

32. IF (iteration > maximum_iteration) OR
(minimum of validation error is reached = true) OR
($\mathbf{J}^T\mathbf{e}$ < minimum_gradient) OR (overflow_λ = B)

33. finished = true

34. END

35. END

† l_∞-norm $\| \mathbf{x} \|_\infty$ is defined as $\max_j | x_j |$.

The added procedures include the setting of the maximum allowed weight change parameter (Step 2), checking of the weight change magnitude and conditional constraint application (Steps 19 to 23).

4.3.2 Method of study

We evaluated the performance of the asynchronous updating with constraint method by comparing it with the performance of the asynchronous updating without constraint method described in Section 4.2 and the original method with full Hessian matrix. The asynchronous updating with constraint method would be used with the correlation, one-unit, three-unit and layer block-diagonal Hessian matrices described

in Sections 2.3.1 to 2.3.4 respectively. These training methods were used to train the layered fully recurrent network described in Section 1.2.3 to predict the single sine, composite sine and sunspot data described in Sections 3.2.1 to 3.2.3 respectively.

When the asynchronous updating with constraint method is used, the value of maximum allowed weight change needs to be set (Step 2 of Algorithm 4.2). To study the effect of the value of maximum allowed weight change, we tested a range of values. The values that we chose are 1, 0.5 and 0.1.

4.3.3 Performance

The performance of the asynchronous updating with constraint method is plotted in Figure 4-10. Figures 4-10a and 4-10b show the final training time and the generalization errors respectively. Each figure is divided into five parts. The first four parts show the performance of using the correlation, one-unit, three-unit and layer block-diagonal Hessian matrices respectively. In each part, the first column shows the performance of the asynchronous updating without constraint method discussed in Section 4.2. The next three columns show the performance of the asynchronous updating with constraint methods used with the maximum allowed weight change of 1, 0.5 and 0.1 respectively. These values are written in the brackets of their respective x-axis labels. The last part of the figure shows the performance of the original method with full Hessian matrix. This figure shows the performance of the training methods used to train the networks to predict the single sine data. The corresponding graphs of the performance of the training methods used to train the networks to predict the composite sine and sunspot data are shown in Figures 4-11 and 4-12 respectively.

4.3.3.1 Generalization performance

The generalization performance of the asynchronous updating with constraint method shown in Figures 4-10b, 4-11b and 4-12b is first examined. Smaller value of maximum allowed weight change means greater constraint. These figures show that as the constraint increased, the average generalization errors decreased and then either decreased slowly or started to increase. The average generalization errors decreased to the levels that were approximately the same as that of the original method with full Hessian matrix. Thus, the asynchronous updating with constraint

method can remedy the poor generalization problem generated by the large incoming weight magnitudes of the hidden units.

The generalization performance of the asynchronous updating with constraint method depends on the value of maximum allowed weight change. The values of 0.5, 0.5 and 1 were chosen when the networks were trained to predict the single sine, composite sine and sunspot data respectively. The choices are based on the following criteria.

i. First, the generalization performance is similar to that of the original method with full Hessian matrix.
ii. Second, the value of maximum allowed weight change should be as large as possible so as not to limit the learning step, that is the weight change, too much.

In our experiments, over 80% of weight magnitudes of the networks that we used the original method to train were below 0.6. Comparing these values with the values of maximum allowed weight change showed that the constraint seldom affected the normal weight update. We suggest the half of the 10 to 90 percentile range of the weights as the value of maximum allowed weight change in the first attempt. First, it can prevent the exceptional large weight update. Second, it does not limit the learning step too much.

4.3.3.2 Training time performance

Then we examine the training time performance of the asynchronous updating with constraint method shown in Figures 4-10a, 4-11a and 4-12a. These figures show that the average final training time decreased and then increased as the constraint increased.

In Table 4-6, the training time performance of the asynchronous updating with constraint method was compared with the training time performance of the asynchronous updating without constraint method. In Table 4-7, it was compared with the training time performance of the original method. The values of maximum allowed weight change used in the comparisons are shown in the second column of Table 4-6. The comparisons were made in terms of the ratios of their average final training time. Table 4-6 shows that the training time performance of the asynchronous updating with constraint method was better than that of the

asynchronous updating without constraint method in most of the experiments. Table 4-7 shows that the training time performance of the asynchronous updating with constraint method was better than that of the original method in all experiments. Moreover, when the networks were trained to predict the single sine or sunspot data, the training time performance of the asynchronous updating with constraint method was significantly better.

Training data	Maximum allowed weight change	Average final training time of the asynchronous updating with constraint method / Average final training time of the asynchronous updating without constraint method			
		correlation block-diagonal matrix	one-unit block-diagonal matrix	three-unit block-diagonal matrix	layer block-diagonal matrix
Single sine	0.5	0.12	0.13	0.46	0.22
Composite sine	0.5	0.33	0.46	0.35	0.65
Sunspot	1	0.16	0.78	0.78	1.32

Table 4-6 Ratios of the average final training time of the asynchronous updating with constraint method to that without constraint method under different training data and block-diagonal Hessian matrices.

Training data	Average final training time of the asynchronous updating with constraint method / Average final training time of the original method			
	correlation block-diagonal matrix	one-unit block-diagonal matrix	three-unit block-diagonal matrix	layer block-diagonal matrix
Single sine	0.12	0.11	0.36	0.18
Composite sine	0.62	0.75	0.49	0.82
Sunspot	0.20	0.18	0.19	0.42

Table 4-7 Ratios of the average final training time of the asynchronous updating with constraint method to that of the original method under different training data and block-diagonal Hessian matrices.

4.3.4 Hidden states and incoming weight magnitudes of the hidden units

After applying the constraint, we examined the hidden states and the incoming weight magnitudes of the hidden units, which are described in Sections 4.3.4.1 and 4.3.4.2 respectively.

4.3.4.1 Hidden states

The hidden states of the networks trained to predict the single sine, composite sine and sunspot data are shown in Figures 4-13, 4-14 and 4-15 respectively. The asynchronous updating with constraint method and the one-unit block-diagonal Hessian matrix were used to train these networks. In most trials and time, the states of most hidden units were not in the saturation region. The average absolute hidden states are 0.51, 0.34 and 0.34 respectively. They are smaller than those of the asynchronous updating without constraint method shown in Table 4-3. The observations about the hidden states were the same if the other block-diagonal Hessian matrices were used instead.

4.3.4.2 Incoming weight magnitudes of the hidden units

The magnitudes of the maximum, minimum, 90th percentile and 10th percentile of the incoming weights of the hidden units of the networks trained to predict the single sine, composite sine and sunspot data are shown in Figures 4-16 to 4-18. The placements of the training methods on the x-axes are the same as those shown in Figures 4-10 to 4-12. These figures show that the ranges of incoming weights of the hidden units decreased and then either decreased slowly or started to increase as the constraint increased. The range of incoming weights of the hidden units of the networks that we used the asynchronous updating with constraint method to train was close to that of the networks that we used the original method to train.

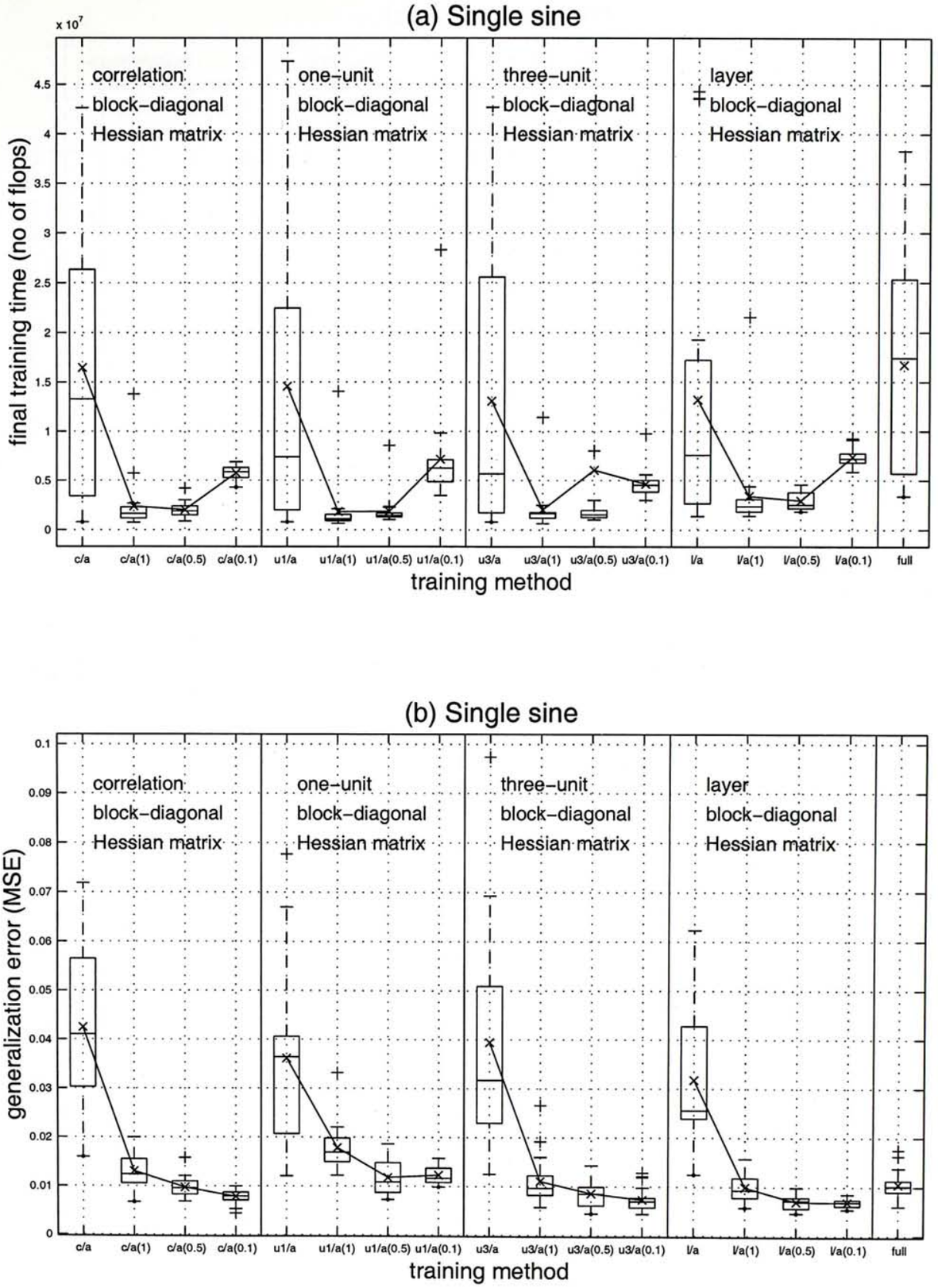


Figure 4-10 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using different asynchronous updating methods and block-diagonal Hessian matrices. The values of maximum allowed weight change are written in the brackets of the labels of the x-axes. The labels of the x-axes are explained in Table 4-4. The single sine training data was used in these experiments.

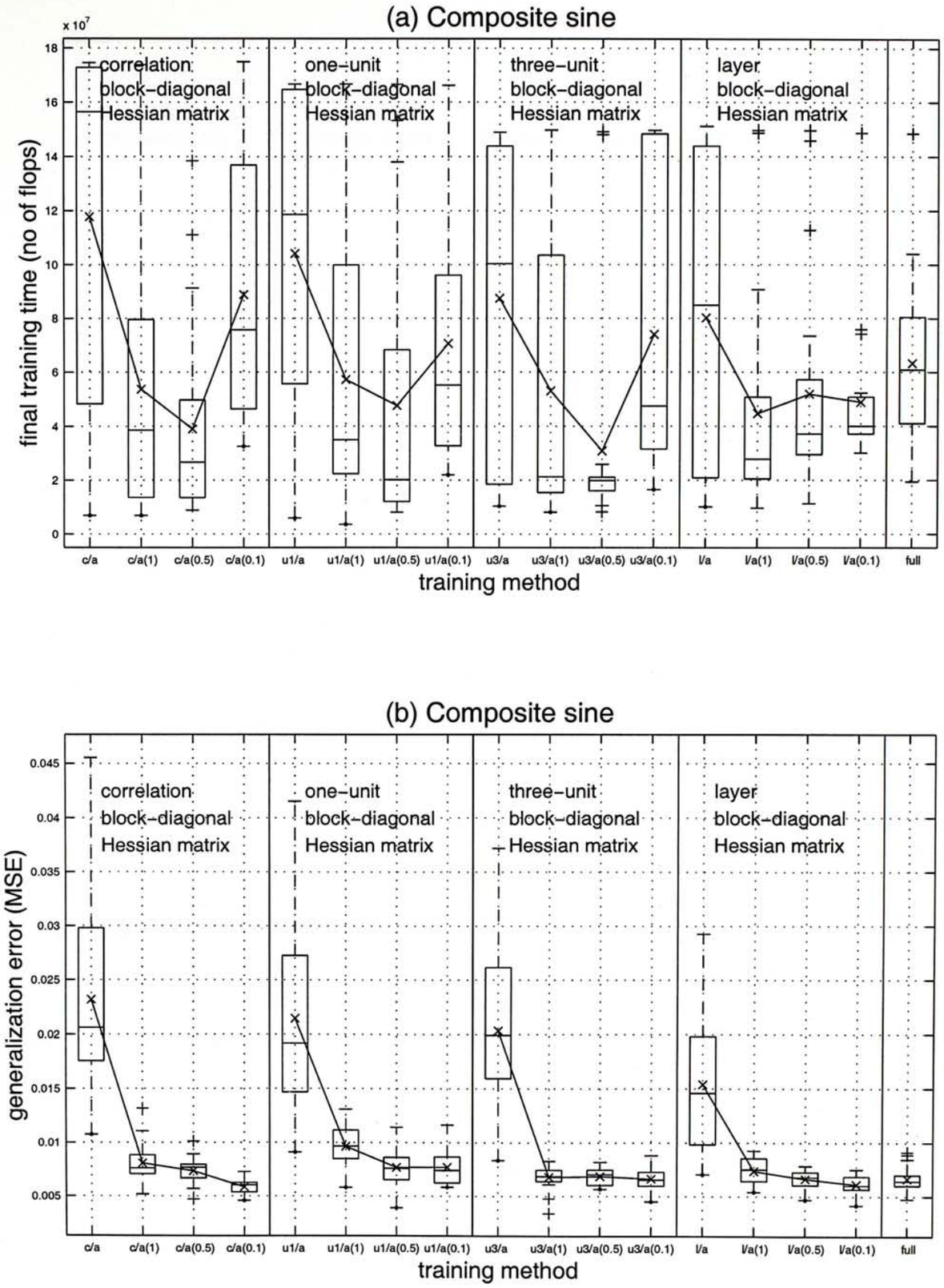


Figure 4-11 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using different asynchronous updating methods and block-diagonal Hessian matrices. The values of maximum allowed weight change are written in the brackets of the labels of the x-axes. The labels of the x-axes are explained in Table 4-4. The composite sine training data was used in these experiments.

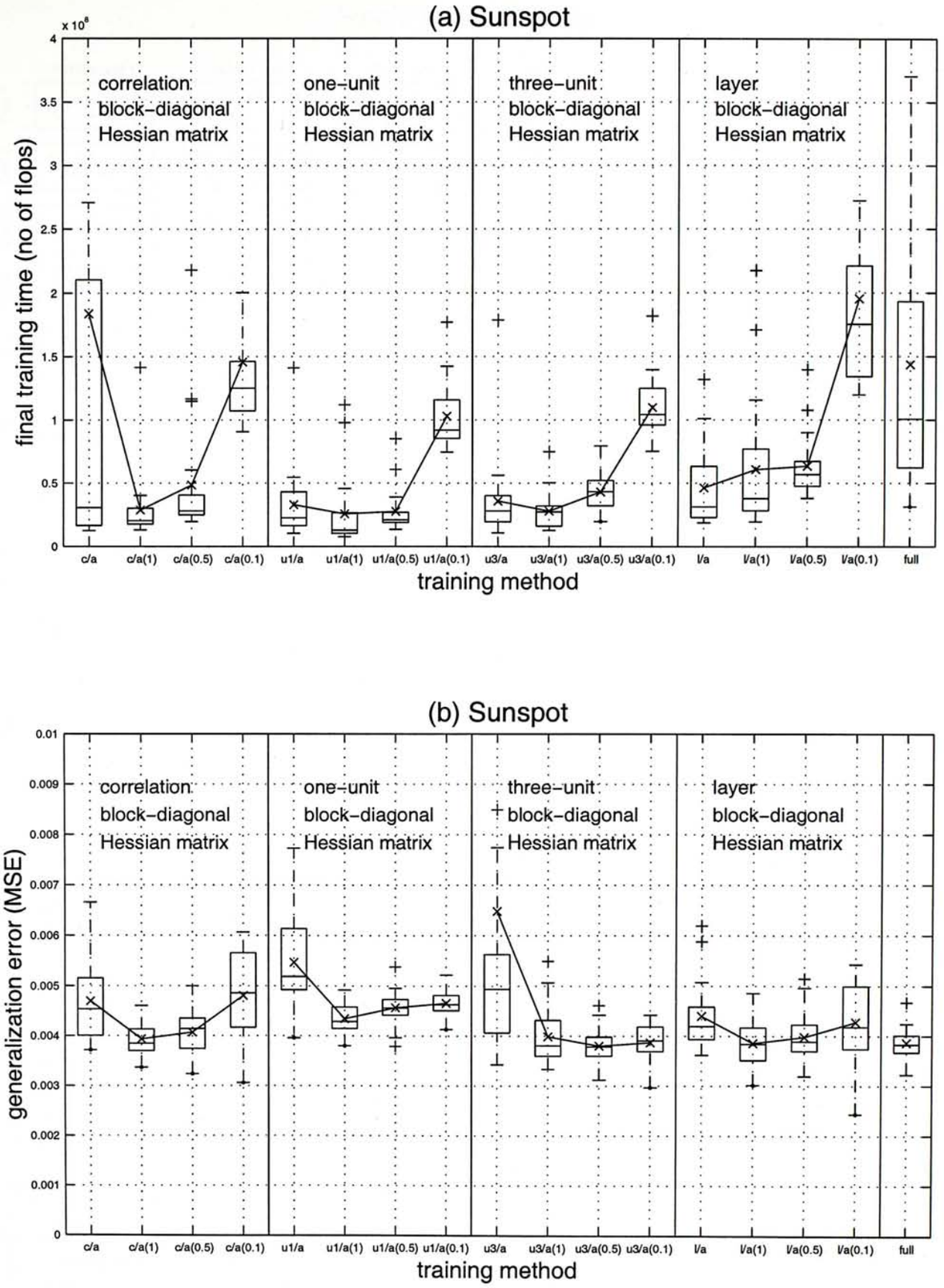


Figure 4-12 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using different asynchronous updating methods and block-diagonal Hessian matrices. The values of maximum allowed weight change are written in the brackets of the labels of the x-axes. The labels of the x-axes are explained in Table 4-4. The sunspot training data was used in these experiments.

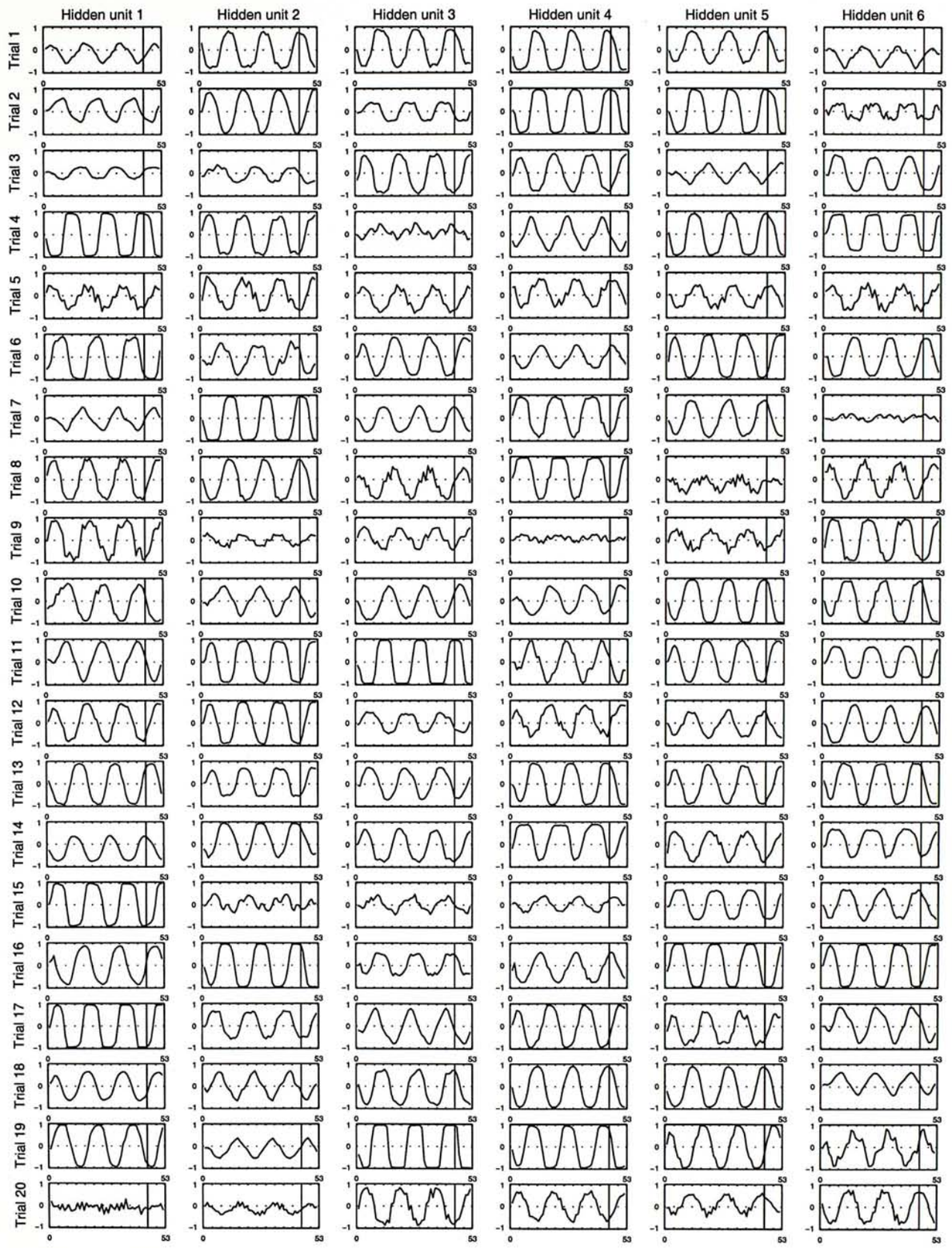


Figure 4-13 Hidden states of 20 networks of different trials. The asynchronous updating with constraint method and the one-unit block-diagonal Hessian matrix were used to train these networks. The maximum allowed weight change of 0.5 was used. The single sine training data was used in these experiments.

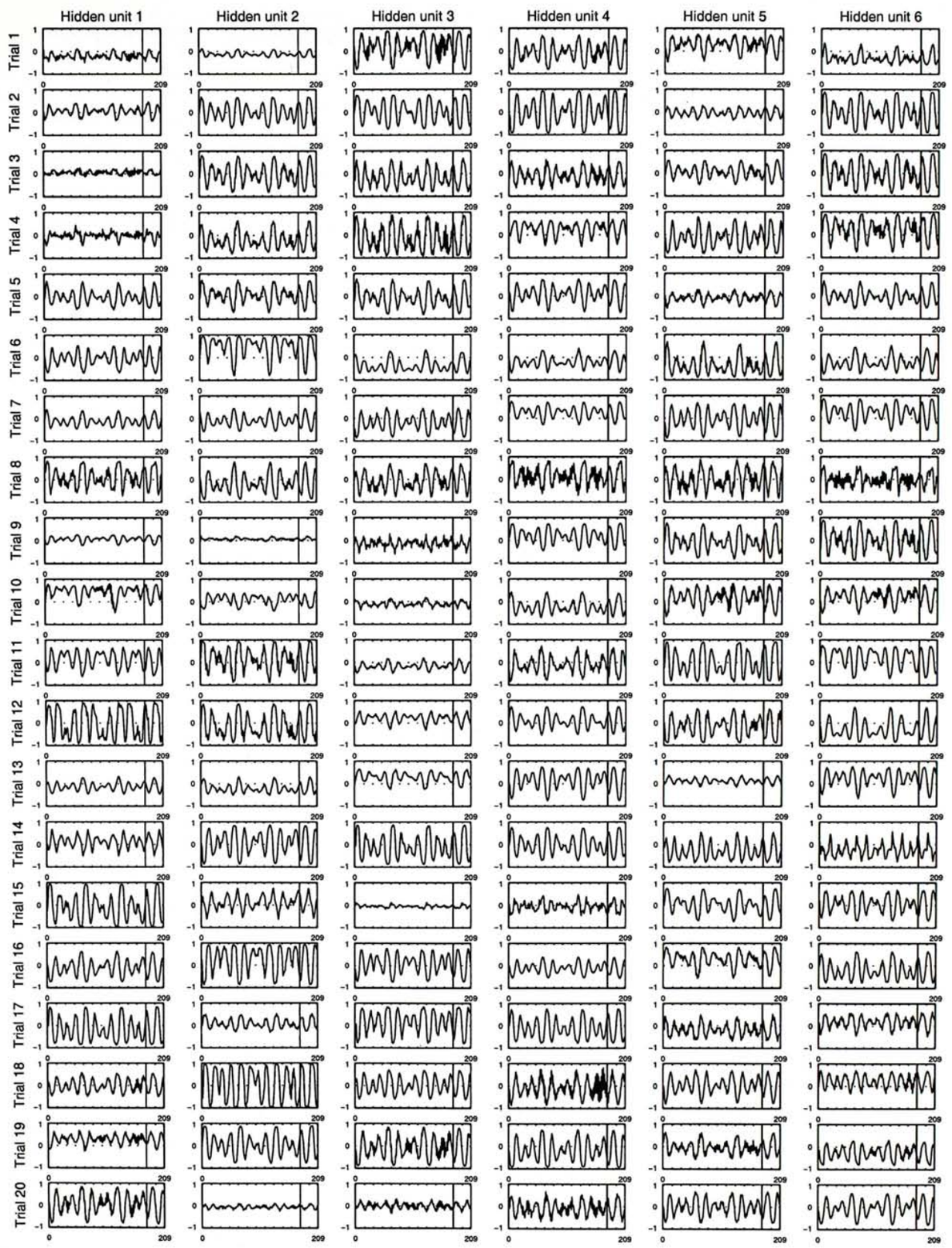


Figure 4-14 Hidden states of 20 networks of different trials. The asynchronous updating with constraint method and the one-unit block-diagonal Hessian matrix were used to train these networks. The maximum allowed weight change of 0.5 was used. The composite sine training data was used in these experiments.

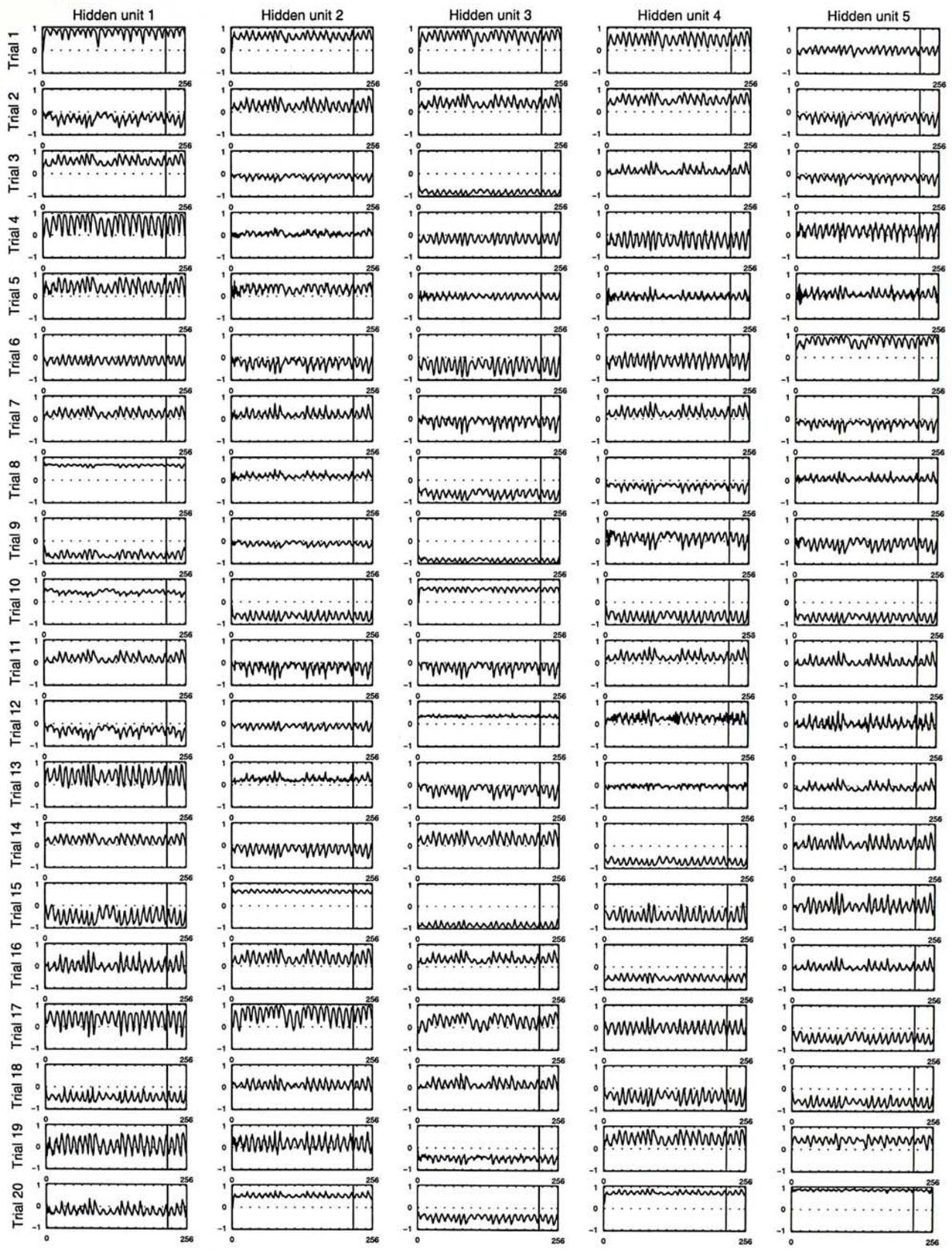


Figure 4-15 Hidden states of 20 networks of different trials. The asynchronous updating with constraint method and the one-unit block-diagonal Hessian matrix were used to train these networks. The maximum allowed weight change of 1 was used. The sunspot training data was used in these experiments.

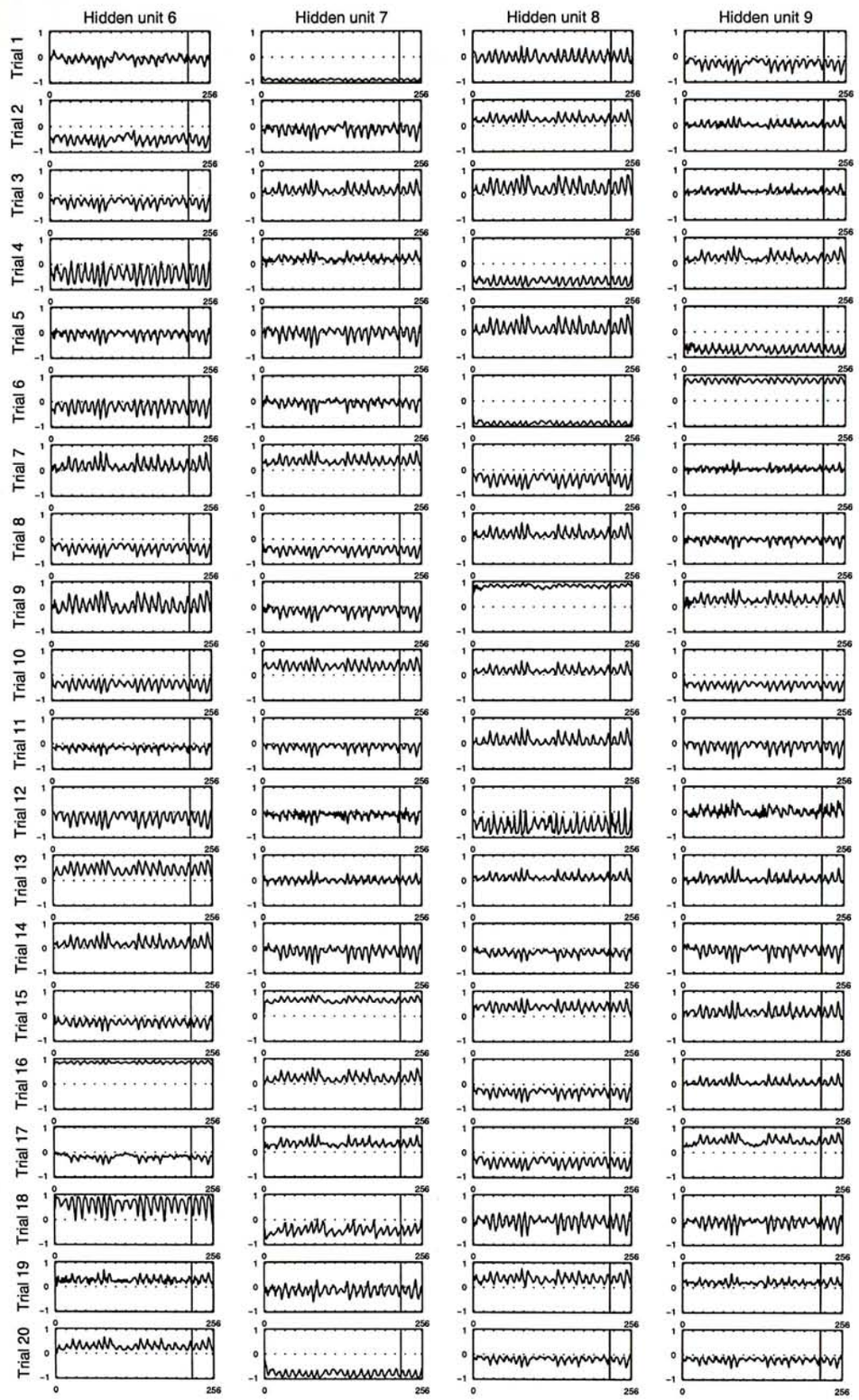
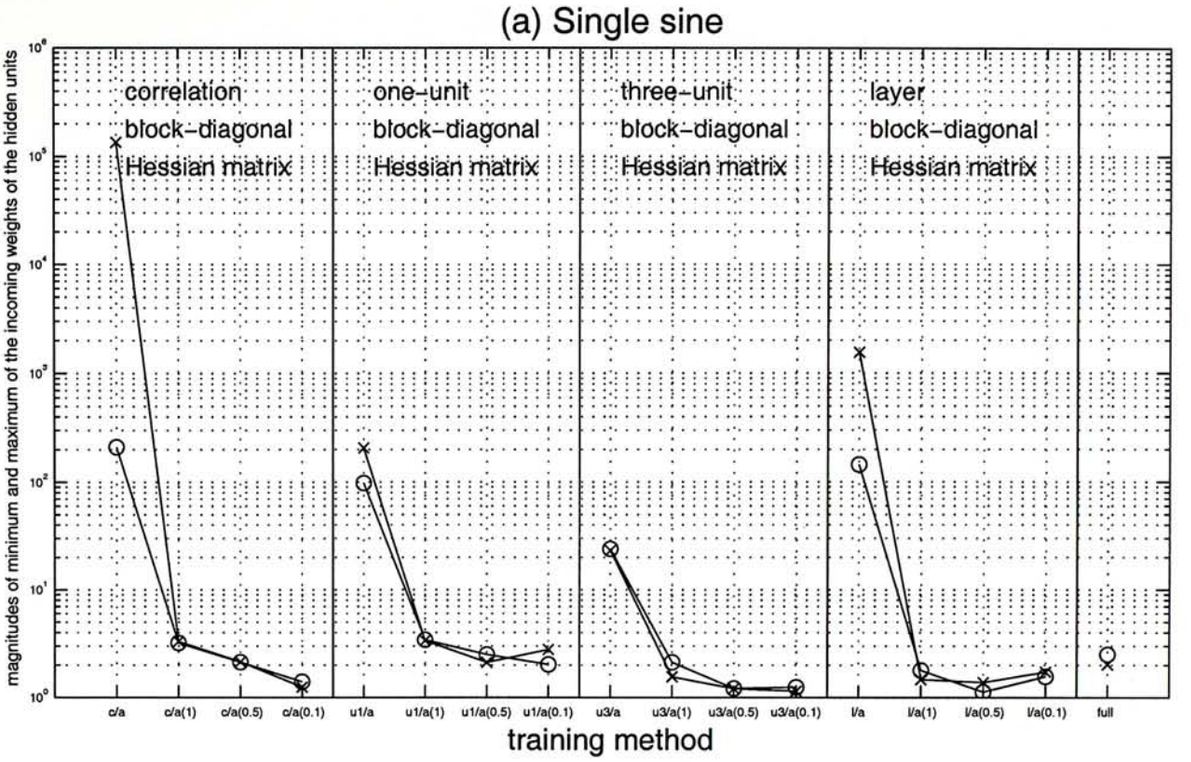


Figure 4-15 Hidden states of 20 networks of different trials. The asynchronous updating with constraint method and the one-unit block-diagonal Hessian matrix were used to train these networks. The maximum allowed weight change of 1 was used. The sunspot training data was used in these experiments. (continued)



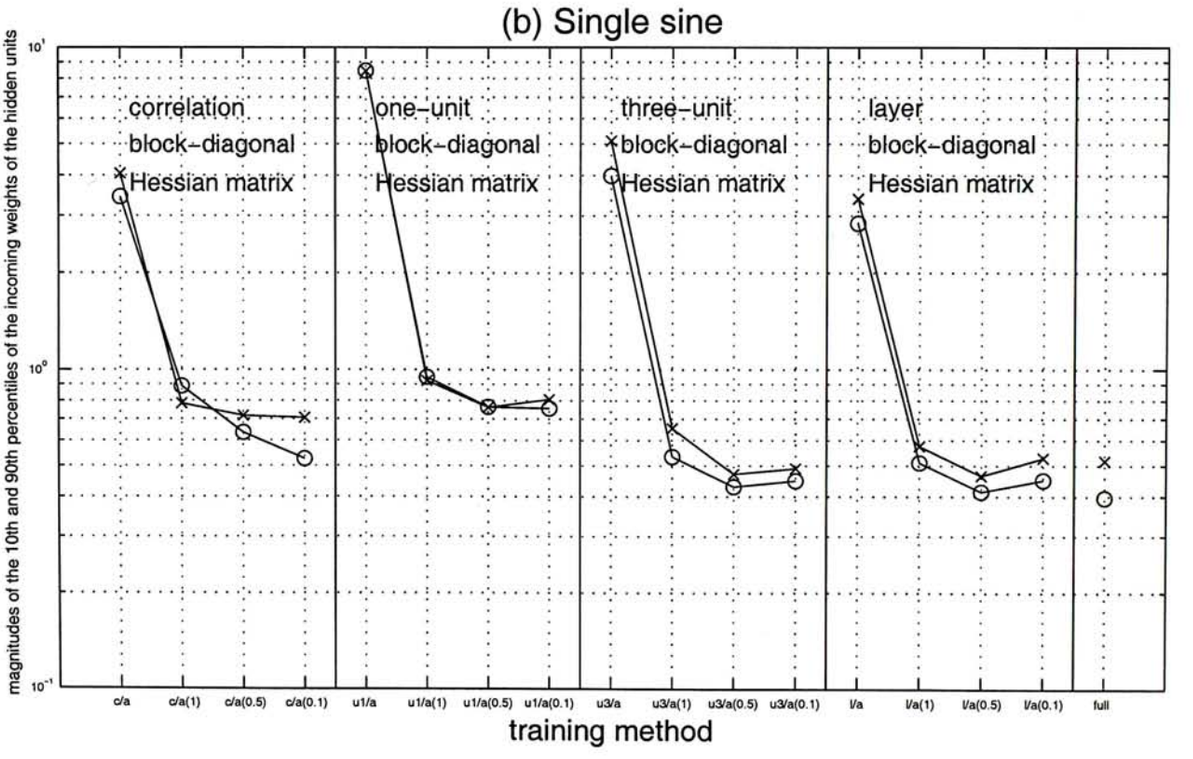
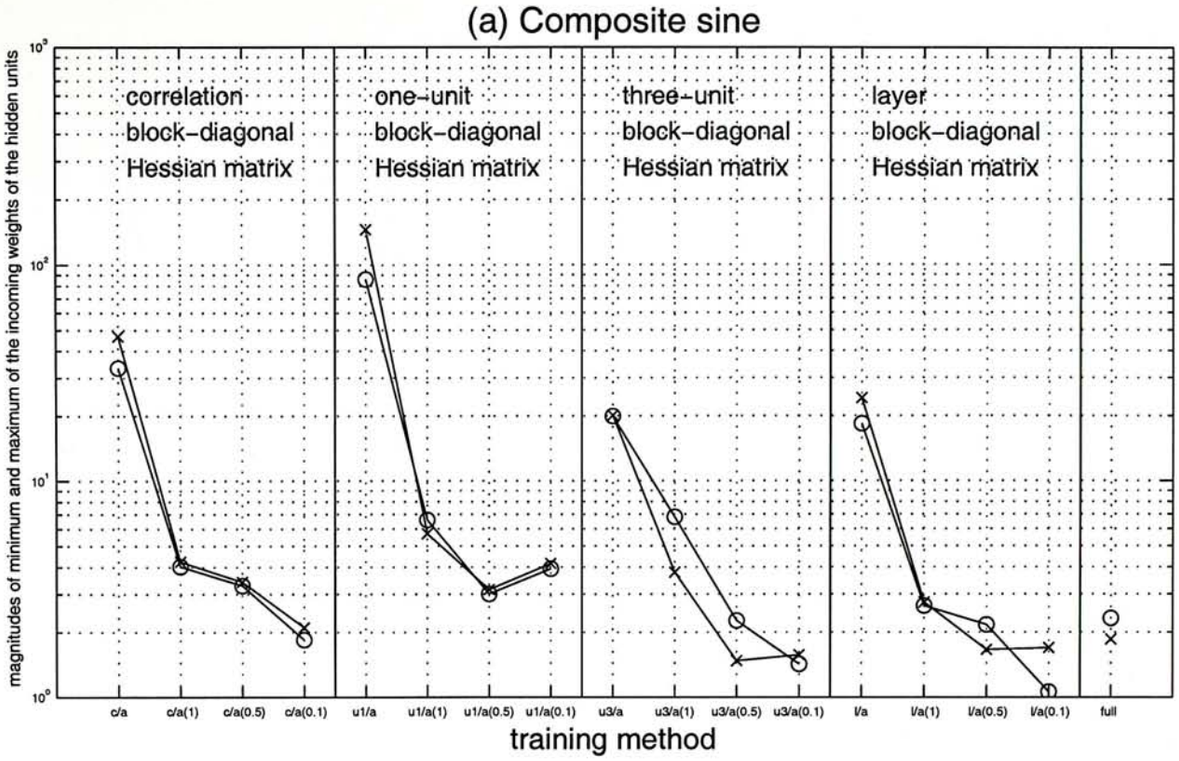


Figure 4-16 Magnitudes of (a) minimum 'o' and maximum 'x' and (b) the 10th percentile 'o' and 90th percentile 'x' of the incoming weights of the hidden units. The x-axes show the training methods using different asynchronous updating methods and block-diagonal Hessian matrices. The values of maximum allowed weight change are written in the brackets of the labels of the x-axes. The labels of the x-axes are explained in Table 4-4. The single sine training data was used in these experiments.



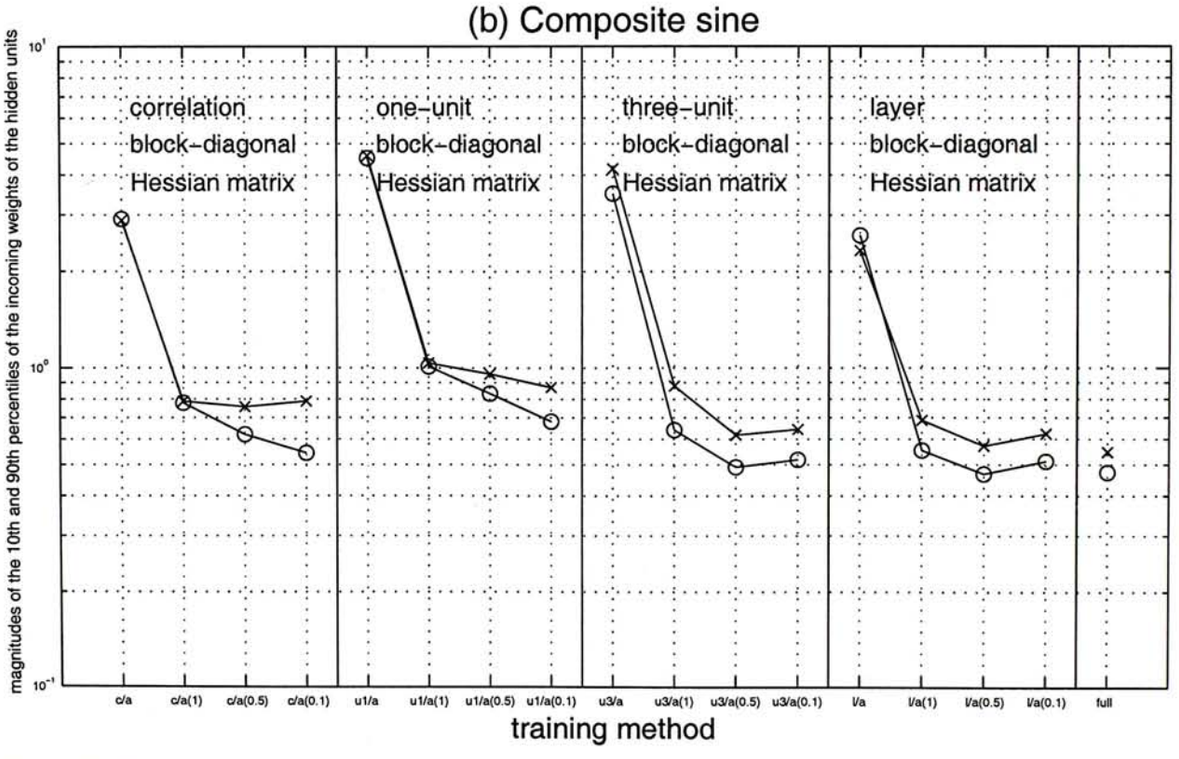
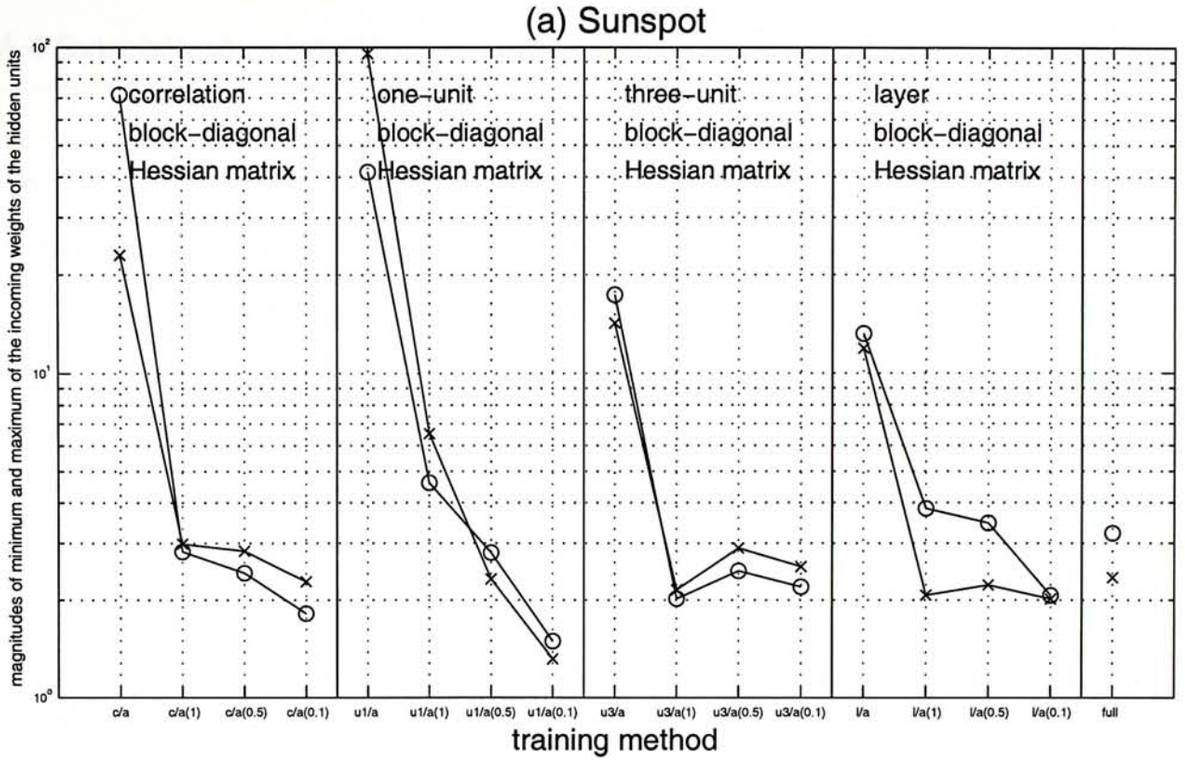


Figure 4-17 Magnitudes of (a) minimum 'o' and maximum 'x' and (b) the 10th percentile 'o' and 90th percentile 'x' of the incoming weights of the hidden units. The x-axes show the training methods using different asynchronous updating methods and block-diagonal Hessian matrices. The values of maximum allowed weight change are written in the brackets of the labels of the x-axes. The labels of the x-axes are explained in Table 4-4. The composite sine training data was used in these experiments.



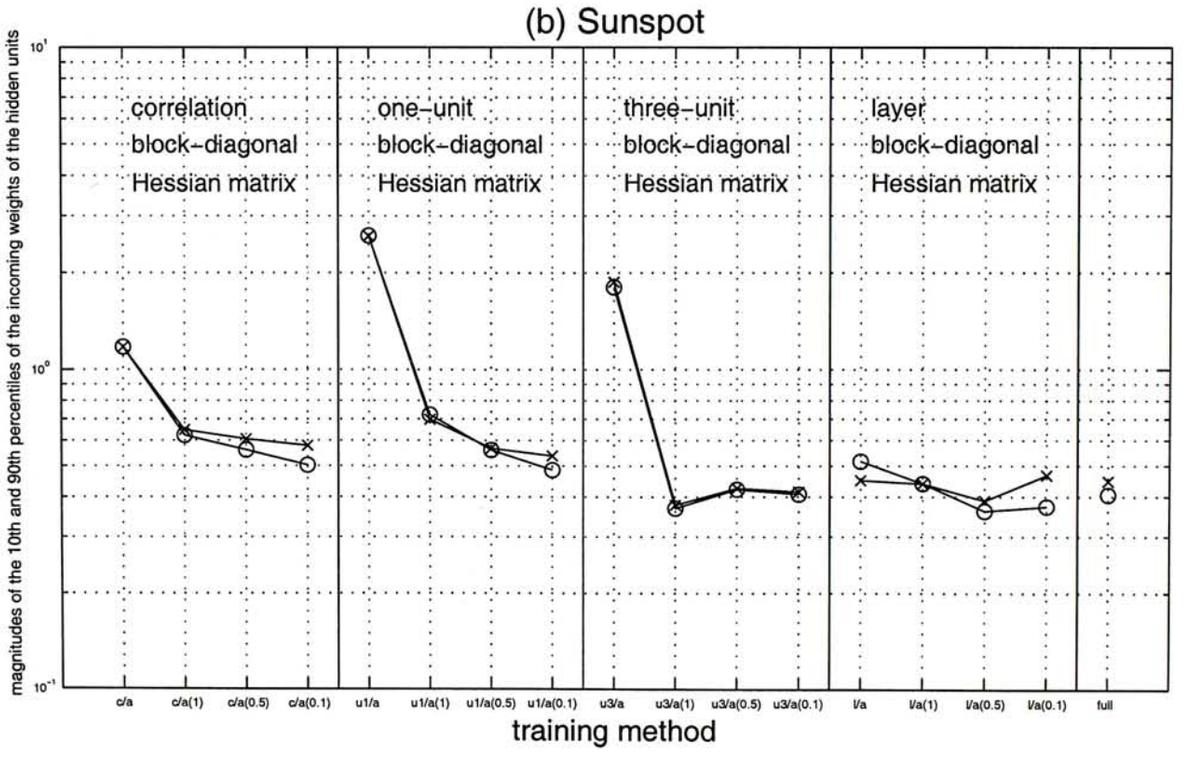


Figure 4-18 Magnitudes of (a) minimum 'o' and maximum 'x' and (b) the 10th percentile 'o' and 90th percentile 'x' of the incoming weights of the hidden units. The x-axes show the training methods using different asynchronous updating methods and block-diagonal Hessian matrices. The values of maximum allowed weight change are written in the brackets of the labels of the x-axes. The labels of the x-axes are explained in Table 4-4. The sunspot training data was used in these experiments.

4.4 Synchronous updating methods

As mentioned in the beginning of this chapter, the weight updating methods are divided into the asynchronous and synchronous updating methods. The asynchronous updating method has been studied in Sections 4.2 and 4.3. In this section, the synchronous updating method, which updates weights of all blocks at a time, is studied. In the following, we will first describe two different ways of implementing the synchronous updating method. The evaluation of these two synchronous updating methods is the same as that of the asynchronous updating method.

4.4.1 Single λ and multiple λ's synchronous updating methods

As shown in Equations 2.14, each decomposed system of updating equations has its own learning parameter λ_i. We treat these learning parameters in two ways leading to two different synchronous updating methods.

i. One way is using one learning parameter λ for weights of all blocks. We call this method the *single λ synchronous updating method* and describe it in Section 4.4.1.1.

ii. The other way is using different λ_i's for weights of different blocks $\mathbf{w}_i$'s. We call this method the *multiple λ's synchronous updating method* and describe it in Section 4.4.1.2.

Lower computation load per complete weight update is the feature of the single λ synchronous updating method while using λ's tailored to weights of different blocks is the feature of the multiple λ's synchronous updating method.

4.4.1.1 Algorithm of single λ synchronous updating method

The implementation of the single λ synchronous updating method is shown in Algorithm 4.3.

Algorithm 4.3 Single λ synchronous updating method

1. $\lambda = 0.001$, $\beta = 10$, iteration = 0 and finished = false
2. WHILE finished = false
3. FOR i = 1 TO B
4. calculate $\Delta \mathbf{w}_i = -(\mathbf{J}_i^T \mathbf{J}_i + \lambda \mathbf{I}_i)^{-1} \mathbf{J}_i^T \mathbf{e}$
5. END
6. calculate $E(\mathbf{w}+\Delta\mathbf{w})$ where $\Delta \mathbf{w} = [\Delta \mathbf{w}_1^T \Delta \mathbf{w}_2^T \ldots \Delta \mathbf{w}_B^T]^T$
7. WHILE ($\lambda \leq$ maximum_λ) AND ($E(\mathbf{w}+\Delta\mathbf{w}) \geq E(\mathbf{w})$)
8. increase λ by a factor β
9. calculate $\Delta \mathbf{w}_1, \Delta \mathbf{w}_2, \ldots \Delta \mathbf{w}_B$
10. calculate $E(\mathbf{w}+\Delta\mathbf{w})$
11. END
12. IF $\lambda \leq$ maximum_λ
13. update $\mathbf{w}$ ($\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$)
14. decrease λ by a factor β
15. iteration $\leftarrow$ iteration + 1
16. END
17. calculate $\mathbf{J}^T\mathbf{e}$
18. IF (iteration > maximum_iteration) OR
    (minimum of validation error is reached = true) OR
    ($\mathbf{J}^T\mathbf{e}$ < minimum_gradient) OR (λ > maximum_λ)
19. finished = true
20. END
21. END

It is similar to the algorithm of the original method with full Hessian matrix described in Section 1.3.2.2. The difference is that we use the block-diagonal Hessian matrix to calculate $\Delta\mathbf{w}$ in the single λ synchronous updating method (Steps 3 to 5 or 9) whereas we use full Hessian matrix in the original method.

4.4.1.2 Algorithm of multiple λ's synchronous updating method

The implementation of the multiple λ's synchronous updating method is shown in Algorithm 4.4.

Algorithm 4.4 Multiple λ's synchronous updating method

1. $\lambda_1, \lambda_2, \ldots, \lambda_B = 0.001$, $\beta = 10$, iteration = 0 and finished = false
2. WHILE finished = false
3. FOR i = 1 TO B
4. calculate $\Delta \mathbf{w}_i = -(\mathbf{J}_i^T \mathbf{J}_i + \lambda_i \mathbf{I}_i)^{-1} \mathbf{J}_i^T \mathbf{e}$
5. calculate $E(\mathbf{w}_i + \Delta \mathbf{w}_i)$
6. WHILE ($\lambda_i \leq$ maximum_λ) AND ($E(\mathbf{w}_i + \Delta \mathbf{w}_i) \geq E(\mathbf{w}_i)$)
7. increase λ_i by a factor β
8. calculate $\Delta \mathbf{w}_i$
9. calculate $E(\mathbf{w}_i + \Delta \mathbf{w}_i)$
10. END
11. END
12. overflow_λ = 0
13. FOR i = 1 TO B
14. IF $\lambda_i >$ maximum_λ
15. overflow_$\lambda \leftarrow$ overflow_λ + 1
16. END
17. END
18. calculate $E(\mathbf{w} + \Delta \mathbf{w})$ where $\Delta \mathbf{w} = [\Delta \mathbf{w}_1^T \Delta \mathbf{w}_2^T \ldots \Delta \mathbf{w}_B^T]^T$
19. WHILE (overflow_λ < B) AND ($E(\mathbf{w} + \Delta \mathbf{w}) \geq E(\mathbf{w})$)
20. increase $\lambda_1, \lambda_2, \ldots, \lambda_B$ by a factor β
21. overflow_λ = 0
22. FOR i = 1 TO B
23. IF $\lambda_i >$ maximum_λ
24. overflow_$\lambda \leftarrow$ overflow_λ + 1
25. END
26. END
27. calculate $\Delta \mathbf{w}_1, \Delta \mathbf{w}_2, \ldots \Delta \mathbf{w}_B$
28. calculate $E(\mathbf{w} + \Delta \mathbf{w})$
29. END
30. IF overflow_λ < B
31. update $\mathbf{w}$ ($\mathbf{w} \leftarrow \mathbf{w} + \Delta \mathbf{w}$)

32. decrease $\lambda_1, \lambda_2, \ldots, \lambda_B$ by a factor β

33. iteration $\leftarrow$ iteration + 1

34. END

35. calculate $\mathbf{J}^T\mathbf{e}$

36. IF (iteration > maximum_iteration) OR
(minimum of validation error is reached = true) OR
($\mathbf{J}^T\mathbf{e}$ < minimum_gradient) OR (overflow_λ = B)

37. finished = true

38. END

39. END

In Steps 3 to 11, λ_i tailored to each decomposed system of updating equations is first found. Then, weights of all blocks are updated synchronously (Step 18). The cost function E may not decrease after taking the step $\Delta\mathbf{w}$ provided that the block-diagonal Hessian matrix deviates from the full Hessian matrix. In that case, all the λ_i's have to be scaled up to obtain a decrease in the cost function E (Step 20).

4.4.1.3 Method of study

We evaluated the performance of both synchronous updating methods by comparing it with the performance of the original method with full Hessian matrix. Like the evaluation of the asynchronous updating methods described in Sections 4.2 and 4.3, the synchronous updating method would be used with the correlation, one-unit, three-unit and layer block-diagonal Hessian matrices described in Sections 2.3.1 to 2.3.4 respectively. These training methods were used to train the layered fully recurrent network described in Section 1.2.3 to predict the single sine, composite sine and sunspot data described in Sections 3.2.1 to 3.2.3 respectively.

4.4.1.4 Performance

The performance of the single λ and multiple λ's synchronous updating methods is plotted in Figure 4-19. Figures 4-19a and 4-19b show the final training time and the generalization errors respectively. Each figure is divided into five parts. The first four parts show the performance of using the correlation, one-unit,

three-unit and layer block-diagonal Hessian matrices respectively. In each part, the performance of the single λ and multiple λ's synchronous updating methods is plotted in the first two columns respectively. The performance of the *multiple λ's with line search method* described in Section 4.4.2 is plotted in the last column. Finally, the last part of the figure shows the performance of the original method with full Hessian matrix. This figure shows the performance of the training methods used to train the networks to predict the single sine data. The corresponding graphs of the performance of the training methods used to train the networks to predict the composite sine and sunspot data are plotted in Figures 4-20 and 4-21 respectively.

These figures show that both single λ and multiple λ's synchronous updating methods required more training time than the original method with full Hessian matrix under different training data and block-diagonal Hessian matrices. The comparisons are summarized in Tables 4-8 and 4-9 in terms of the ratios of their average final training time. These tables show that the average final training time of the synchronous updating methods was about 2.1 times more than that of the original method when the networks were trained to predict the single and composite sine data. The average final training time of the synchronous updating methods was about 4.4 times more than that of the original method when the networks were trained to predict the sunspot data. The long training time problem of both synchronous updating methods is investigated in the next section.

Training data	Average final training time of the single λ synchronous updating method / Average final training time of the original method			
	correlation block-diagonal matrix	one-unit block-diagonal matrix	three-unit block-diagonal matrix	layer block-diagonal matrix
Single sine	2.45	1.74	1.55	1.80
Composite sine	2.27	2.26	2.06	2.11
Sunspot	4.77	4.83	4.47	3.98

Table 4-8 Ratios of the average final training time of the single λ synchronous updating method to that of the original method under different training data and block-diagonal Hessian matrices.

Training data	Average final training time of the multiple λ's synchronous updating method / Average final training time of the original method			
	correlation block-diagonal matrix	one-unit block-diagonal matrix	three-unit block-diagonal matrix	layer block-diagonal matrix
Single sine	2.41	2.07	1.35	2.36
Composite sine	2.25	2.23	2.17	2.24
Sunspot	4.28	4.67	4.03	4.44

Table 4-9 Ratios of the average final training time of the multiple λ's synchronous updating method to that of the original method under different training data and block-diagonal Hessian matrices.

4.4.1.5 Investigation on long training time: analysis of λ

We investigated the long training time problem of the single λ and multiple λ's synchronous updating methods by examining their learning parameters λ's. The values of λ's during the whole training period were recorded. They are shown in Figure 4-22. These values are the median values of λ's of 20 trials. The one-unit block-diagonal Hessian matrix was used. The light-colored lines in Figures 4-22a, 4-22b and 4-22c represent λ's of the single λ, multiple λ's and *multiple λ's with line search* (described in Section 4.4.2) synchronous updating methods respectively. In each figure, λ of the original method represented by the dark-colored line is plotted for comparison. This figure shows the values of λ's of the training methods used to train the networks to predict the single sine data. The corresponding graphs of the values of λ's of the training methods used to train the networks to predict the composite sine and sunspot data are shown in Figures 4-23 and 4-24 respectively.

These figures show that the value of λ of the single λ synchronous updating method and the values of most λ's of the multiple λ's synchronous updating method were often larger than that of the original method in the early training period. The observations about the values of λ's were the same if the other block-diagonal Hessian matrices were used instead.

The values of λ's of both synchronous updating methods are larger because of the deviation between the full Hessian matrix and the block-diagonal Hessian matrix. Larger λ means that the step size is smaller and the direction is closer to the gradient descent direction. Both factors make the algorithms learn slowly. To retain

the approximate Gauss-Newton direction, we introduce the step size parameter, which is adapted after λ for each decomposed system of updating equations is found. We call this the *multiple λ's with line search method*, which will further be described in Section 4.4.2.

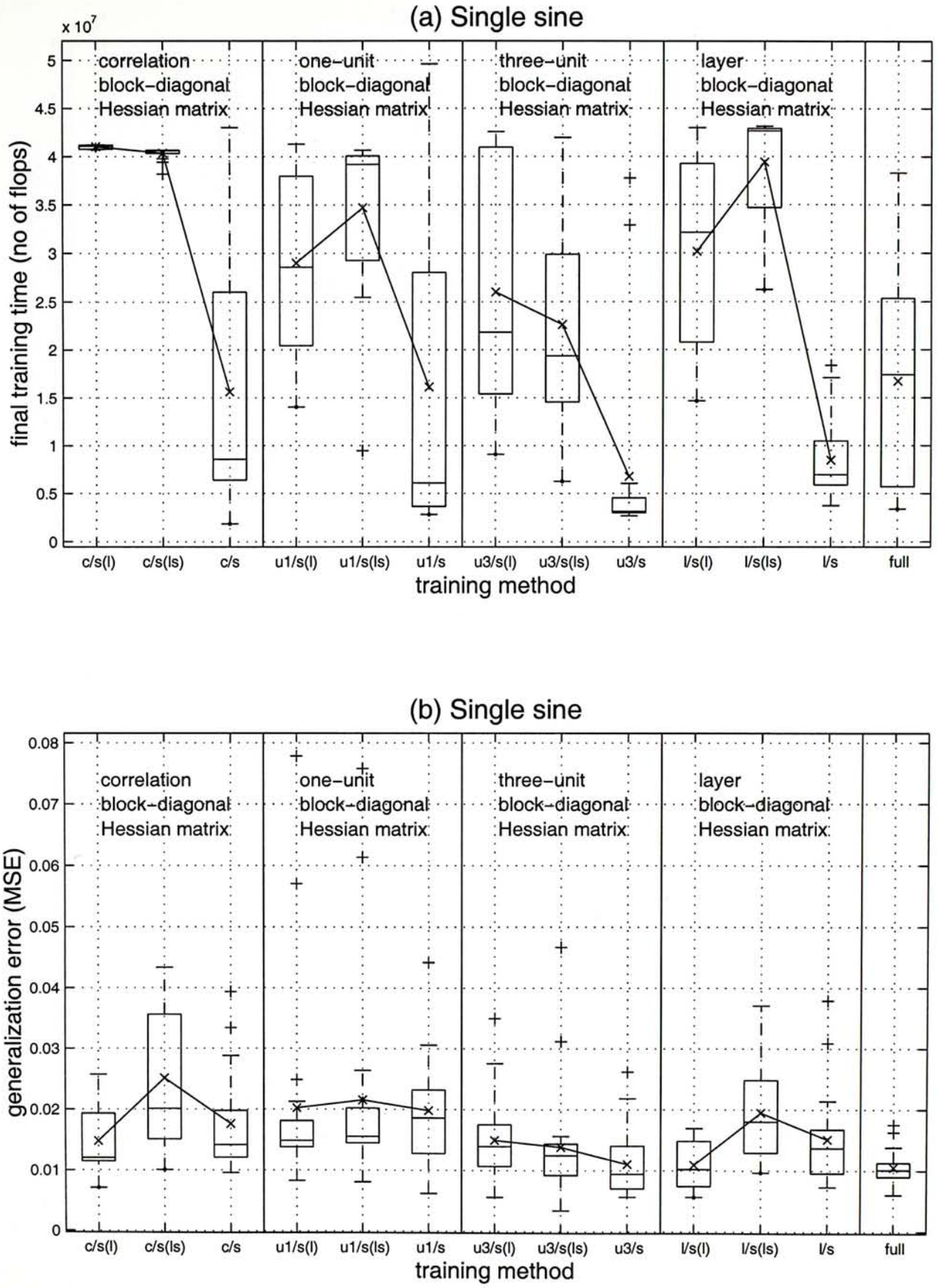


Figure 4-19 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using different implementation of synchronous updating methods and block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 4-5. The single sine training data was used in these experiments.

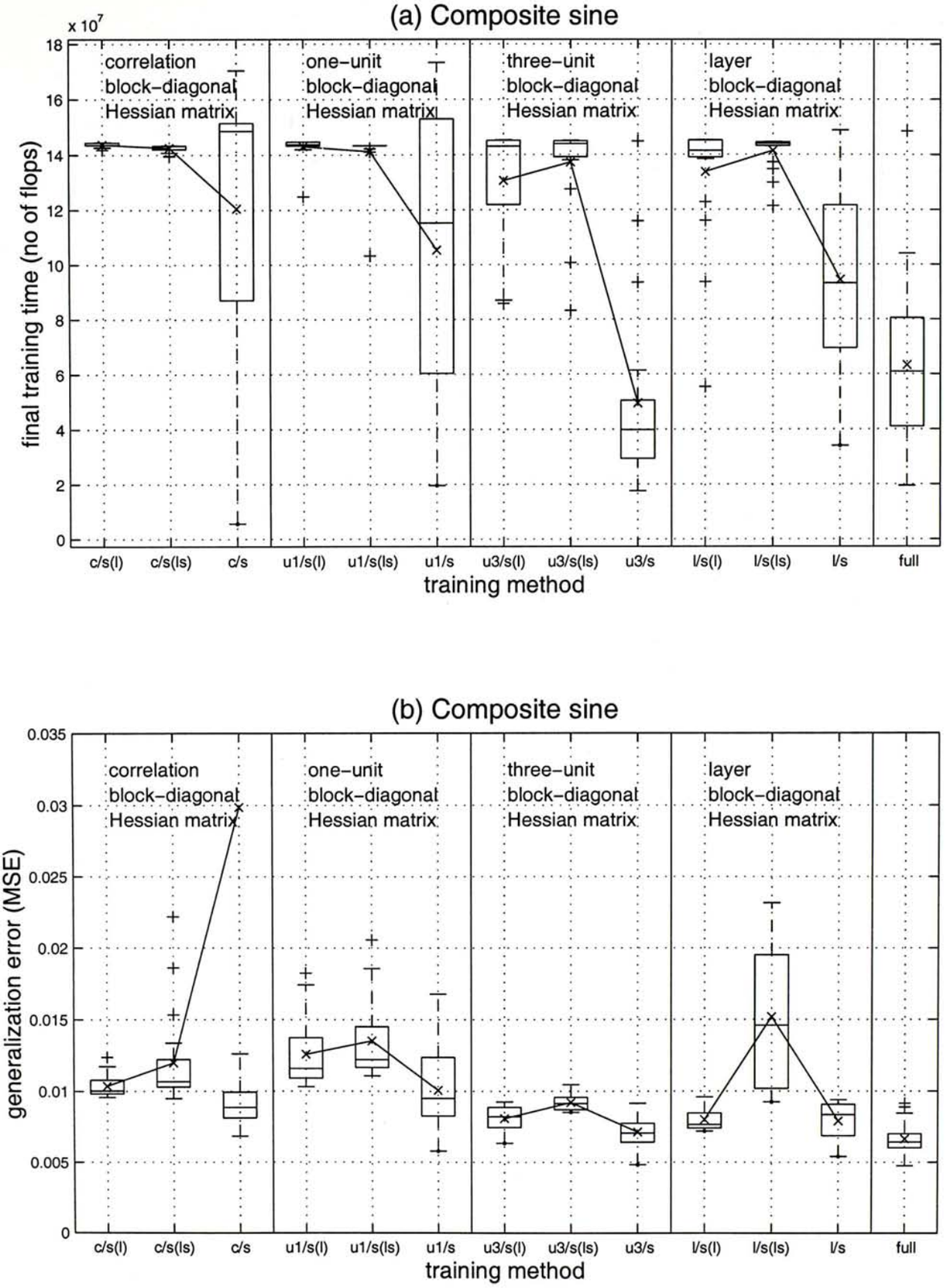


Figure 4-20 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using different implementation of synchronous updating methods and block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 4-5. The composite sine training data was used in these experiments.

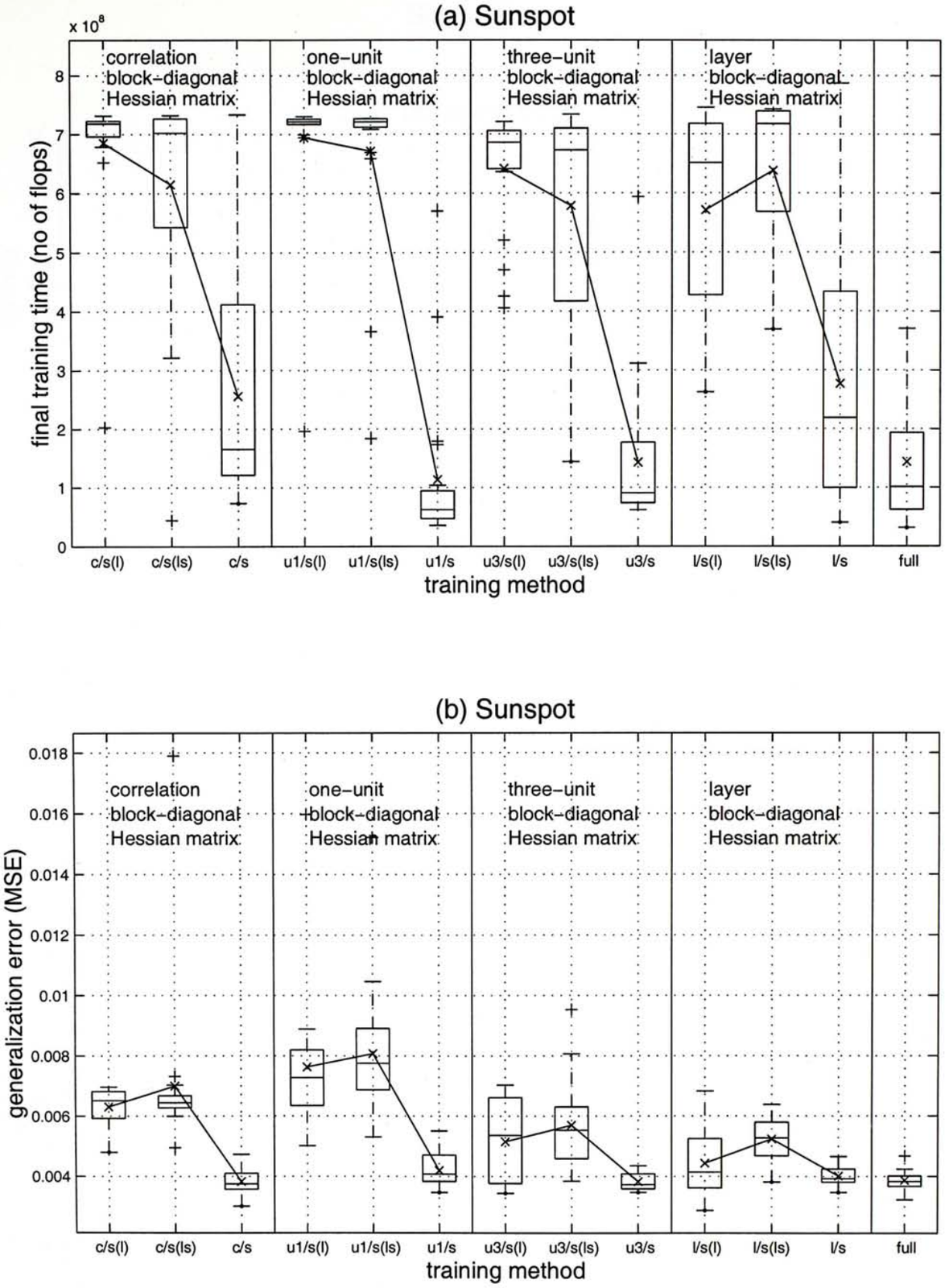


Figure 4-21 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using different implementation of synchronous updating methods and block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 4-5. The sunspot training data was used in these experiments.

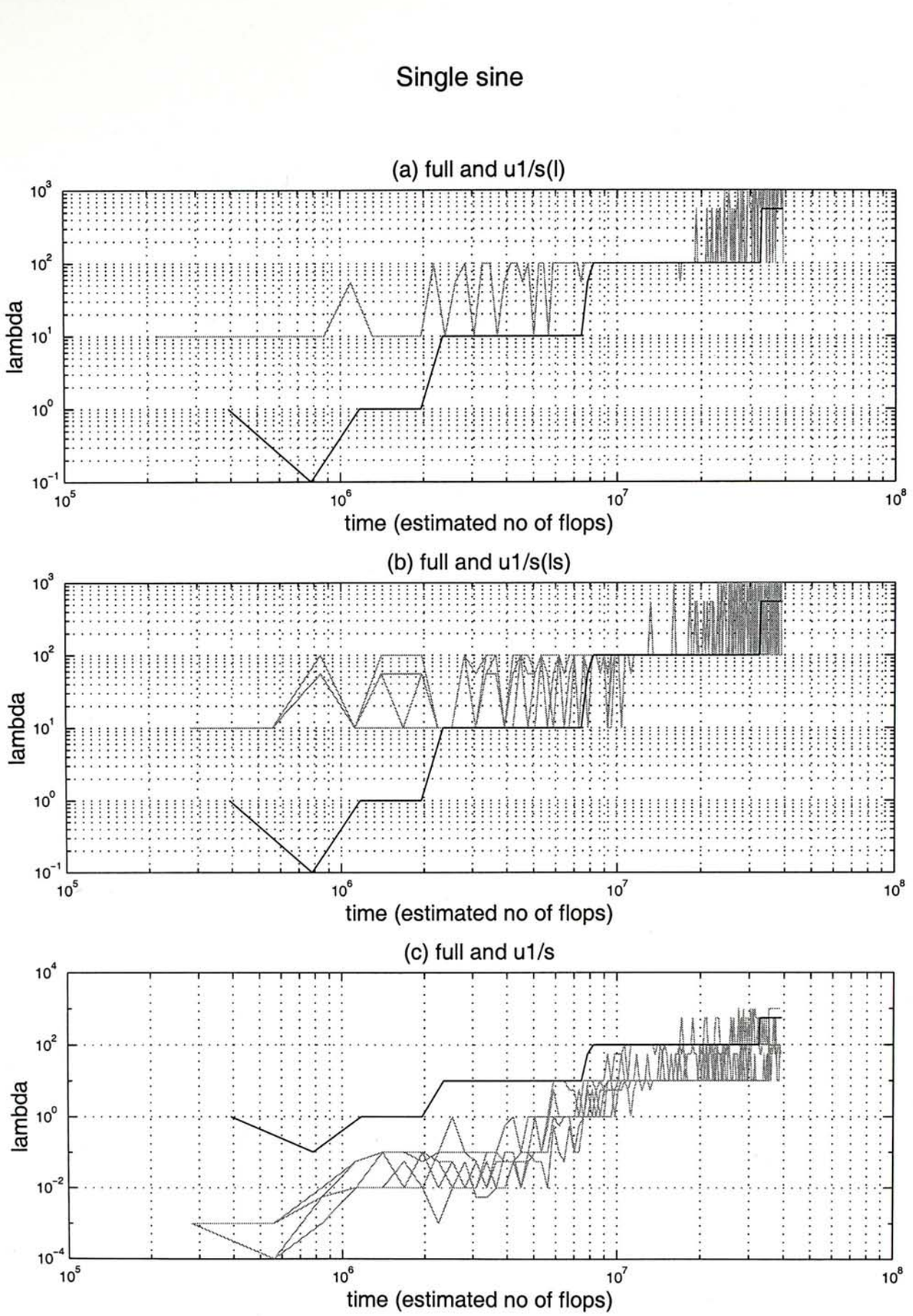


Figure 4-22 Values of λ's during the whole training period. λ of the original method with full Hessian matrix is represented by the dark-colored line. λ's of the single λ, multiple λ's and multiple λ's with line search synchronous updating methods shown in (a), (b) and (c) respectively are represented by the light-colored lines. The one-unit block-diagonal Hessian matrix and the single sine training data were used in these experiments.

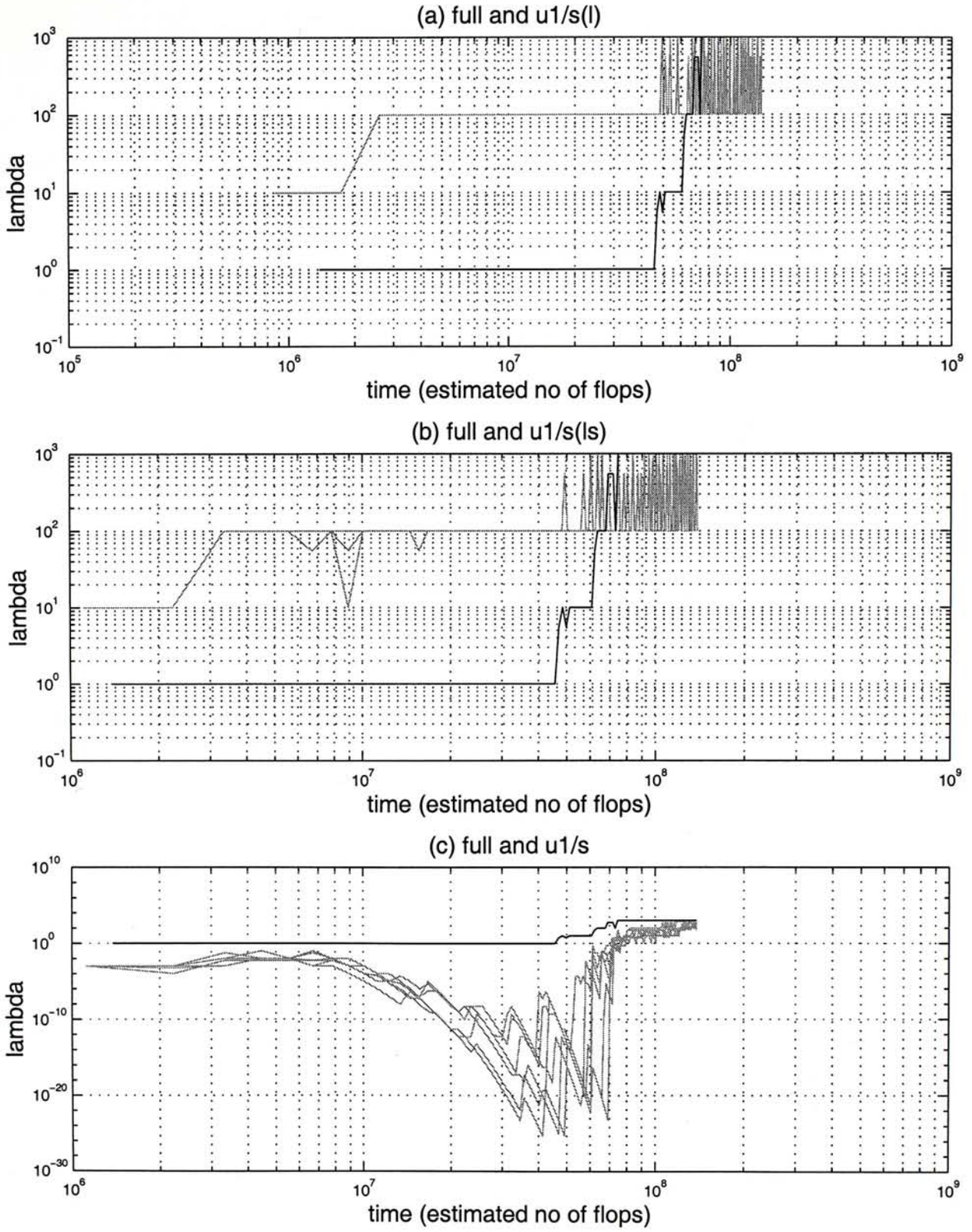


Figure 4-23 Values of λ's during the whole training period. λ of the original method with full Hessian matrix is represented by the dark-colored line. λ's of the single λ, multiple λ's and multiple λ's with line search synchronous updating methods shown in (a), (b) and (c) respectively are represented by the light-colored lines. The one-unit block-diagonal Hessian matrix and the composite sine training data were used in these experiments.

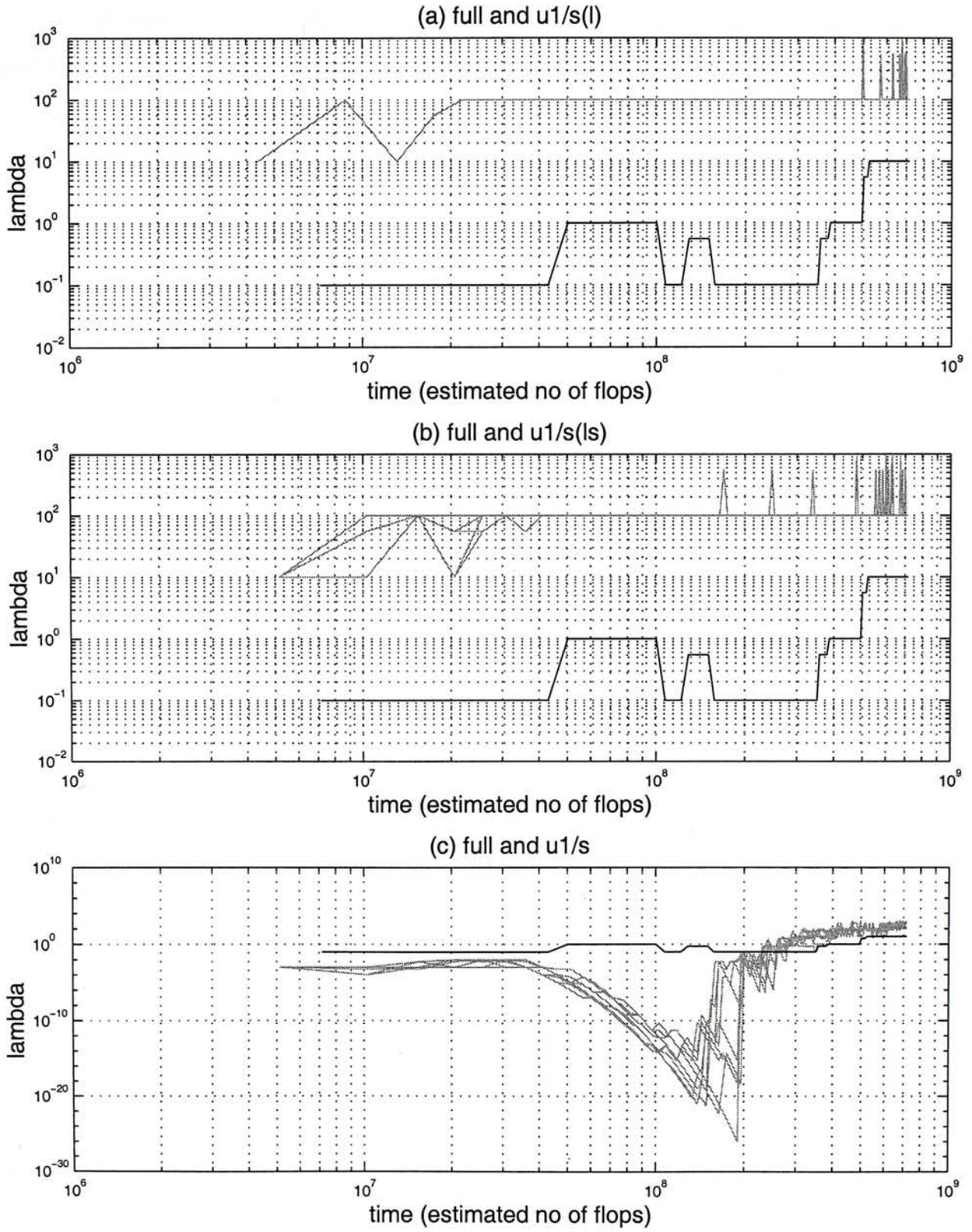


Figure 4-24 Values of λ's during the whole training period. λ of the original method with full Hessian matrix is represented by the dark-colored line. λ's of the single λ, multiple λ's and multiple λ's with line search synchronous updating methods shown in (a), (b) and (c) respectively are represented by the light-colored lines. The one-unit block-diagonal Hessian matrix and the sunspot training data were used in these experiments.

4.4.2 Multiple λ's with line search synchronous updating method

4.4.2.1 Algorithm

As mentioned in the previous section, we devise the multiple λ's with line search synchronous updating method to retain the approximate Gauss-Newton direction. The algorithm of this method is as follows.

Algorithm 4.5 Multiple λ's with line search synchronous updating method

1. $\lambda_1, \lambda_2, \ldots, \lambda_B = 0.001$, $\beta = 10$, $\alpha = 2$, iteration = 0 and finished = false
2. WHILE finished = false
3. FOR i = 1 TO B
4. calculate $\Delta \mathbf{w}_i = -(\mathbf{J}_i^T \mathbf{J}_i + \lambda_i \mathbf{I}_i)^{-1} \mathbf{J}_i^T \mathbf{e}$
5. calculate $E(\mathbf{w}_i + \Delta \mathbf{w}_i)$
6. WHILE $E(\mathbf{w}_i + \Delta \mathbf{w}_i) \geq E(\mathbf{w}_i)$
7. increase λ_i by a factor β
8. calculate $\Delta \mathbf{w}_i$
9. calculate $E(\mathbf{w}_i + \Delta \mathbf{w}_i)$
10. END
11. END
12. $\mu = 1$
13. calculate $E(\mathbf{w} + \mu \Delta \mathbf{w})$ where $\Delta \mathbf{w} = [\Delta \mathbf{w}_1^T \Delta \mathbf{w}_2^T \ldots \Delta \mathbf{w}_B^T]^T$
14. WHILE ($\mu \geq$ minimum_μ) AND ($E(w + \mu \Delta \mathbf{w}) \geq E(\mathbf{w})$)
15. decrease μ by a factor α
16. calculate $E(\mathbf{w} + \mu \Delta \mathbf{w})$
17. END
18. IF $\mu \geq$ minimum_μ
19. update $\mathbf{w}$ ($\mathbf{w} \leftarrow \mathbf{w} + \Delta \mathbf{w}$)
20. decrease $\lambda_1, \lambda_2, \ldots, \lambda_B$ by a factor β
21. iteration $\leftarrow$ iteration + 1
22. END
23. calculate $\mathbf{J}^T \mathbf{e}$
24. IF (iteration > maximum_iteration) OR
(minimum of validation error is reached = true) OR

($\mathbf{J}^T\mathbf{e}$ < minimum_gradient) OR (μ < minimum_μ)

25. finished = true

26. END

27. END

The first part of the algorithm (Steps 2 to 11) is to find λ_i tailored to each decomposed system of updating equations. It is the same as that of the multiple λ's synchronous updating method. After that, we adapt the step size parameter μ to obtain a decrease in the cost function E.

4.4.2.2 Performance

The performance of the multiple λ's with line search synchronous updating method is shown in the last column of the first four parts of Figures 4-19, 4-20 and 4-21. These figures show that the training time performance of the multiple λ's with line search synchronous updating method is better than that of the single λ and multiple λ's synchronous updating methods under different block-diagonal Hessian matrices and training data. The comparisons are summarized in Tables 4-10 and 4-11 in terms of the ratios of their average final training time. Moreover, Figures 4-19 to 4-21 show that the training time performance of the multiple λ's with line search synchronous updating method was slightly worse than that of the original method. On average, the average final training time of the multiple λ's with line search synchronous updating method is about 1.2 times more than that of the original method. The individual comparisons are summarized in Table 4-12 in terms of the ratios of their average final training time.

Training data	Average final training time of the multiple λ's with line search synchronous updating method / Average final training time of the single λ synchronous updating method			
	correlation block-diagonal matrix	one-unit block-diagonal matrix	three-unit block-diagonal matrix	layer block-diagonal matrix
Single sine	0.38	0.56	0.26	0.28
Composite sine	0.84	0.74	0.38	0.71
Sunspot	0.37	0.16	0.22	0.48

Table 4-10 Ratios of the average final training time of the multiple λ's with line search synchronous updating method to that of the single λ synchronous updating method under different training data and block-diagonal Hessian matrices.

Training data	Average final training time of the multiple λ's with line search synchronous updating method / Average final training time of the multiple λ's synchronous updating method			
	correlation block-diagonal matrix	one-unit block-diagonal matrix	three-unit block-diagonal matrix	layer block-diagonal matrix
Single sine	0.39	0.46	0.30	0.22
Composite sine	0.85	0.75	0.36	0.67
Sunspot	0.42	0.17	0.25	0.43

Table 4-11 Ratios of the average final training time of the multiple λ's with line search synchronous updating method to that of the multiple λ's synchronous updating method under different training data and block-diagonal Hessian matrices.

Training data	Average final training time of the multiple λ's with line search synchronous updating method / Average final training time of the original method			
	correlation block-diagonal matrix	one-unit block-diagonal matrix	three-unit block-diagonal matrix	layer block-diagonal matrix
Single sine	0.93	0.96	0.40	0.51
Composite sine	1.91	1.67	0.78	1.49
Sunspot	1.78	0.79	0.99	1.93

Table 4-12 Ratios of the average final training time of the multiple λ's with line search synchronous updating method to that of the original method under different training data and block-diagonal Hessian matrices.

4.4.2.3 Comparison of λ

We examined the values of λ's after applying the step size parameter μ. The values of λ's of the multiple λ's with line search synchronous updating method during the whole training period were recorded when the networks were trained to predict the single sine, composite sine and sunspot data. They are shown in Figures 4-22c to 4-24c respectively. These values are the median values of λ's of 20 trials and are represented by the light colored lines. Moreover, λ of the original method represented by the dark colored line is plotted in each figure for comparison.

These figures show that most λ's of the multiple λ's with line search synchronous updating method were often smaller than that of the original method with full Hessian matrix in the early training period. The observations about the values of λ's were the same if the other block-diagonal Hessian matrices were used instead. Large λ's were not observed when this algorithm was used.

4.5 Comparison between asynchronous and synchronous updating methods

The results of the previous sections show that the asynchronous updating method works best with constraint on the weight change and the synchronous updating method works best with line search strategy. In this section, these two updating methods, namely the asynchronous updating with constraint method and the multiple λ's with line search synchronous updating method, are compared. Their final training time is compared in Section 4.5.1.

The final training time depends on two factors, which are the computation load per complete weight update and convergence speed. The relation between the final training time and these factors is shown in Equation 4.2.

$$\text{final training time} \propto \frac{\text{computation load per complete weight update}}{\text{convergence speed}} \qquad (4.2)$$

The effects of the updating methods on these two factors are compared in Sections 4.5.2 and 4.5.3 respectively.

4.5.1 Final training time

The performance of the asynchronous and synchronous updating methods is plotted together in Figure 4-25. Figures 4-25a and 4-25b show the final training time and the generalization errors respectively. Each figure is divided into three parts. These parts show the performance of the asynchronous, synchronous and original updating methods respectively. Four columns are shown in each of the first two parts. They represent the performance of using the correlation, one-unit, three-unit and layer block-diagonal Hessian matrices respectively. This figure shows the performance of the training methods used to train the networks to predict the single sine data. The corresponding graphs of the performance of the training methods used to train the networks to predict the composite sine and sunspot data are shown in Figures 4-26 and 4-27 respectively.

These figures show that the asynchronous updating method learned faster than the synchronous updating method under different block-diagonal Hessian matrices and training data. The comparisons are summarized in Table 4-13 in terms of the ratios of their average final training time.

Training data	Average final training time of the synchronous updating method / Average final training time of the asynchronous updating method			
	correlation block-diagonal matrix	one-unit block-diagonal matrix	three-unit block-diagonal matrix	layer block-diagonal matrix
Single sine	7.76	8.70	1.12	2.88
Composite sine	3.10	2.21	1.61	1.82
Sunspot	8.93	4.43	5.10	4.55

Table 4-13 Ratios of the average final training time of the synchronous updating method to that of the asynchronous updating method under different training data and block-diagonal Hessian matrices.

4.5.2 Computation load per complete weight update

The computation load per complete weight update is one of the factors of final training time. We used its average value over the whole training process to measure this factor. The average computation loads per complete weight update of the training methods used to train the networks to predict the single sine, composite sine and sunspot data are shown in Figures 4-28, 4-29 and 4-30 respectively. They were measured in terms of number of flops. The placements of the training methods on the x-axes are the same as those shown in Figures 4-25, 4-26 and 4-27.

Table 4-14 summarizes the comparisons between the average computation loads per complete weight update of the synchronous and asynchronous updating methods under different training data and block-diagonal Hessian matrices shown in Figures 4-28, 4-29 and 4-30. The comparisons were made in terms of the ratios of their average values of 20 trials. This table shows that the average computation load per complete weight update of the synchronous updating method was slightly larger than that of the synchronous updating method in most of the experiments and was about 1.04 times larger on average.

Training data	Average computation load per complete weight update of the synchronous updating method / Average computation load per complete weight update of the asynchronous updating method			
	correlation block-diagonal matrix	one-unit block-diagonal matrix	three-unit block-diagonal matrix	layer block-diagonal matrix
Single sine	0.97	1.05	1.01	1.04
Composite sine	1.10	1.06	1.04	1.05
Sunspot	1.05	1.01	1.02	1.03

Table 4-14 Ratios of the average computation load per complete weight update of the synchronous updating method to that of the asynchronous updating method under different training data and block-diagonal Hessian matrices. Average value of 20 trials was used in calculating these ratios.

4.5.3 Convergence speed

The convergence speed, which is inversely proportional to the number of cycles of complete weight update of the whole training process, is another factor of final training time. To compare the convergence speed of the asynchronous and synchronous updating methods, we plotted the numbers of cycles of complete weight update of the whole training process of the training methods used to train the networks to predict the single sine, composite sine and sunspot data on Figures 4-31, 4-32 and 4-33 respectively. The placements of the training methods on the x-axes are the same as those shown in Figures 4-25, 4-26 and 4-27.

These figures show that the asynchronous updating method converged faster than the synchronous updating method and the original method with full Hessian matrix. Since the factor of computation loads per complete weight update of the asynchronous and synchronous updating methods are approximately the same, the factor of convergence speed becomes the determinant of the final training time. So, the plots of the inverse of convergence speed and final training time are very similar. We can verify this by comparing Figures 4-31, 4-32 and 4-33 with Figures 4-25, 4-26 and 4-27.

The fast convergence speed of the asynchronous updating method can be explained as follows. Asynchronous updating method finds an optimal λ for weights of each block. In contrast, the original method finds one optimal λ for all weights. This λ probably is not the best for individual weight. The synchronous updating

with line search method finds an optimal λ for weights of each block at first. But, it limits the learning step size later because of the deviation between the full and block-diagonal Hessian matrices.

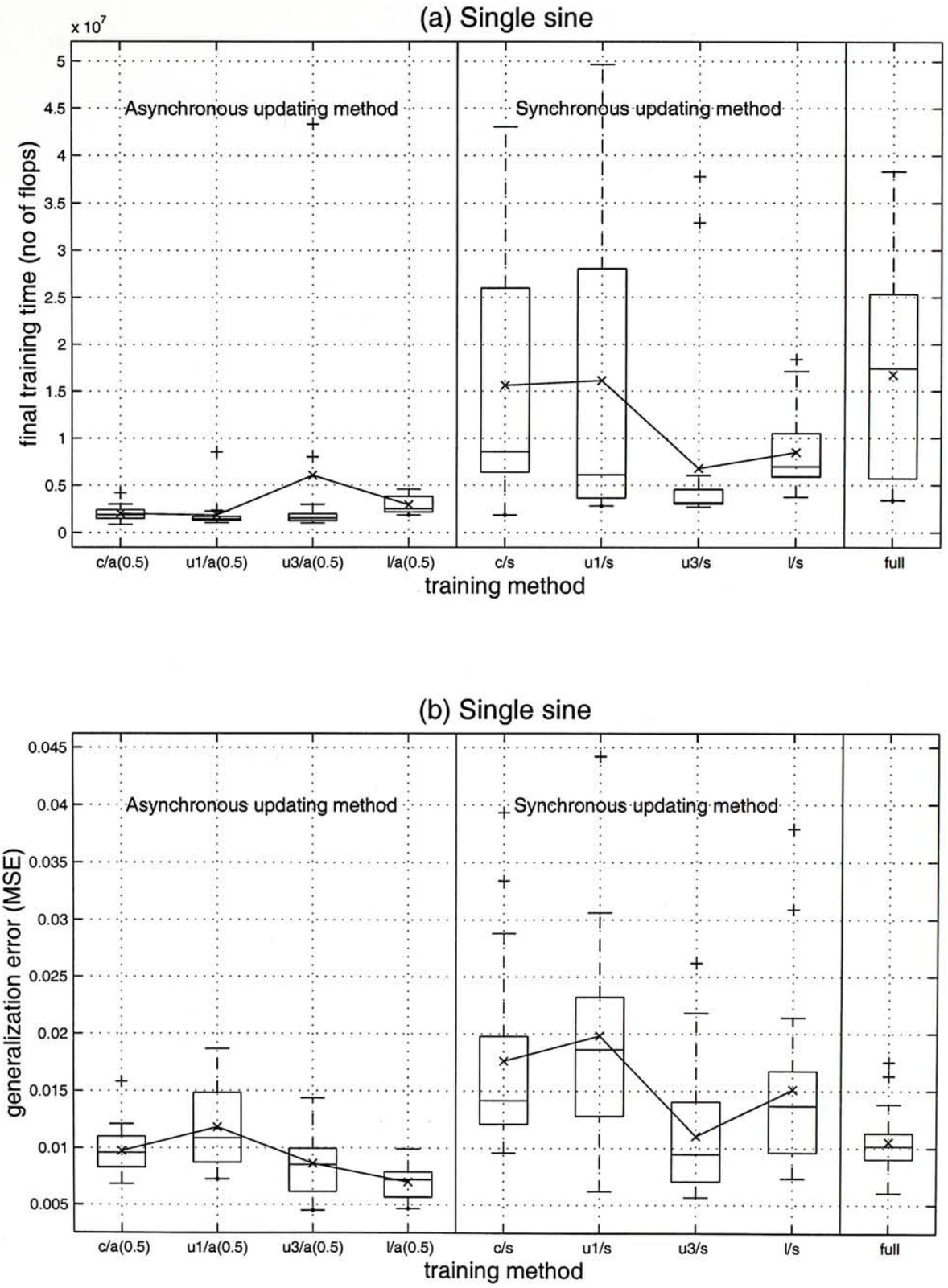


Figure 4-25 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axes are explained in Tables 4-4 and 4-5. The single sine training data was used in these experiments.

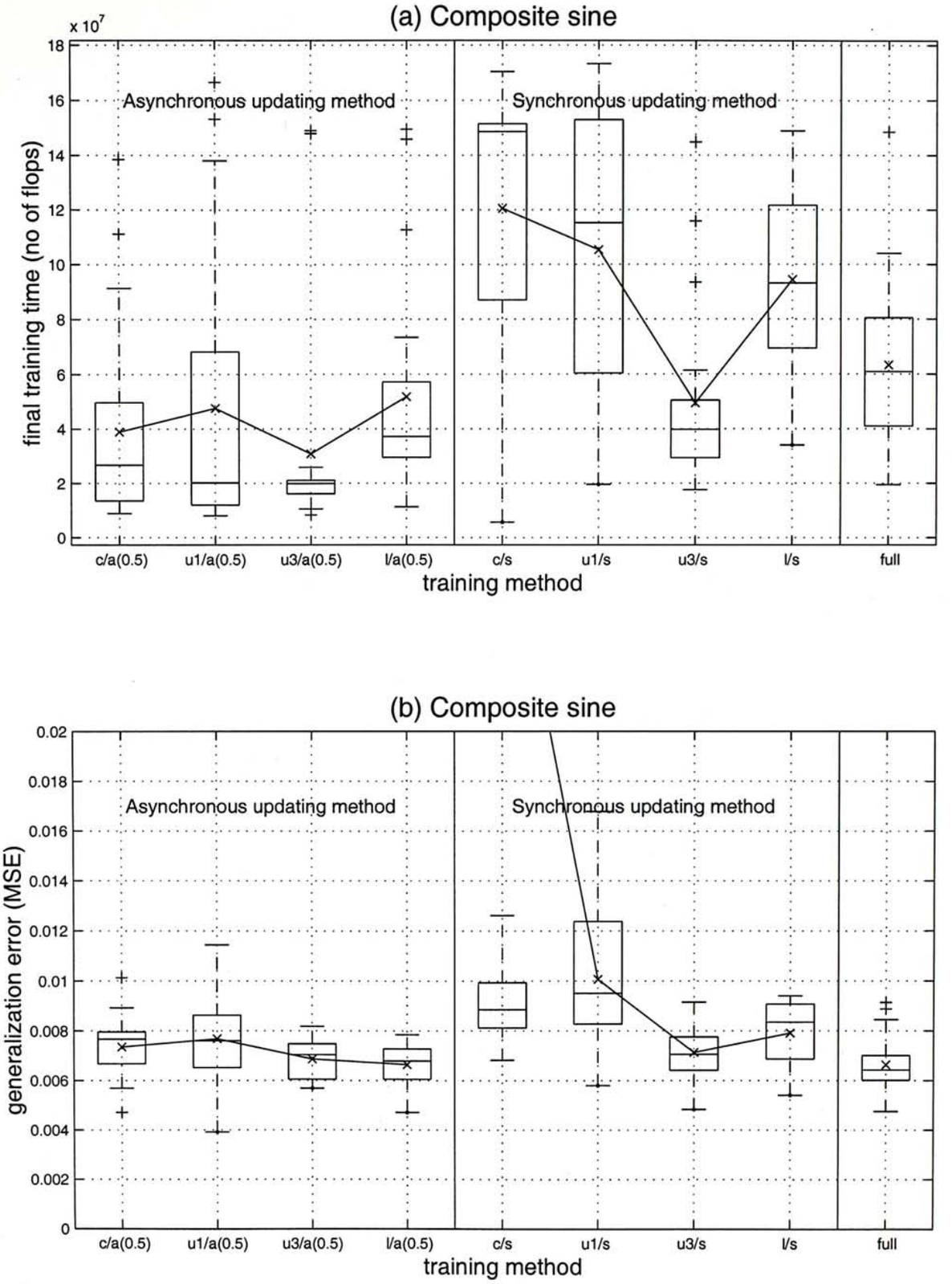


Figure 4-26 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axes are explained in Tables 4-4 and 4-5. The composite sine training data was used in these experiments.

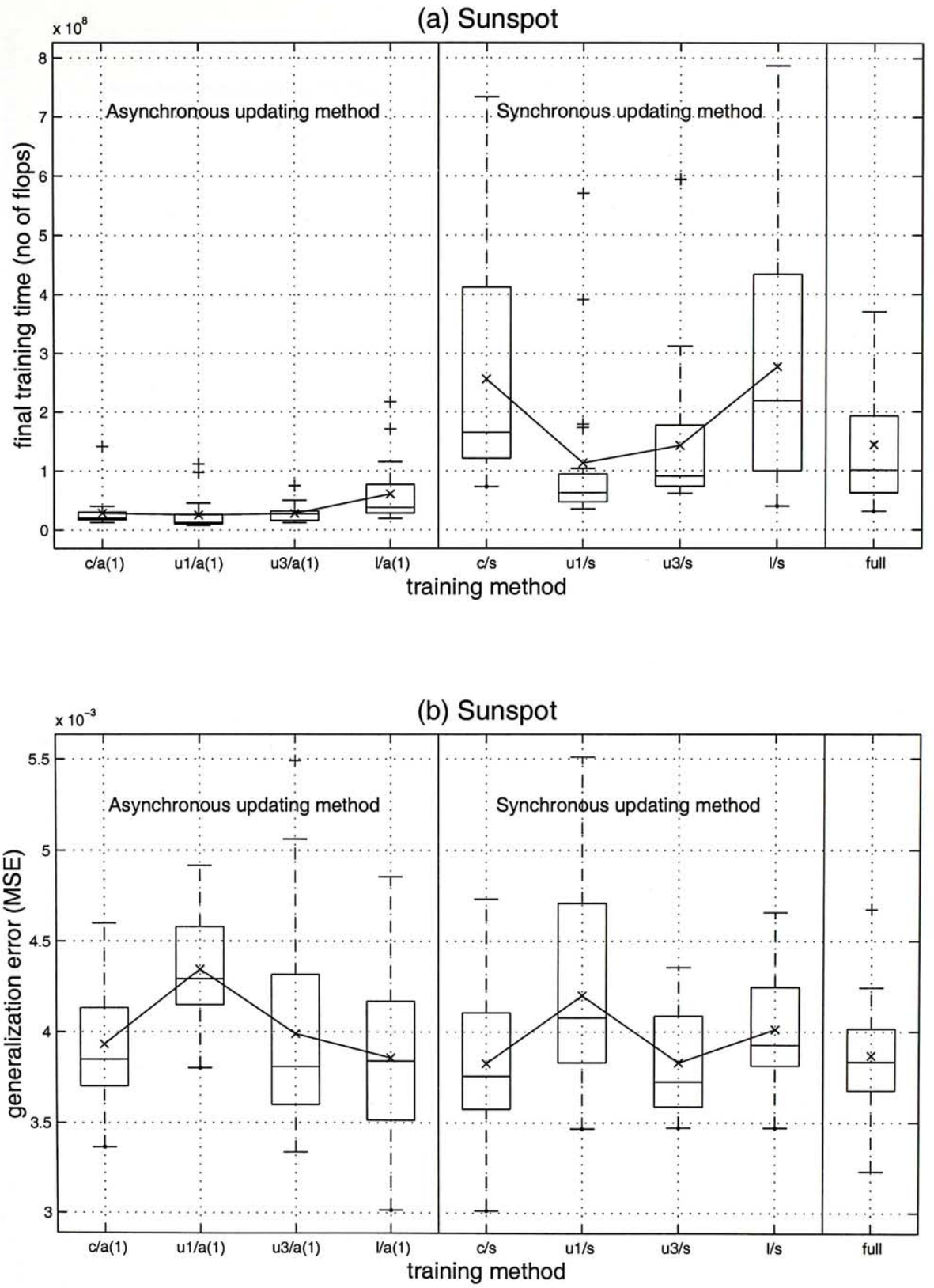


Figure 4-27 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axes are explained in Tables 4-4 and 4-5. The sunspot training data was used in these experiments.

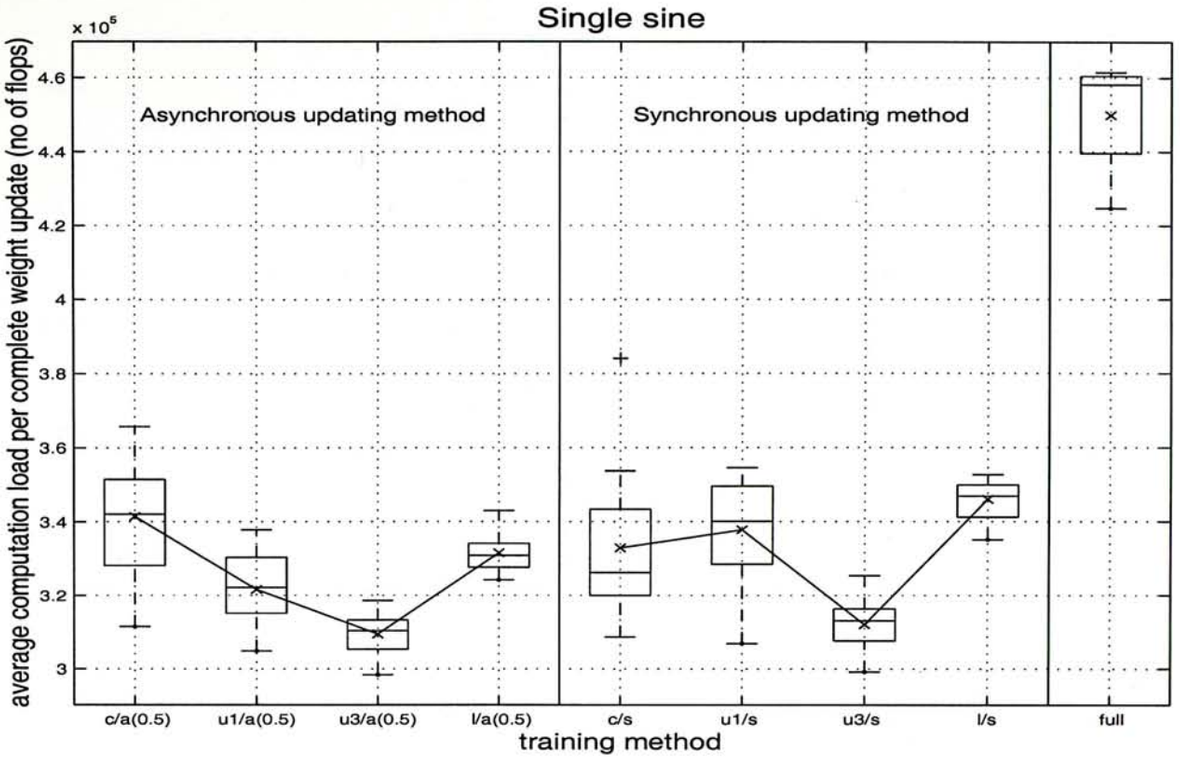


Figure 4-28 Average computation load per complete weight update measured in terms of number of flops. The x-axis shows the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axis are explained in Tables 4-4 and 4-5. The single sine training data was used in these experiments.

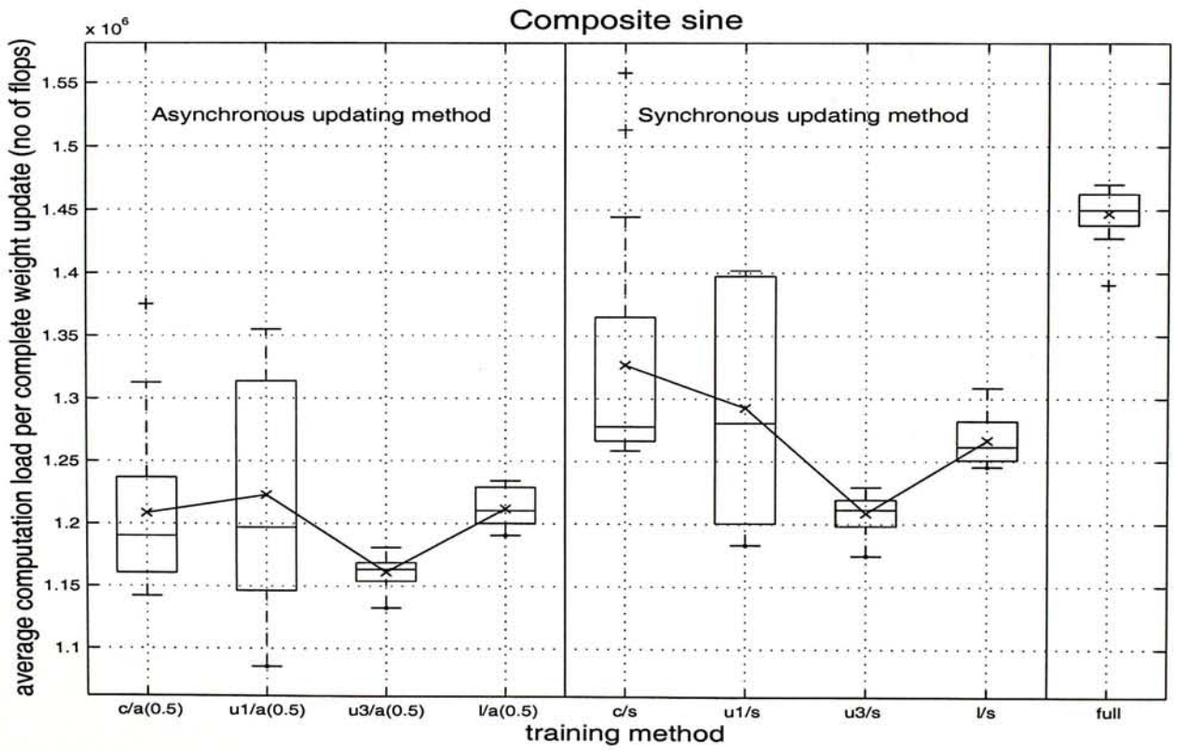


Figure 4-29 Average computation load per complete weight update measured in terms of number of flops. The x-axis shows the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axis are explained in Tables 4-4 and 4-5. The composite sine training data was used in these experiments.

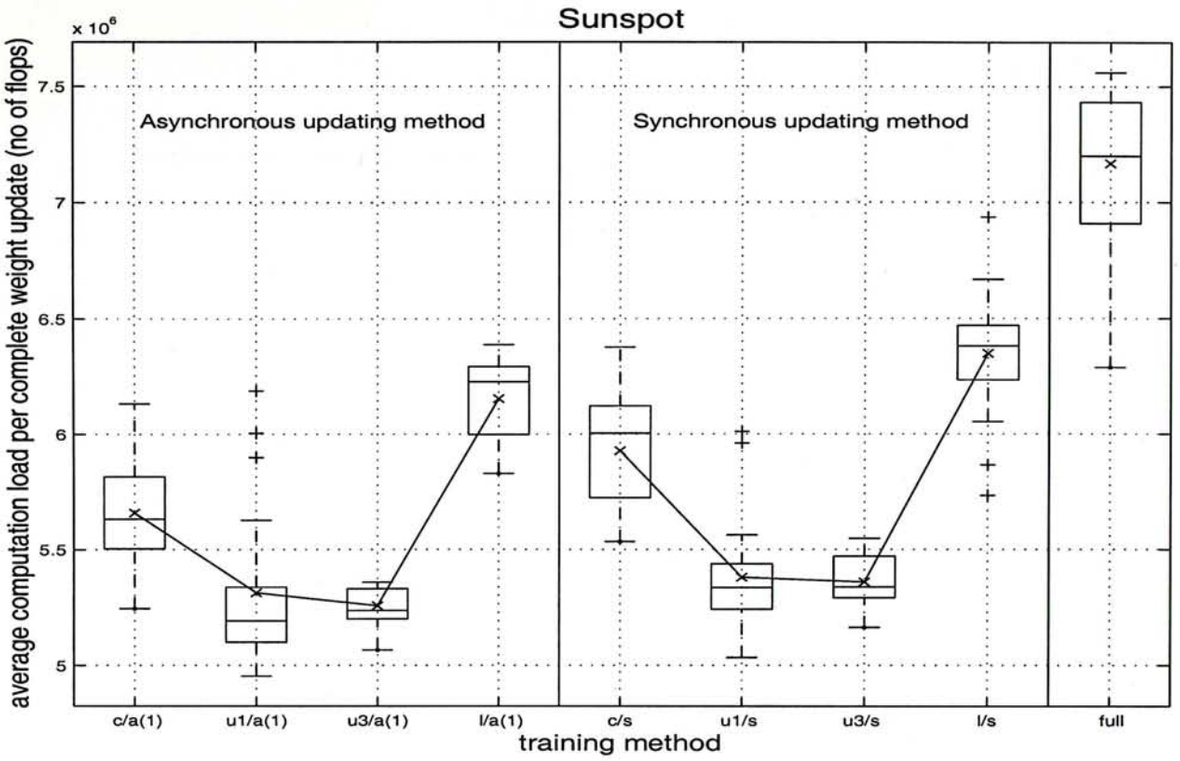


Figure 4-30 Average computation load per complete weight update measured in terms of number of flops. The x-axis shows the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axis are explained in Tables 4-4 and 4-5. The sunspot training data was used in these experiments.

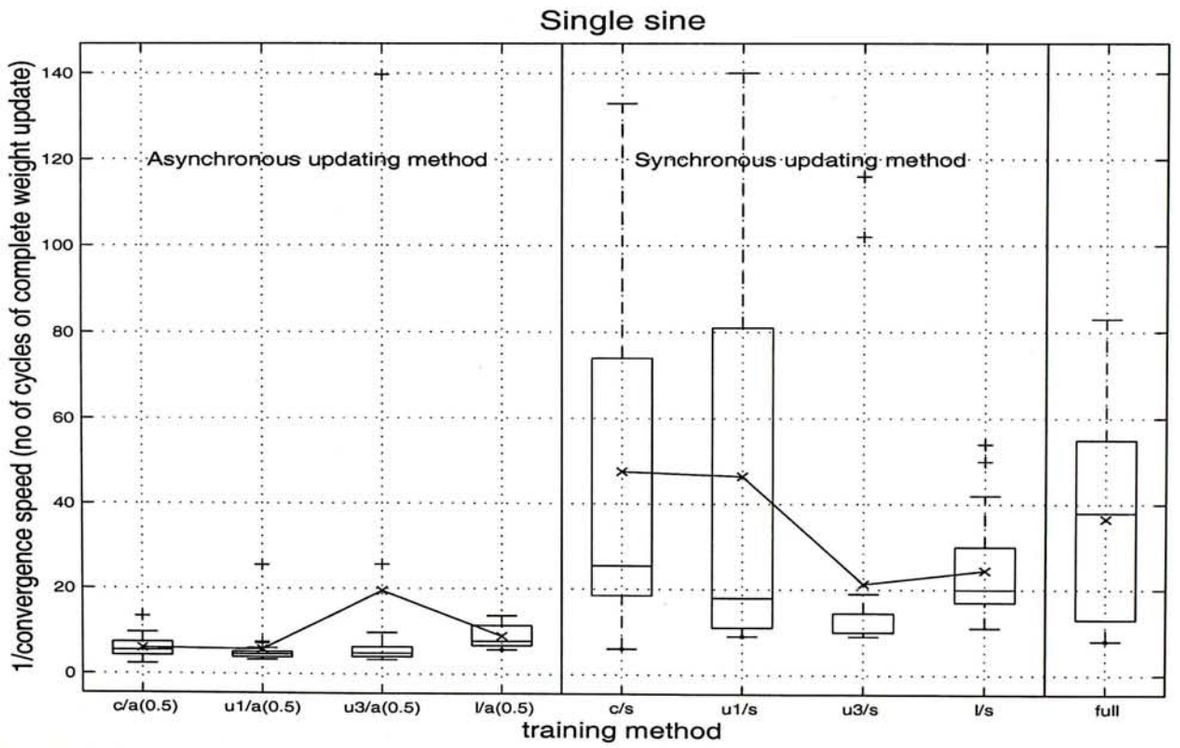


Figure 4-31 Inverse of convergence speed measured in terms of number of cycles of complete weight update. The x-axis shows the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axis are explained in Tables 4-4 and 4-5. The single sine training data was used in these experiments.

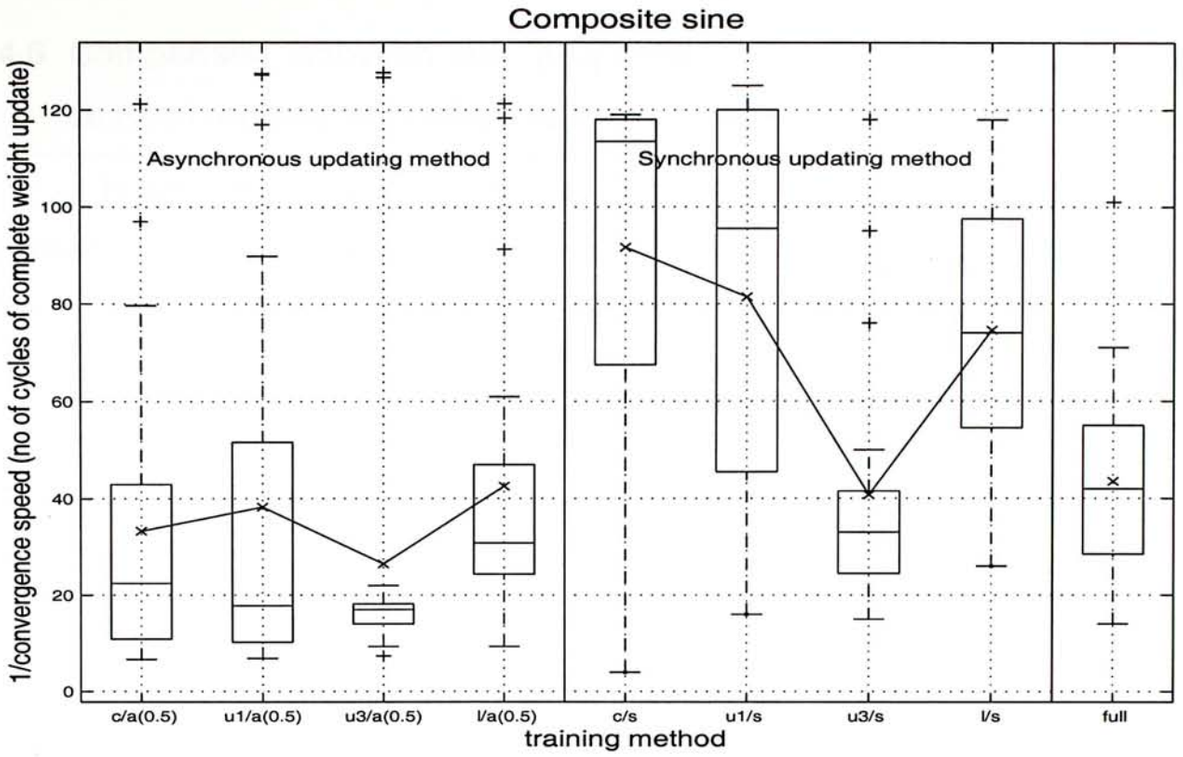


Figure 4-32 Inverse of convergence speed measured in terms of number of cycles of complete weight update. The x-axis shows the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axis are explained in Tables 4-4 and 4-5. The composite sine training data was used in these experiments.

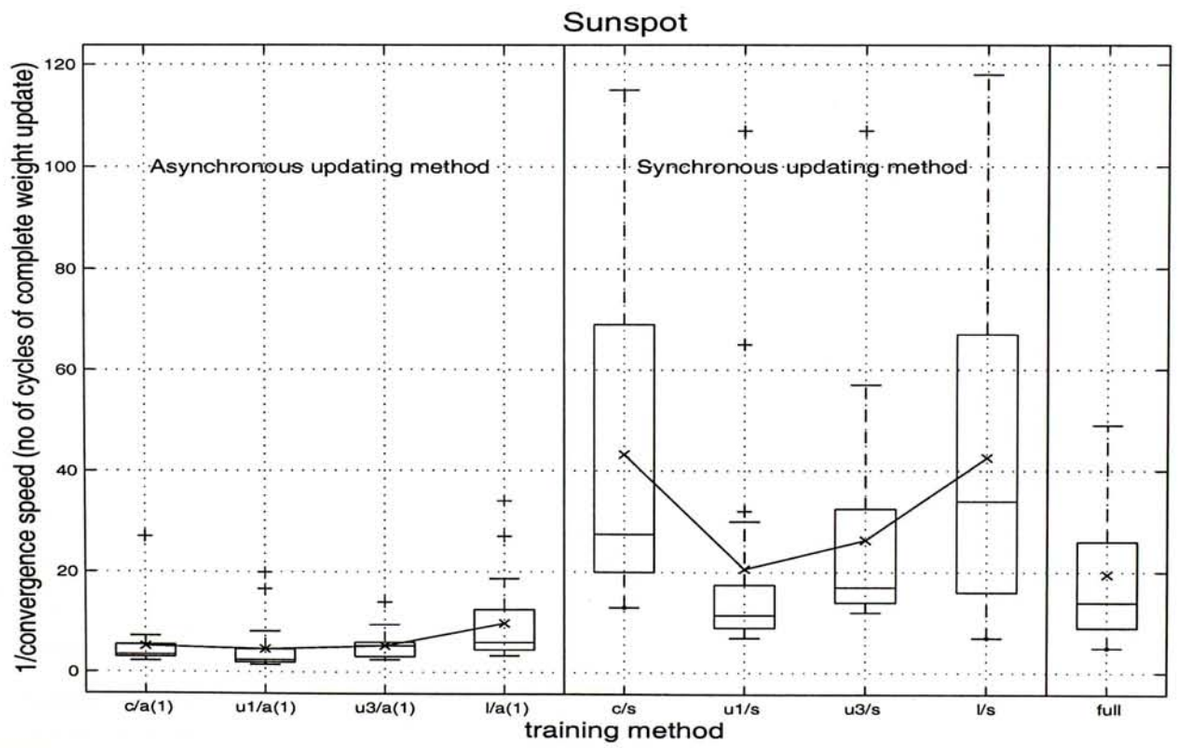


Figure 4-33 Inverse of convergence speed measured in terms of number of cycles of complete weight update. The x-axis shows the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axis are explained in Tables 4-4 and 4-5. The sunspot training data was used in these experiments.

4.6 Comparison between our proposed methods and the gradient descent method with adaptive learning rate and momentum

In this section, we compare our proposed methods with the traditional method: the gradient descent method with adaptive learning rate and momentum. The learning rate and momentum of the gradient descent method are adapted using the strategy described in [Chan87].

In the comparison, our proposed methods and the gradient descent method were used to train the layered fully recurrent network described in Section 1.2.3 to predict the single sine, composite sine and sunspot data described in Sections 3.2.1 to 3.2.3 respectively. The results of the training time and generalization performance are shown in Tables 4-15 and 4-16 respectively.

We found from Table 4-15 that the training time performance of the asynchronous updating method is better than the gradient descent method in all cases. The asynchronous updating method is from 2 to 15 times faster than the gradient descent method. The training time performance of the synchronous updating method is better than or close to that of the gradient descent method. The synchronous updating method is from 1 to 3.5 times faster than the gradient descent method.

Training data	Average final training time of our proposed methods / Average final training time of the gradient descent method							
	Asynchronous updating method				Synchronous updating method			
	correlation block-diagonal matrix	one-unit block-diagonal matrix	three-unit block-diagonal matrix	layer block-diagonal matrix	correlation block-diagonal matrix	one-unit block-diagonal matrix	three-unit block-diagonal matrix	layer block-diagonal matrix
Single sine	0.086	0.079	0.26	0.13	0.66	0.69	0.29	0.36
Composite sine	0.35	0.42	0.27	0.46	1.07	0.94	0.44	0.84
Sunspot	0.071	0.064	0.070	0.15	0.64	0.28	0.36	0.69

Table 4-15 Ratios of the average final training time of our proposed methods to that of the gradient descent method with adaptive learning rate and momentum under different training data and block-diagonal Hessian matrices.

Table 4-16 shows that the generalization errors of our proposed methods and the gradient descent method are similar in most cases. The high average generalization error of the synchronous updating method when the correlation block-diagonal Hessian matrix is used is due to a few outliers which can be observed from Figure 4-26b.

Training data	Average generalization errors of our proposed methods / Average generalization error of the gradient descent method							
	Asynchronous updating method				Synchronous updating method			
	correlation block-diagonal matrix	one-unit block-diagonal matrix	three-unit block-diagonal matrix	layer block-diagonal matrix	correlation block-diagonal matrix	one-unit block-diagonal matrix	three-unit block-diagonal matrix	layer block-diagonal matrix
Single sine	0.69	0.83	0.61	0.49	1.24	1.39	0.78	1.06
Composite sine	0.93	0.98	0.87	0.84	3.79	1.28	0.91	1.01
Sunspot	1.05	1.16	1.07	1.03	1.03	1.13	1.03	1.08

Table 4-16 Ratios of the average generalization errors of our proposed methods to that of the gradient descent method with adaptive learning rate and momentum under different training data and block-diagonal Hessian matrices.

Chapter 5 Number and sizes of the blocks

5.1 Introduction

As discussed in Chapter 2, we need to consider two factors when the technique of block-diagonal Hessian approximation is applied. These two factors are

i. the choice of weight updating methods and
ii. the choice of block-diagonal Hessian matrices

The first factor has been studied in Chapter 4. In this and the next chapters, the second factor will be studied.

The performance of our proposed method depends on the block-diagonal Hessian matrix and this matrix depends on the following two factors.

i. Number and sizes of the blocks
ii. Weight-grouping method

In this chapter, the effect of the first factor on the performance of our proposed method will be studied. The effect of the second factor will be studied in the next chapter.

The subsequent sections are organized as follows. In Section 5.2, the effect of number and sizes of the blocks on the final training time and generalization error is studied. In Sections 5.3 and 5.4, we will study the effects of number and sizes of the blocks on the computation load per complete weight update and convergence speed respectively, which are the factors of the final training time.

5.2 Performance

In this section, the effect of number and sizes of the blocks on the final training time and generalization error is studied.

5.2.1 Method of study

To simplify the study, we restricted ourselves to block-diagonal Hessian matrix whose block sizes are approximately equal. That is,

$$q_i \approx \frac{Wt}{B}, i = 1, 2, \ldots, B \qquad (5.1)$$

where q_i is the block size of the ith block

Wt is the number of network weights

B is the number of blocks

As the number of blocks varied, the block sizes varied according to Equation 5.1. In each variation in the number of blocks, the performance was measured and plotted on a graph to show the trend.

As mentioned in the introduction, the performance of our proposed method depends on the weight-grouping method. To eliminate the effect of weight-grouping method, we used the same weight-grouping method in all experiments. In our experiments, we chose the sub-network weight-grouping method, which enabled us to vary the number of blocks easily.

The Hessian matrices that we chose in the experiments are shown in Table 5-1. These matrices satisfy our restrictions described above. Their numbers of blocks are summarized in Table 5-1. These Hessian matrices would be used with the asynchronous updating method described in Section 4.3 and the synchronous updating method described in Section 4.4.2. These training methods were used to train the layered fully recurrent network described in Section 1.2.3 to predict the single sine, composite sine and sunspot data described in Sections 3.2.1 to 3.2.3 respectively.

Number of hidden units	Number of blocks				
	one-unit block-diagonal Hessian matrix	two-unit block-diagonal Hessian matrix	three-unit block-diagonal Hessian matrix	five-unit block diagonal Hessian matrix	full Hessian matrix
6†	7	4	3	--	1
9‡	10	6	4	3	1

† Networks with six hidden units were used to predict the single and composite sine data

‡ Networks with nine hidden units were used to predict the sunspot data

Table 5-1 Numbers of blocks of Hessian matrices under different numbers of hidden units.

5.2.2 Trend of performance

The trend of performance, which is a function of number of blocks, is shown in Figure 5-1. Figures 5-1a and 5-1b show the final training time and the generalization errors respectively. Each column shows the performance of different training methods. These training methods are different from each other in updating methods and/or block-diagonal Hessian matrices. Each figure is divided into two parts. The left and right parts of each figure show the performance of using the asynchronous and synchronous updating methods respectively. In each part of the figures, the columns show the performance of using the one-unit block-diagonal, two-unit block-diagonal, three-unit block-diagonal and full Hessian matrices. They are arranged in descending order of their numbers of blocks. This figure shows the performance of the training methods used to train the networks to predict the single sine data. The corresponding graphs of the performance of the training methods used to train the networks to predict the composite sine and sunspot data are shown in Figures 5-2 and 5-3 respectively.

5.2.2.1 Asynchronous updating method

The case of using the asynchronous updating method shown in the left halves of Figures 5-1, 5-2 and 5-3 is first discussed. As the number of blocks increased, the average final training time first decreased and then increased. However, in the experiments of predicting the single sine data, the average final training time kept on decreasing.

The performance of using the two-unit block-diagonal Hessian matrix was the best in our experiments. The reasons are as follows. First, the average final training time might start to increase when the number of blocks became large. Although using a block-diagonal Hessian matrix with large number of blocks might further reduce the training time but the reduction was small. Second, the generalization error was slightly larger when a block-diagonal Hessian matrix with large number of blocks was used. This is illustrated in Figures 5.1b, 5.2b and 5.3b. So, using a block-diagonal Hessian matrix with large number of blocks was not suggested.

5.2.2.2 Synchronous updating method

We then discuss the case of using the synchronous updating method shown in the right halves of Figures 5-1, 5-2 and 5-3. As the number of blocks increased, the average final training time first decreased and then increased

The performance of using the two-unit block-diagonal Hessian matrix was the best in our experiments. However, the reduction in training time was not as drastic as that in the asynchronous cases in most of the experiments. The average final training time of the synchronous updating method started to increase when the number of blocks became large. Moreover, the generalization error was slightly larger when a block-diagonal Hessian matrix with large number of blocks was used. So, using a block-diagonal Hessian matrix with large number of blocks was not suggested. It is suggested that the degradation of the performance with the increase of the number of blocks is caused by the increasing of the Hessian approximation error.

We analyzed the trend of final training time by studying its factors. These factors are the computation load per complete weight update and the convergence speed. They are described in Sections 5.3 and 5.4 respectively.

5.3 Computation load per complete weight update

The computation load per complete weight update is one of the factors of final training time. We used its average value over the whole training process to measure this factor. The average computation loads per complete weight update of the training methods used to train the networks to predict the single sine, composite sine and sunspot data are shown in Figures 5-4, 5-5 and 5-6 respectively. The placements of the training methods on the x-axes are the same as those shown in Figures 5-1, 5-2 and 5-3.

These figures show that as the number of blocks increased, the average computation load per complete weight update first decreased and then increased. This indicated that using a block-diagonal Hessian matrix with a large number of blocks did not have the advantage of lower computation load per complete weight update and was not suggested.

The trend of the average computation load per complete weight update can be explained as follows. As the number of blocks increases, both the computation loads of calculating Hessian matrix and solving systems of updating equations decrease. On the other hand, the number of cost function calculations increases.

5.4 Convergence speed

5.4.1 Trend of inverse of convergence speed

The convergence speed, which is inversely proportional to the number of cycles of complete weight update of the whole training process, is another factor of final training time. To study its trend, we plotted the numbers of cycles of complete weight update of the training methods used to train the networks to predict the single sine, composite sine and sunspot data on Figures 5-7, 5-8 and 5-9 respectively. The placements of the training methods on the x-axes are the same as those shown in Figures 5-1, 5-2 and 5-3.

The trends shown in these figures are very similar to those shown in Figures 5-1, 5-2 and 5-3 respectively. This shows that the convergence speed is a crucial factor in determining the final training time.

5.4.2 Factors affecting the convergence speed

There are two factors affecting the convergence speed. They are described as follows.

i. On the one hand, as the number of blocks increases, the number of non-zero Hessian elements decreases and the accuracy of second-order information decreases. Hence the convergence speed is expected to decrease.

ii. On the other hand, as the number of blocks increases, the number of λ's increases. Using λ's tailored to different weights can increase the convergence speed.

These two factors have opposite effects on the convergence speed. From the results of our experiments, we suggested that the effect of the second factor dominated when the number of blocks was small. The effect of the first factor dominated when the number of blocks was large.

Label	Descriptions		
	Hessian matrix	updating method	maximum allowed weight change
u1/s	one-unit block-diagonal	synchronous	N/A
u1/a(0.5)		asynchronous	0.5
u1/a(1)			1
u2/s	two-unit block-diagonal	synchronous	N/A
u2/a(0.5)		asynchronous	0.5
u2/a(1)			1
u3/s	three-unit block-diagonal	synchronous	N/A
u3/a(0.5)		asynchronous	0.5
u3/a(1)			1
u5/s	five-unit block-diagonal	synchronous	N/A
u5/a(1)		asynchronous	1
full	full	N/A	N/A

Table 5-2 Descriptions of labels shown in Figures 5-1 to 5-9.

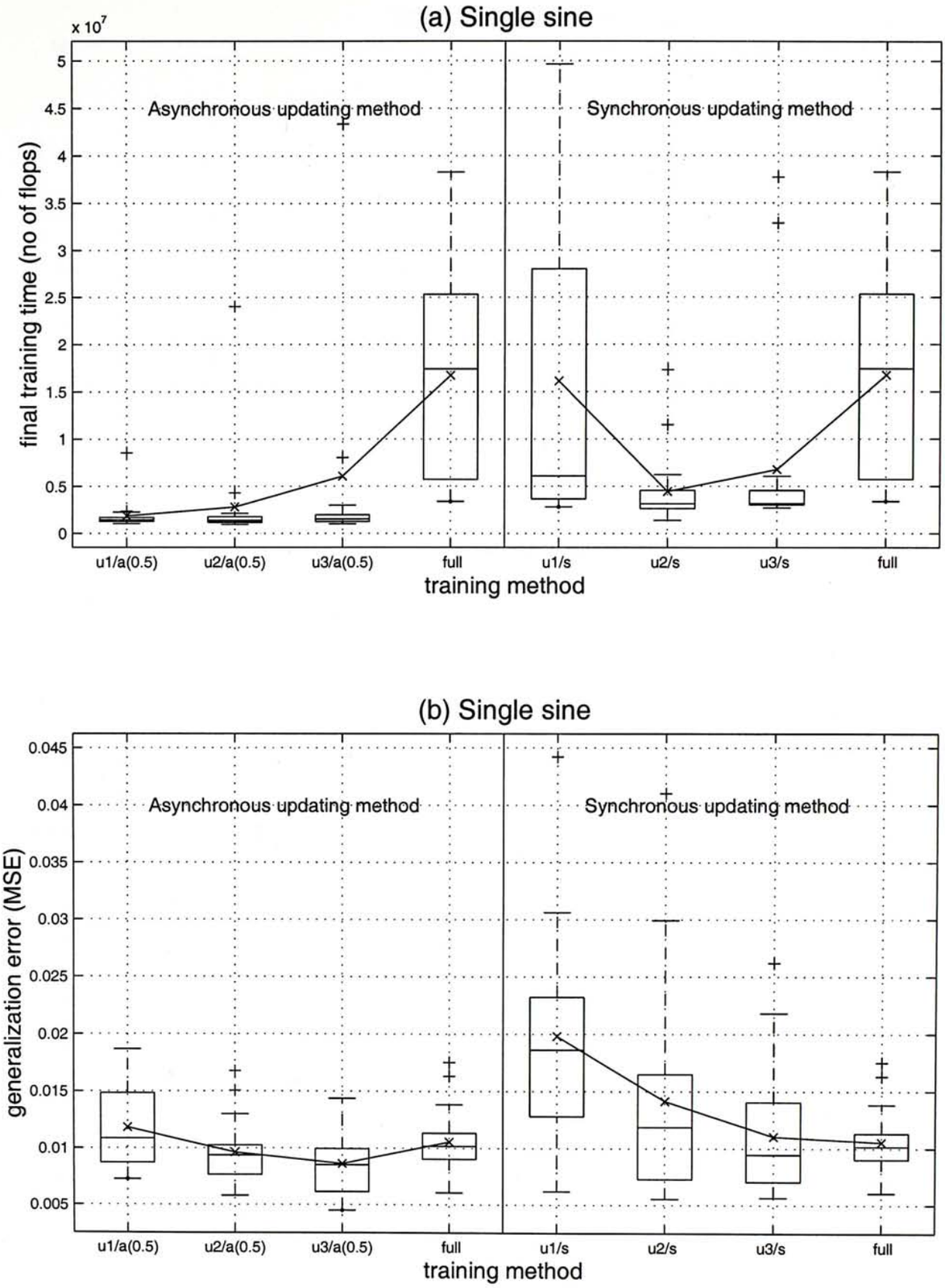


Figure 5-1 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 5-2. The single sine training data was used in these experiments.

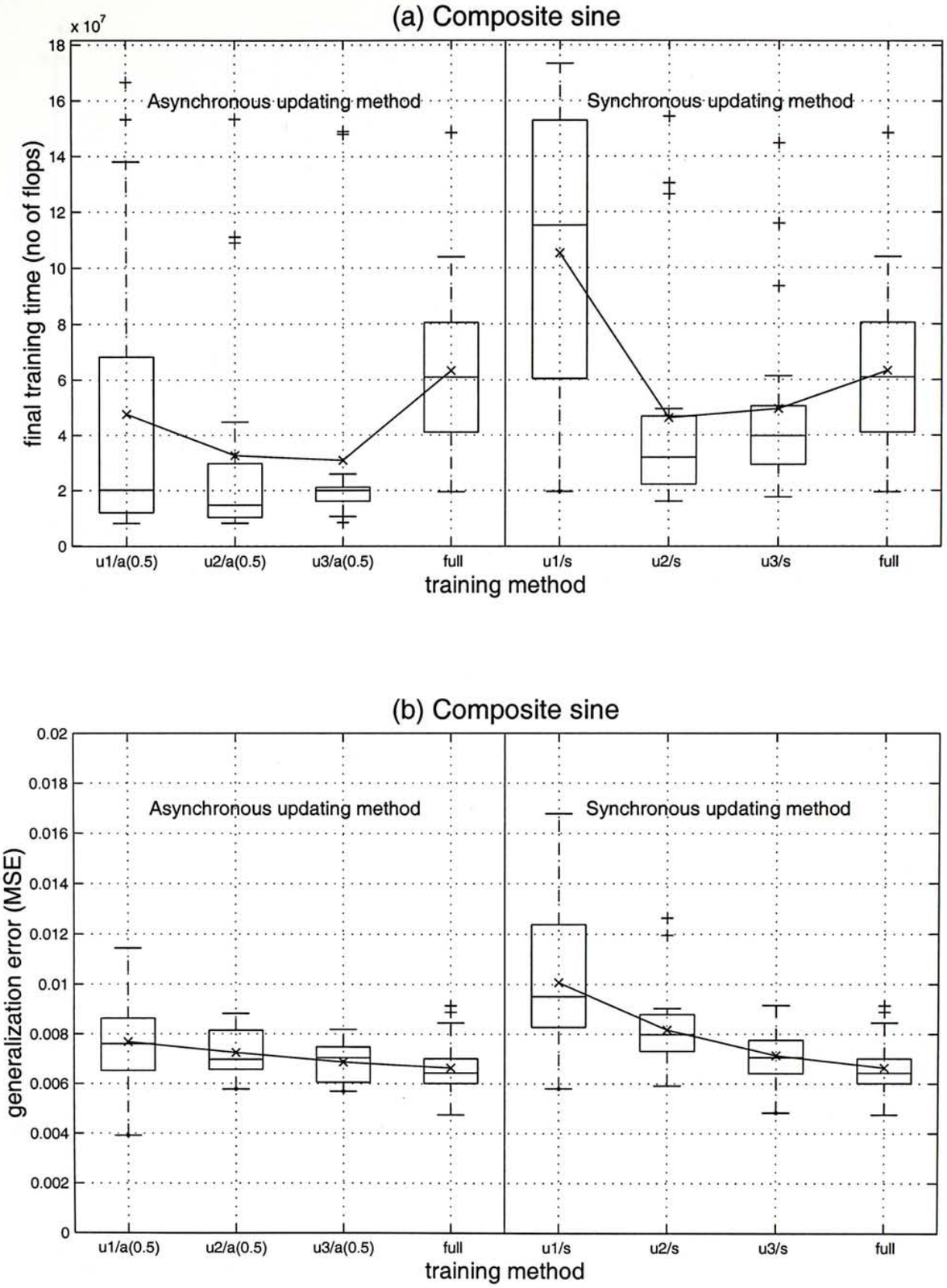


Figure 5-2 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 5-2. The composite sine training data was used in these experiments.

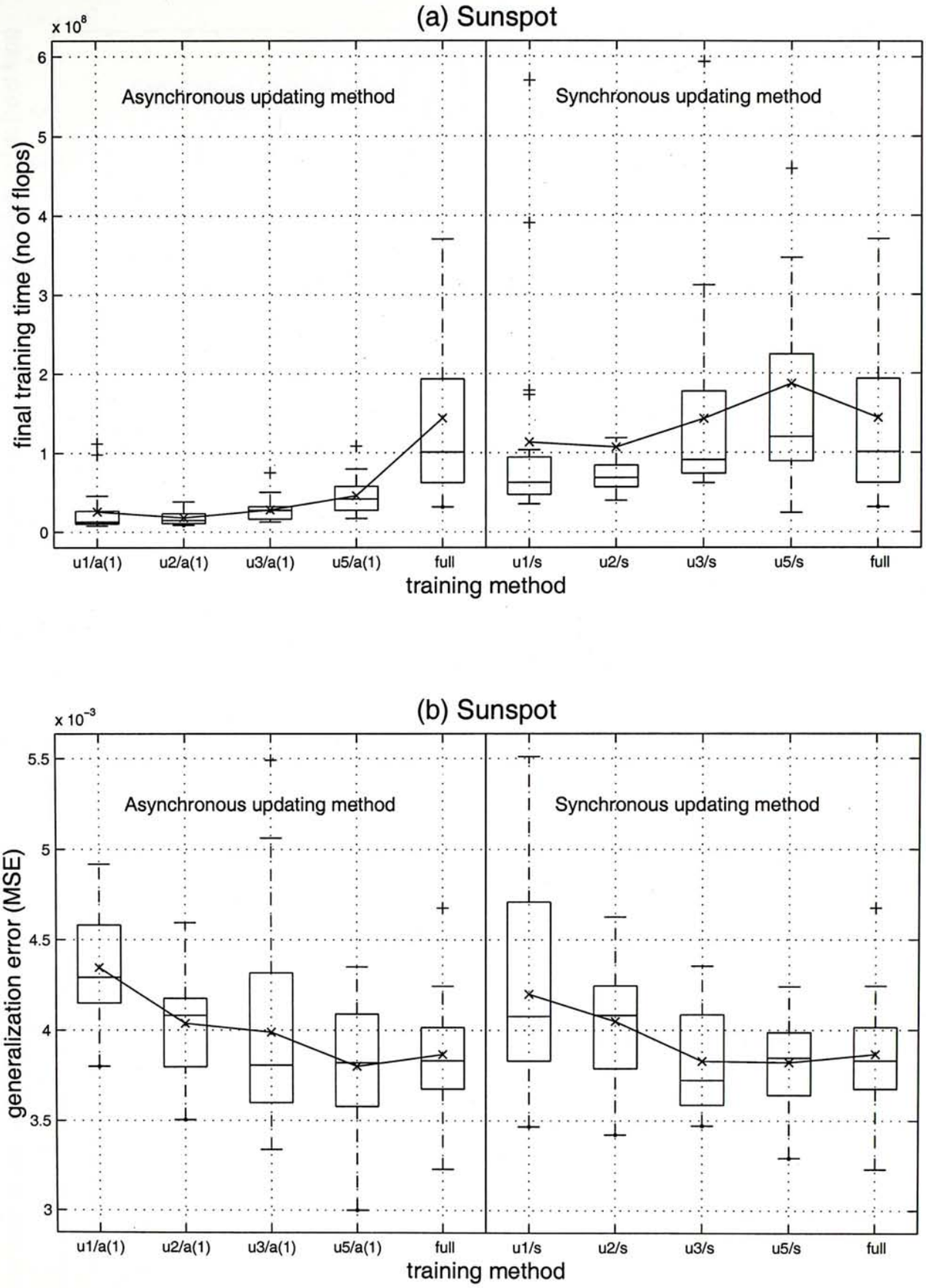


Figure 5-3 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 5-2. The sunspot training data was used in these experiments.

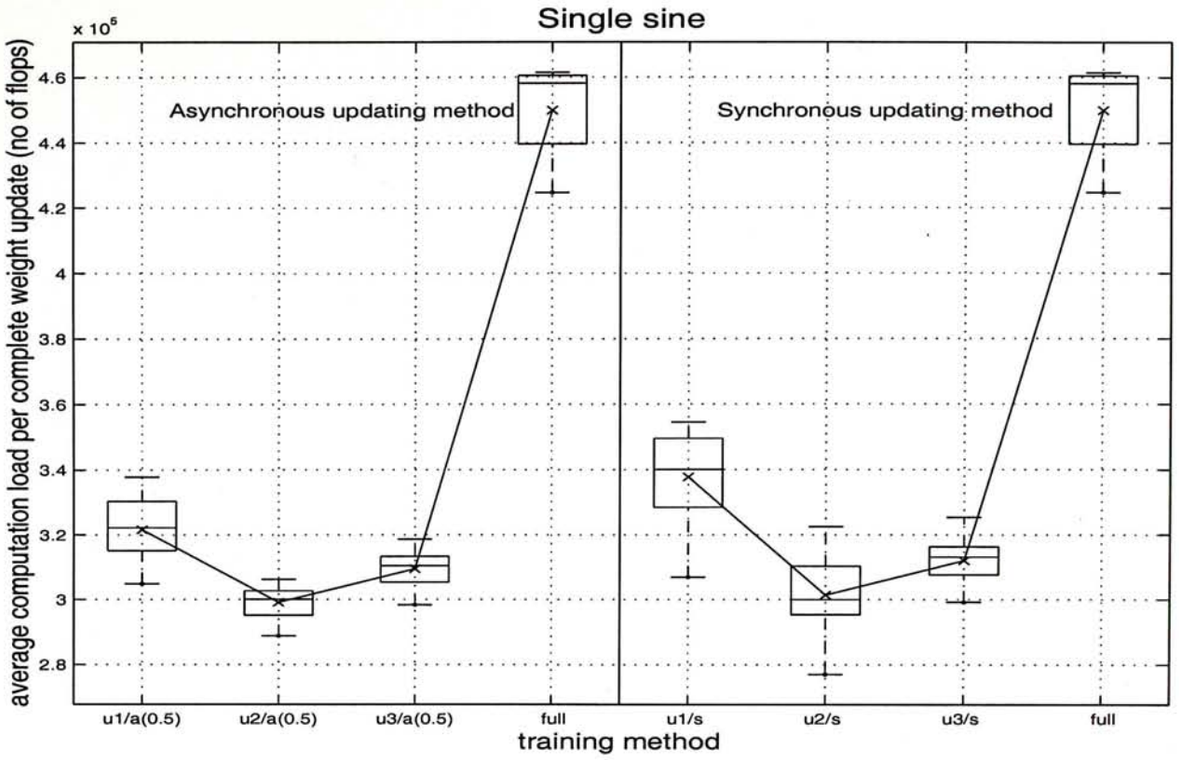


Figure 5-4 Average computation load per complete weight update measured in terms of number of flops. The x-axis shows the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axis are explained in Table 5-2. The single sine training data was used in these experiments.

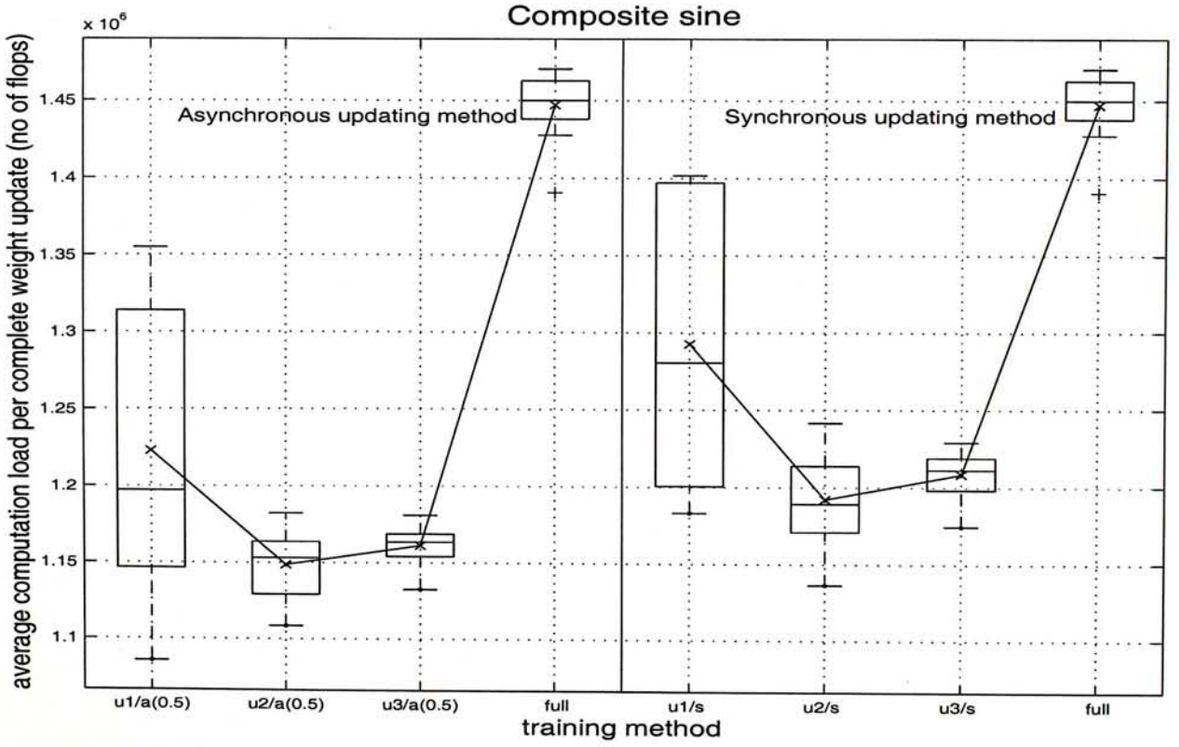


Figure 5-5 Average computation load per complete weight update measured in terms of number of flops. The x-axis shows the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axis are explained in Table 5-2. The composite sine training data was used in these experiments.

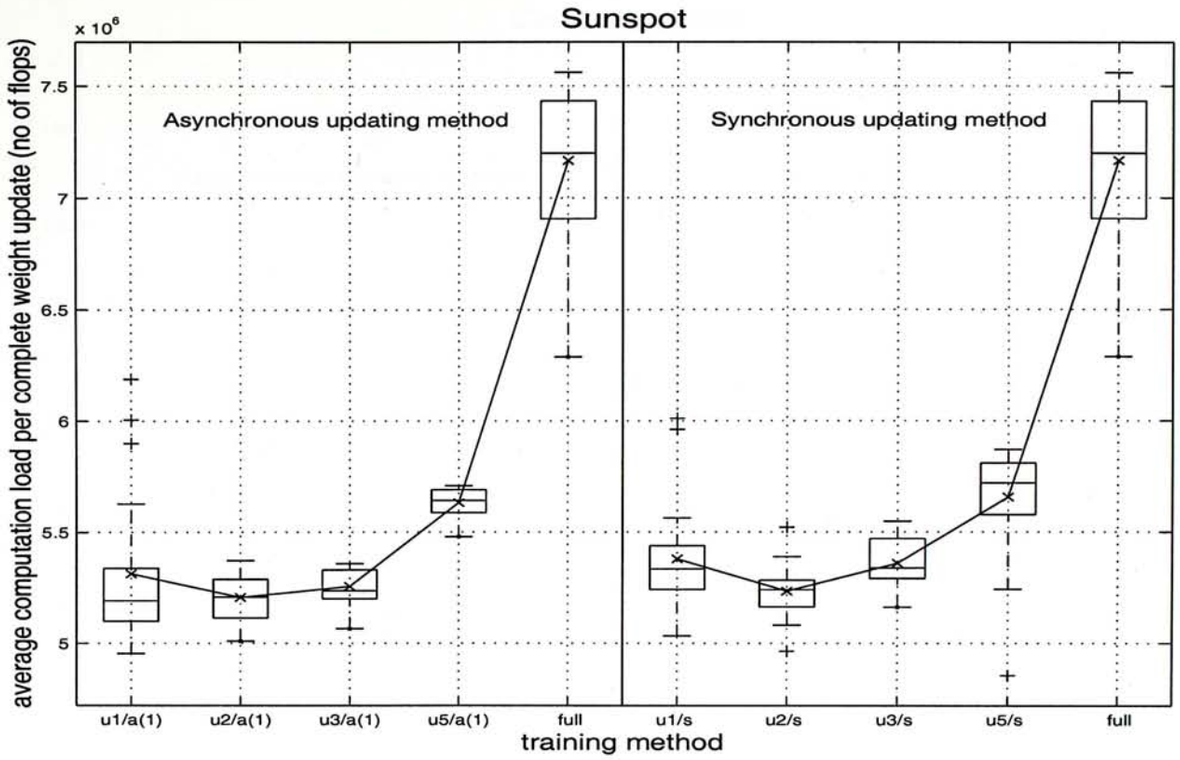


Figure 5-6 Average computation load per complete weight update measured in terms of number of flops. The x-axis shows the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axis are explained in Table 5-2. The sunspot training data was used in these experiments.

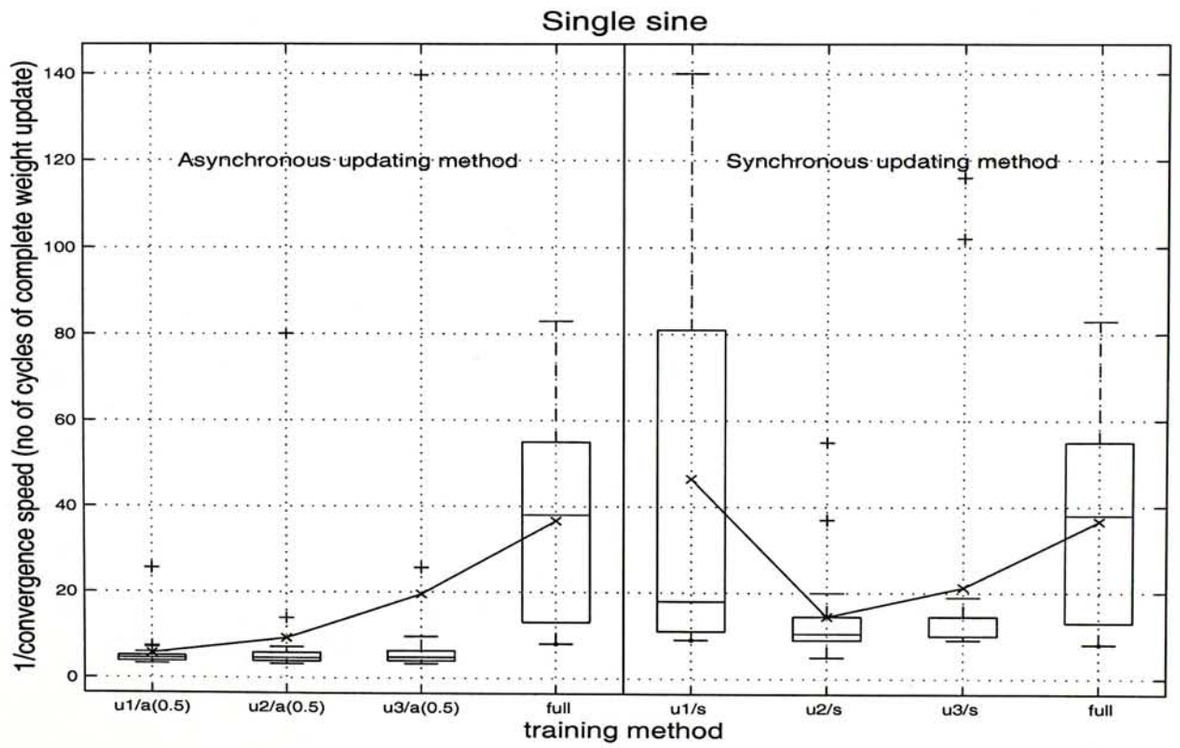


Figure 5-7 Inverse of convergence speed measured in terms of number of cycles of complete weight update. The x-axis shows the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axis are explained in Table 5-2. The single sine training data was used in these experiments.

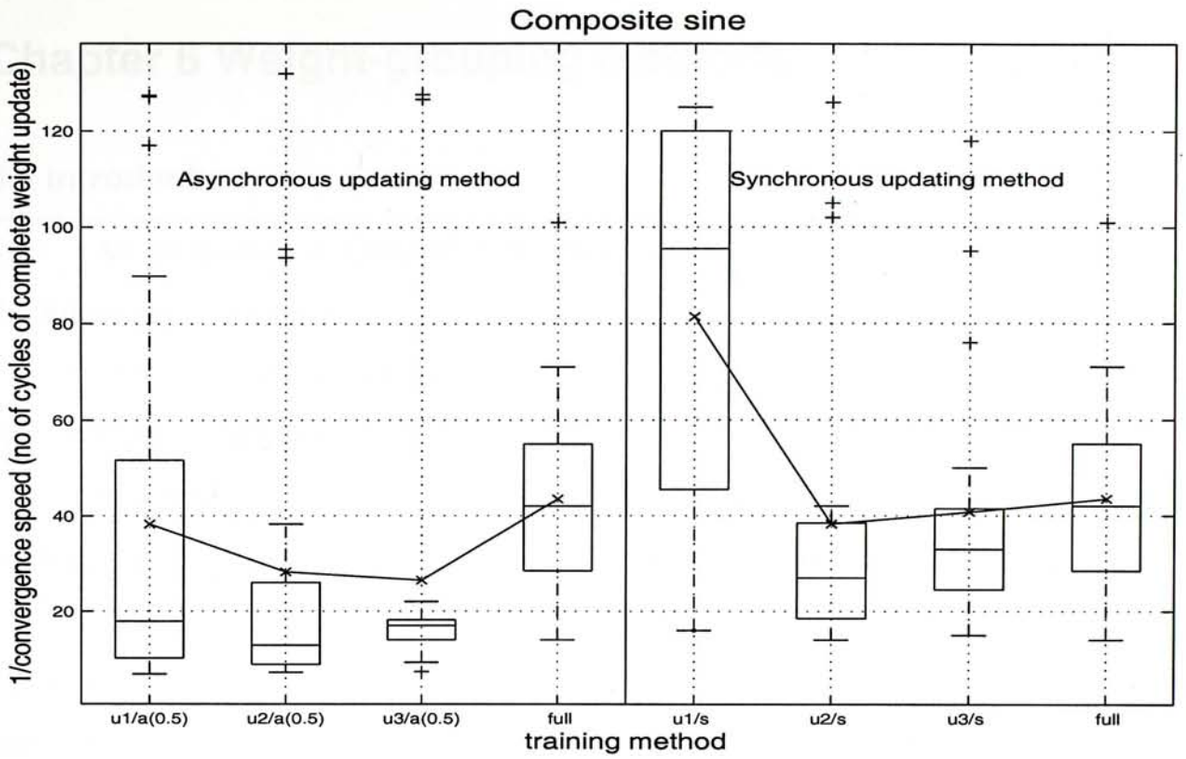


Figure 5-8 Inverse of convergence speed measured in terms of number of cycles of complete weight update. The x-axis shows the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axis are explained in Table 5-2. The composite sine training data was used in these experiments.

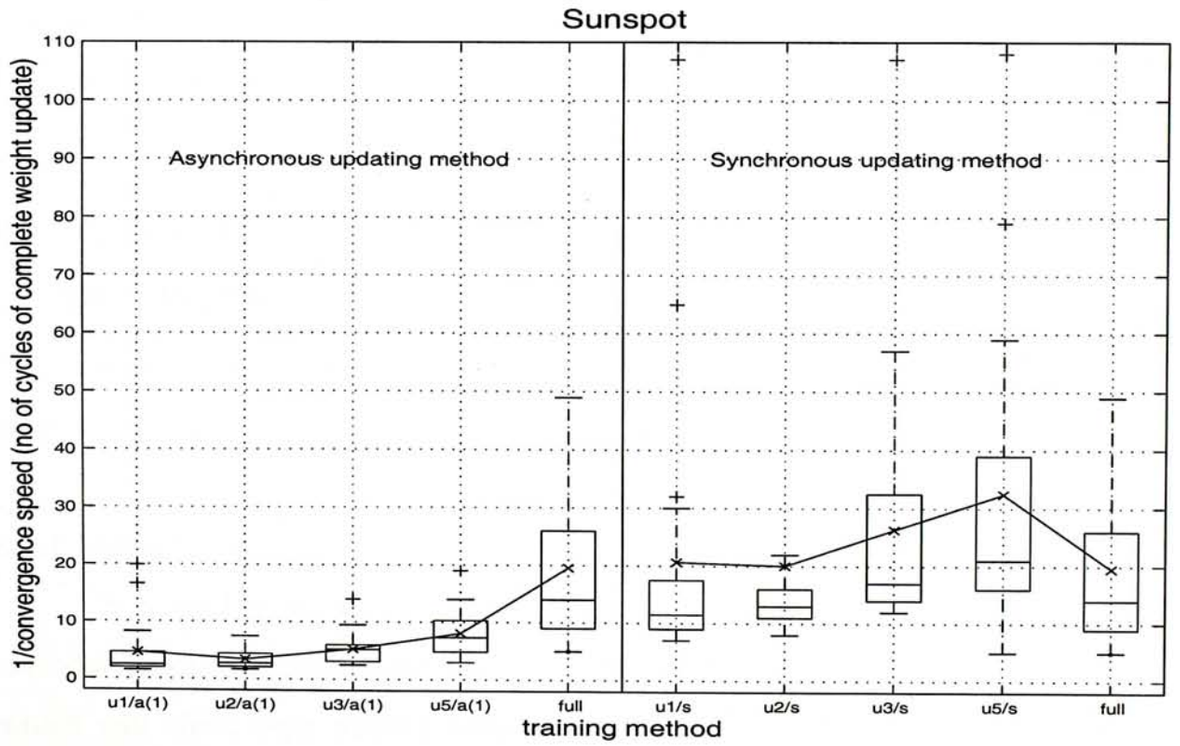


Figure 5-9 Inverse of convergence speed measured in terms of number of cycles of complete weight update. The x-axis shows the training methods using different updating methods and block-diagonal Hessian matrices. The labels of the x-axis are explained in Table 5-2. The sunspot training data was used in these experiments.

Chapter 6 Weight-grouping methods

6.1 Introduction

As mentioned in Chapter 2, the block-diagonal Hessian matrix depends on the following two factors.

i. Number and sizes of the blocks
ii. Weight-grouping method

In the previous chapter, the effect of number and sizes of the blocks has been studied. In this chapter, the effect of weight-grouping method will be studied.

Four particular Hessian matrices are introduced in Sections 2.3.1 to 2.3.4. They are the correlation, one-unit, three-unit and layer block-diagonal Hessian matrices. Their respective weight-grouping methods are called the correlation, one-unit, three-unit and layer weight-grouping methods.

Here, these weight-grouping methods are compared. We will have two studies. The first one described in Section 6.2 is to compare their final training time and their generalization errors. The second one described in Section 6.3 is to compare the degree of approximation of the block-diagonal Hessian matrices with these weight-grouping methods.

6.2 Training time and generalization performance of different weight-grouping methods

In this section, the performance of the correlation, one-unit, three-unit and layer weight-grouping methods is evaluated.

6.2.1 Method of study

It is not fair to compare these weight-grouping methods directly since these methods produce Hessian matrices with different numbers and sizes of the blocks, which will affect their training time performance. Instead, these weight-grouping methods were compared with the *arbitrary weight-grouping method.* The arbitrary weight-grouping method arranges the positions of elements in the weight vector **w** randomly. The number and sizes of the blocks of the block-diagonal Hessian matrix

with this weight-grouping method were set to be the same as those with the weight-grouping methods under study.

The block-diagonal Hessian matrices with these weight-grouping methods would be used with the asynchronous updating method described in Section 4.3 and the synchronous updating method described in Section 4.4.2. These training methods were used to train the layered fully recurrent network described in Section 1.2.3 to predict the single sine, composite sine and sunspot data described in Sections 3.2.1 to 3.2.3 respectively.

6.2.2 Performance

The performance of the weight-grouping methods is plotted in Figures 6-1 and 6-2. Figures 6-1 and 6-2 show the performance of using the synchronous and asynchronous updating methods respectively. Figures 6-1a and 6-2a show the training time performance. Figures 6-1b and 6-2b show the generalization performance. Each figure has four parts. These four parts show the comparisons between the performance of the weight-grouping methods under study and the performance of the arbitrary weight-grouping method. The methods under study are the correlation, one-unit, three-unit and layer weight-grouping methods. In each part of the figures, the performance of the weight-grouping method under study is shown in the left column and the performance of the arbitrary weight-grouping method used for reference is shown in the right column. Figures 6-1 to 6-2 show the performance of the training methods used to train the networks to predict the single sine data. The corresponding graphs of the performance of the training methods used to train the networks to predict the composite sine and sunspot data are shown in Figures 6-3 to 6-4 and Figures 6-5 to 6-6 respectively.

Table 6-1 summarizes the comparisons between the final training time of the weight-grouping methods under study and the final training time of the arbitrary weight-grouping method shown in Figures 6-1 to 6-6. The ratio of final training time of the weight-grouping method under study to the final training time of the arbitrary weight-grouping method was used in the comparisons. If the ratio is less than one, the training time performance of the weight-grouping method under study is better than that of the arbitrary weight-grouping method. Otherwise, the training time performance of the weight-grouping method under study is worse. Median

value of 20 trials was used in calculating this ratio because outliers were found in some cases.

Weight-grouping method	Percentage of non-zero Hessian elements		Training time performance relative to that of the arbitrary weight-grouping method					
	Networks with six hidden units†	Networks with nine hidden units‡	Synchronous updating			Asynchronous updating		
			Single sine data	Composite sine data	Sunspot data	Single sine data	Composite sine data	Sunspot data
Correlation	0.11	0.08	0.96	1.06	1.50	1.22	0.40	1.37
One-unit	0.14	0.10	1.08	1.66	0.57	1.05	0.88	0.55
Three-unit	0.40	0.28	0.93	0.65	0.89	0.78	0.76	0.93
Layer	0.49	0.59	1.73	1.51	1.04	1.21	1.06	0.95

† Networks with six hidden units were used to predict the single and composite sine data
‡ Networks with nine hidden units were used to predict the sunspot data

Table 6-1 Comparisons of different weight-grouping methods: Percentage of non-zero Hessian elements under different numbers of hidden units. Ratios of the final training time of different weight-grouping methods to that of the arbitrary weight-grouping method under different training data and updating methods. Median value of 20 trials was used in calculating these ratios.

Table 6-1 shows that the training time performance of the three-unit weight-grouping method was better than that of the arbitrary weight-grouping method under different updating methods and training data. However, the training time performance of the layer weight-grouping method was worse in most of the experiments. The relative training time performance of the correlation and one-unit weight-grouping methods with respect to the arbitrary weight-grouping method varied with the updating methods and/or the training data.

In Table 6-1, each weight-grouping method is listed with the percentage of non-zero Hessian elements. The relative training time performance of the block-diagonal Hessian matrices with small number of non-zero elements, like the correlation and one-unit block-diagonal matrices, varied with the updating methods and/or the training data. In contrast, the relative training time performance of the block-diagonal Hessian matrices with moderate number of non-zero elements, like the three-unit and layer block-diagonal matrices, was relatively insensitive to the updating methods and the training data.

Weight-grouping methods affect the degree of approximation of a block-diagonal Hessian matrix, which we used the sum of the absolute values of all the Hessian elements of a block-diagonal Hessian matrix relative to that of the full Hessian matrix to measure. And the degree of approximation of a block-diagonal Hessian matrix affects the training performance of our proposed method. In the next section, we will compare the degree of Hessian approximation of the block-diagonal Hessian matrices with different weight-grouping methods and study its relationship with the training time performance.

6.3 Degree of approximation of block-diagonal Hessian matrices with different weight-grouping methods

In this section, the degree of Hessian approximation of the block-diagonal Hessian matrices with the correlation, one-unit, three-unit and layer weight-grouping methods is evaluated.

6.3.1 Method of study

Like what we did in Section 6.2, we do not directly compare the degree of approximation of the block-diagonal Hessian matrices with the correlation, one-unit, three-unit and layer weight-grouping methods. It is because these methods produce Hessian matrices with different numbers and sizes of the blocks, which will affect the degree of approximation. Instead, we compared the degree of approximation of the block-diagonal Hessian matrices with the weight-grouping methods under study to that with the arbitrary weight-grouping method. The number and sizes of the blocks of the block-diagonal Hessian matrix with the arbitrary weight-grouping method were set to be the same as those with the weight-grouping methods under study.

6.3.2 Performance

The degree of approximation of the block-diagonal Hessian matrices with the correlation and arbitrary weight-grouping methods over the whole training process was recorded and plotted on Figure 6-7. The x and y axes represent the training time and the degree of approximation respectively. The correlation and arbitrary weight-

grouping methods are represented by the line types '—x—' and '—o—' respectively. The degree of approximation shown in the figure is the mean value of 20 trials. Figures 6.7a and 6.7b show the degree of approximation of using the synchronous and asynchronous updating methods respectively. Similar plots of the degree of approximation of the block-diagonal Hessian matrices with the one-unit, three-unit and layer weight-grouping methods are shown in Figures 6-8 to 6-10 respectively. Figures 6-7 to 6-10 show the degree of approximation of training the networks to predict the single sine data. The corresponding graphs of the degree of approximation of training the networks to predict the composite sine and sunspot data are shown in Figures 6-11 to 6-14 and Figures 6-15 to 6-18 respectively.

These figures show that the degree of approximation of the block-diagonal Hessian matrices with the correlation and layer weight-grouping methods was often lower than that with the arbitrary weight-grouping methods during the whole training period. However, the degree of approximation of the block-diagonal Hessian matrices with the one-unit and three-unit weight-grouping methods was often higher. Table 6-2 summarizes the comparisons. These results showed that the magnitudes of some Hessian elements were often larger than those of the others. If these elements are identified, a block-diagonal Hessian matrix with higher degree of approximation can be constructed.

Weight-grouping method	Degree of approximation of the block-diagonal Hessian matrices relative to that with the arbitrary weight-grouping method
Correlation	Lower
One-unit	Higher
Three-unit	Higher
Layer	Lower

Table 6-2 Degree of approximation of the block-diagonal Hessian matrices with the correlation, one-unit, three-unit and layer weight-grouping methods relative to that with the arbitrary weight-grouping method under different training data and updating methods.

Finally, we combined the results in Section 6.2 and this section and related the degree of Hessian approximation to the training time performance. If the block-diagonal Hessian matrices with moderate number of non-zero elements were used, the weight-grouping methods with higher degree of Hessian approximation had

better training time performance. From our experimental results, there is only one exceptional case to the above statement. The degree of approximation of the block-diagonal Hessian matrix with the layer weight-grouping method is lower than that with the arbitrary weight-grouping method. But, when the asynchronous updating method was used to train the networks to predict the sunspot data, the training time performance of the layer weight-grouping method is better. We observed from Figure 6.18 that although the relative degree of approximation is lower, the absolute degrees of both curves are high. They have reached 0.65 for the layer weight-grouping method. This implies that it is not necessary to have very accurate approximation to the Hessian matrix.

If the block-diagonal Hessian matrices with small number of non-zero elements were used, we did not observe any correlation between the degree of approximation and the training time performance. The reason is that the degree of approximation of these block-diagonal Hessian matrices was very poor. So, its effect on the training time performance became less dominant.

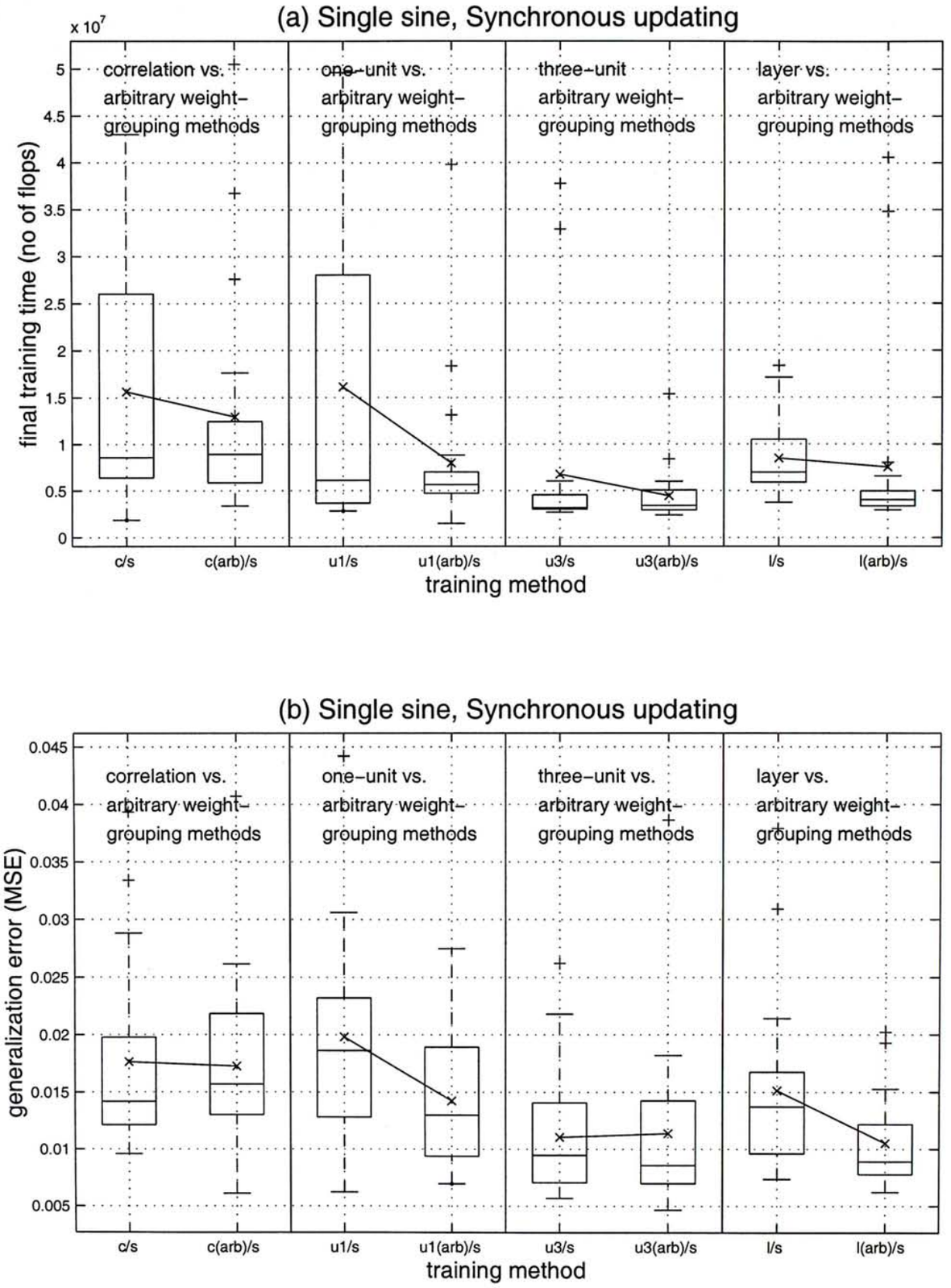


Figure 6-1 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using the synchronous updating method and different block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 6-3. The single sine training data was used in these experiments.

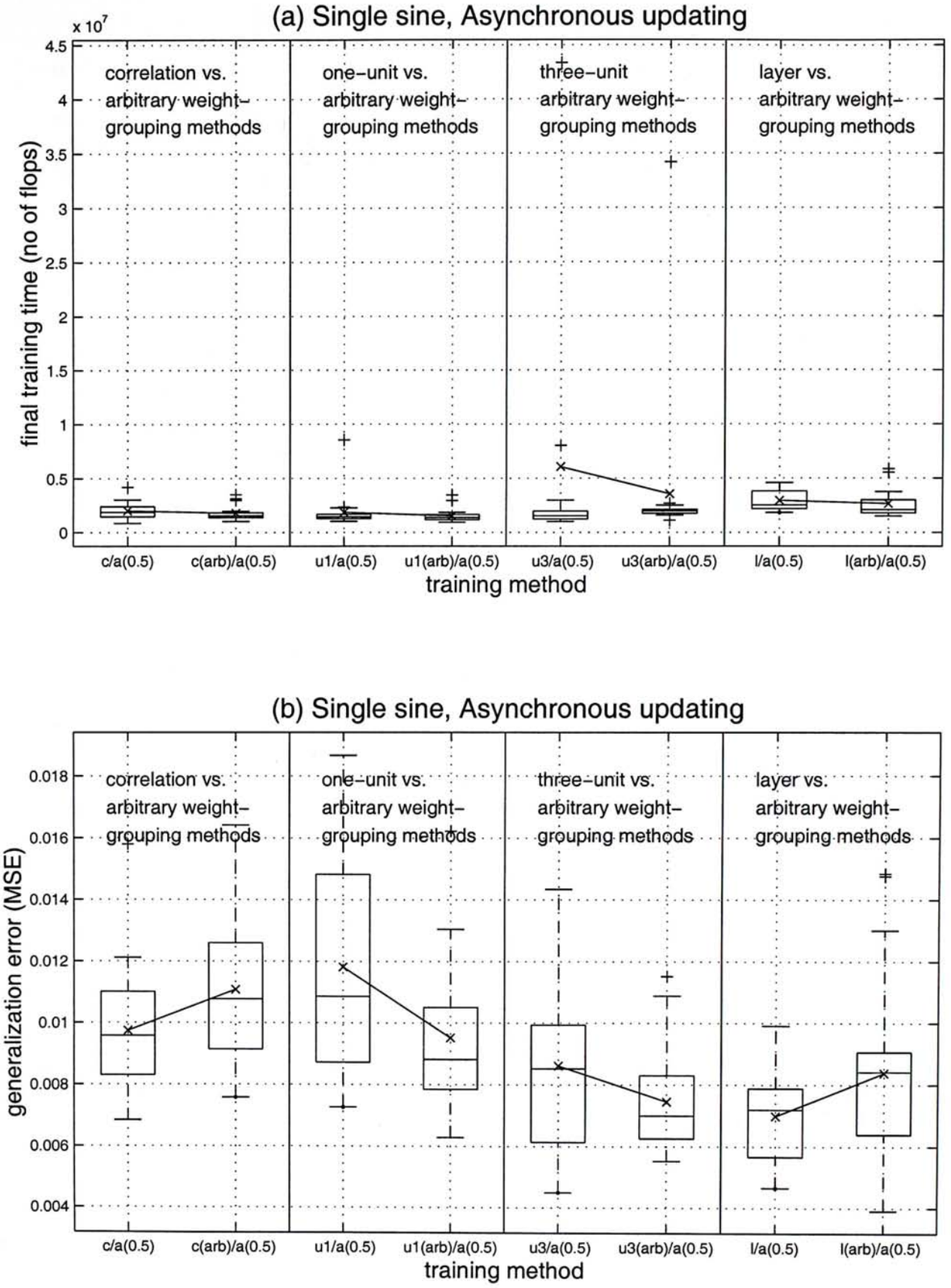


Figure 6-2 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using the asynchronous updating method and different block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 6-3. The single sine training data was used in these experiments.

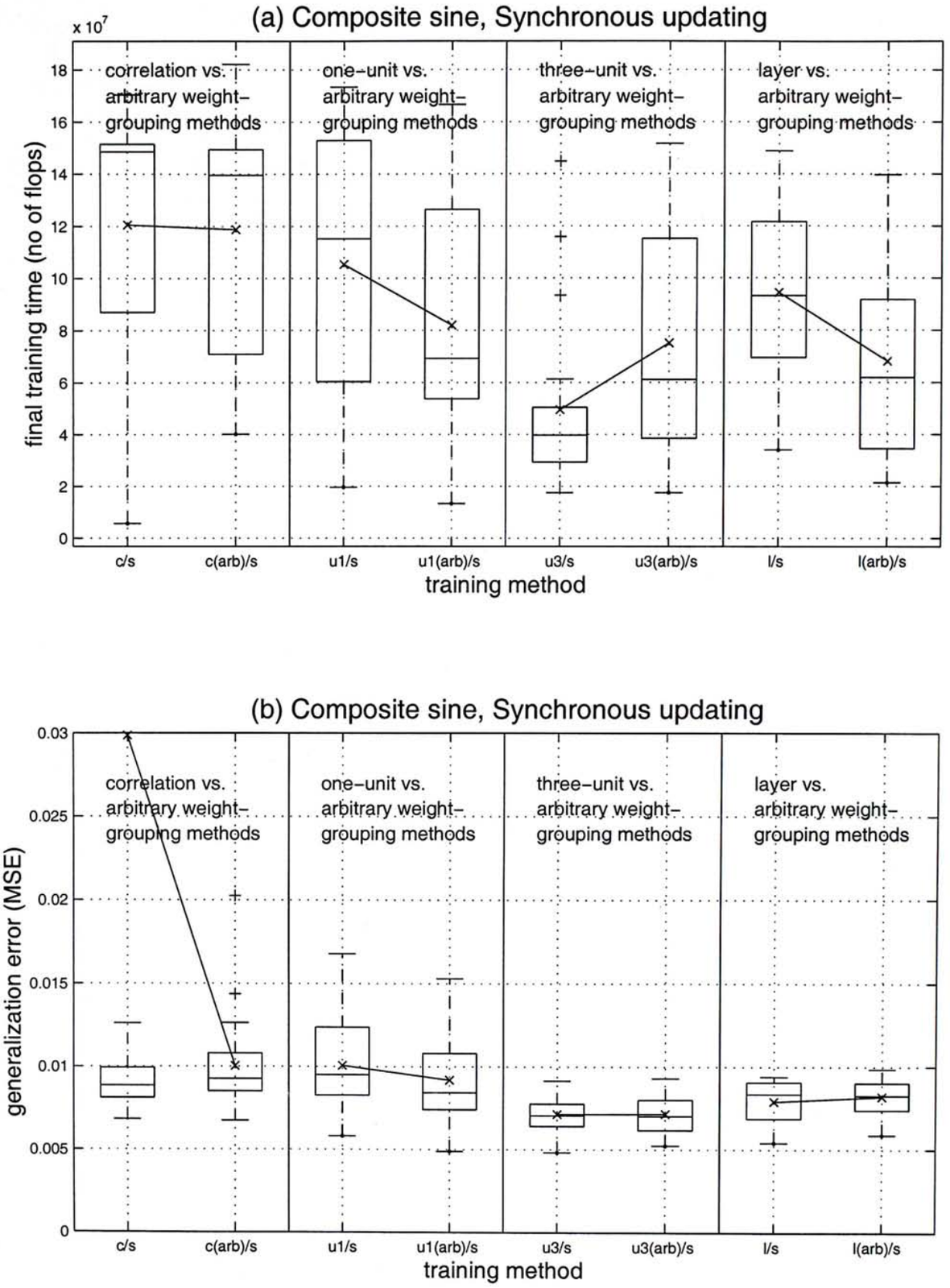
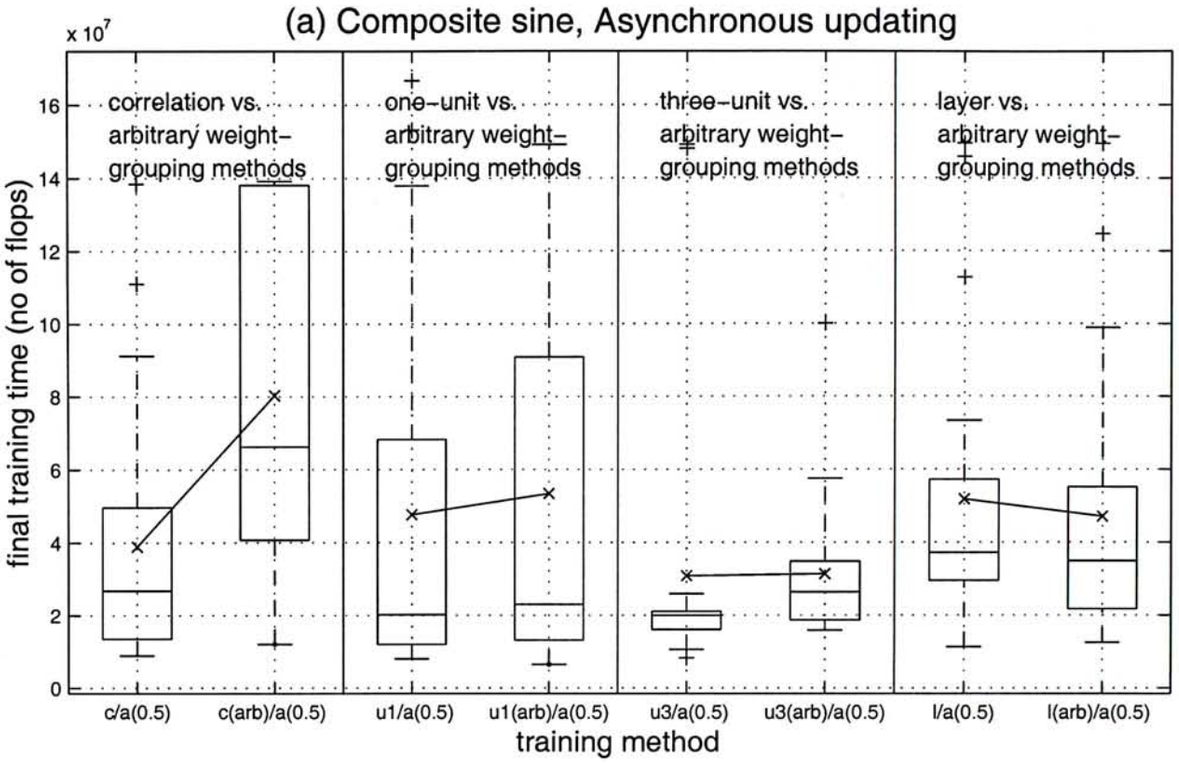


Figure 6-3 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using the synchronous updating method and different block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 6-3. The composite sine training data was used in these experiments.



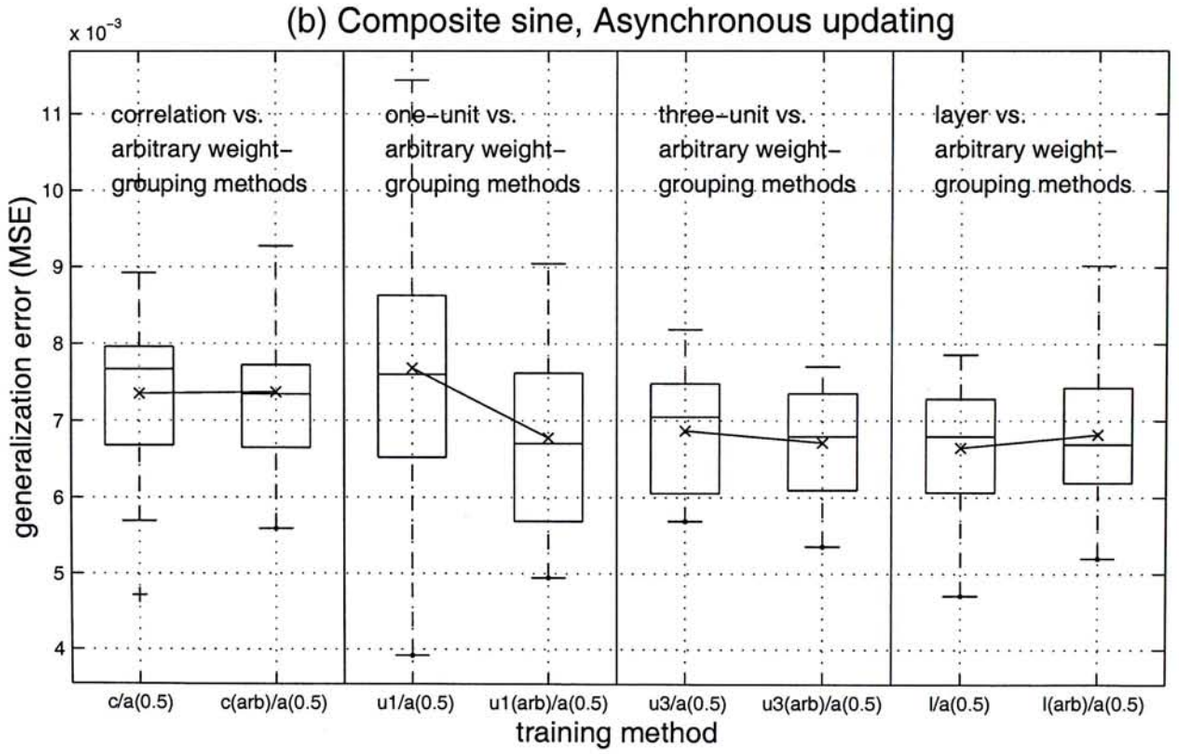


Figure 6-4 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using the asynchronous updating method and different block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 6-3. The composite sine training data was used in these experiments.

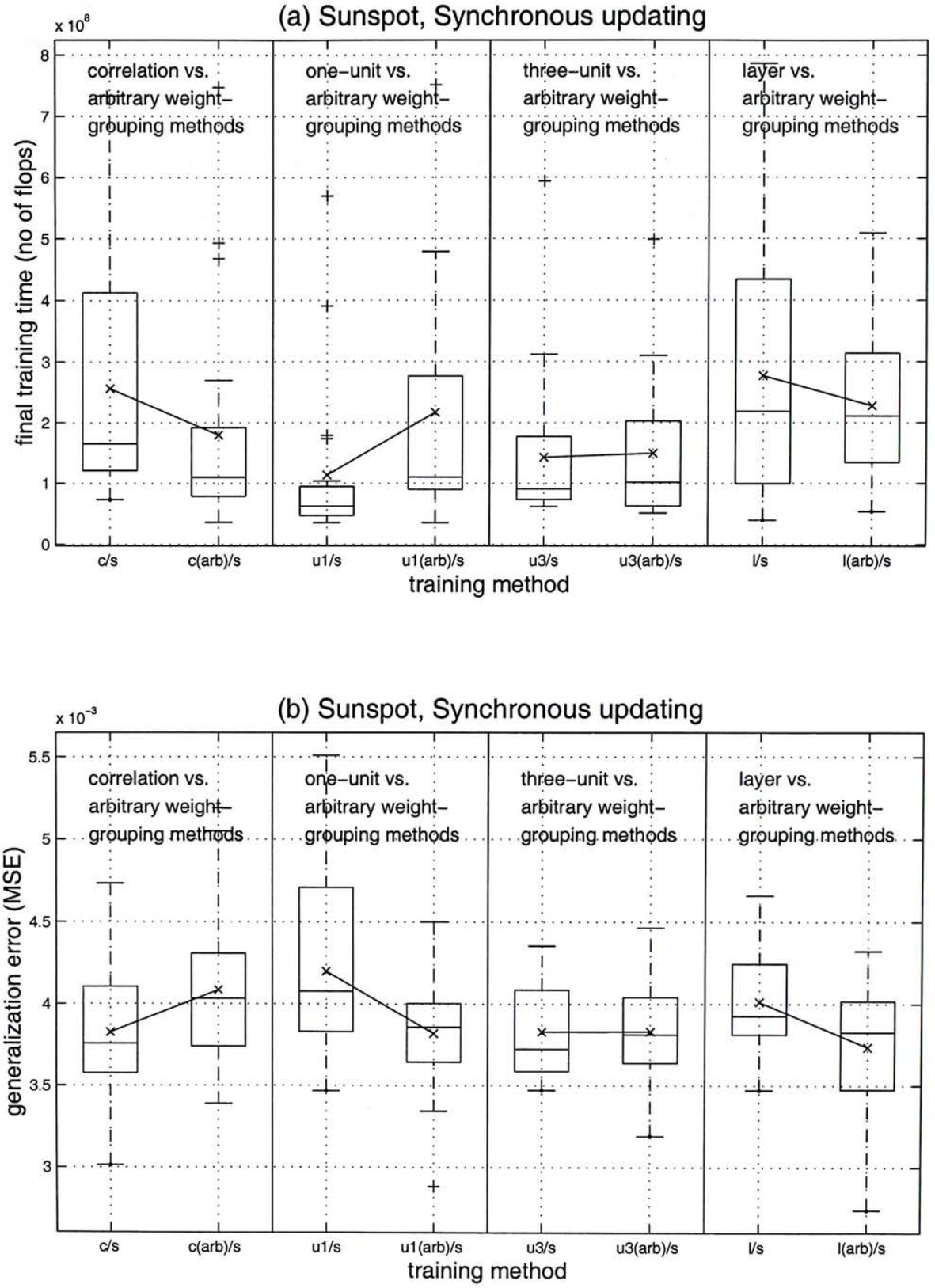
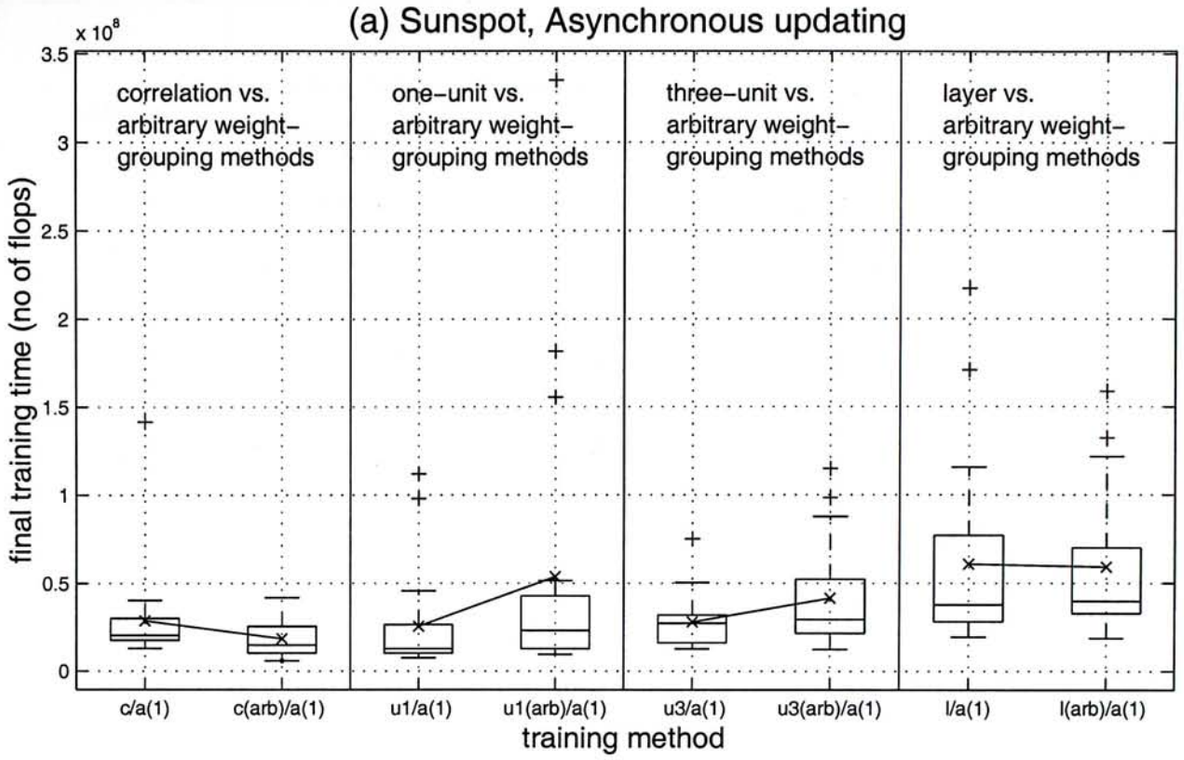


Figure 6-5 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using the synchronous updating method and different block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 6-3. The sunspot training data was used in these experiments.



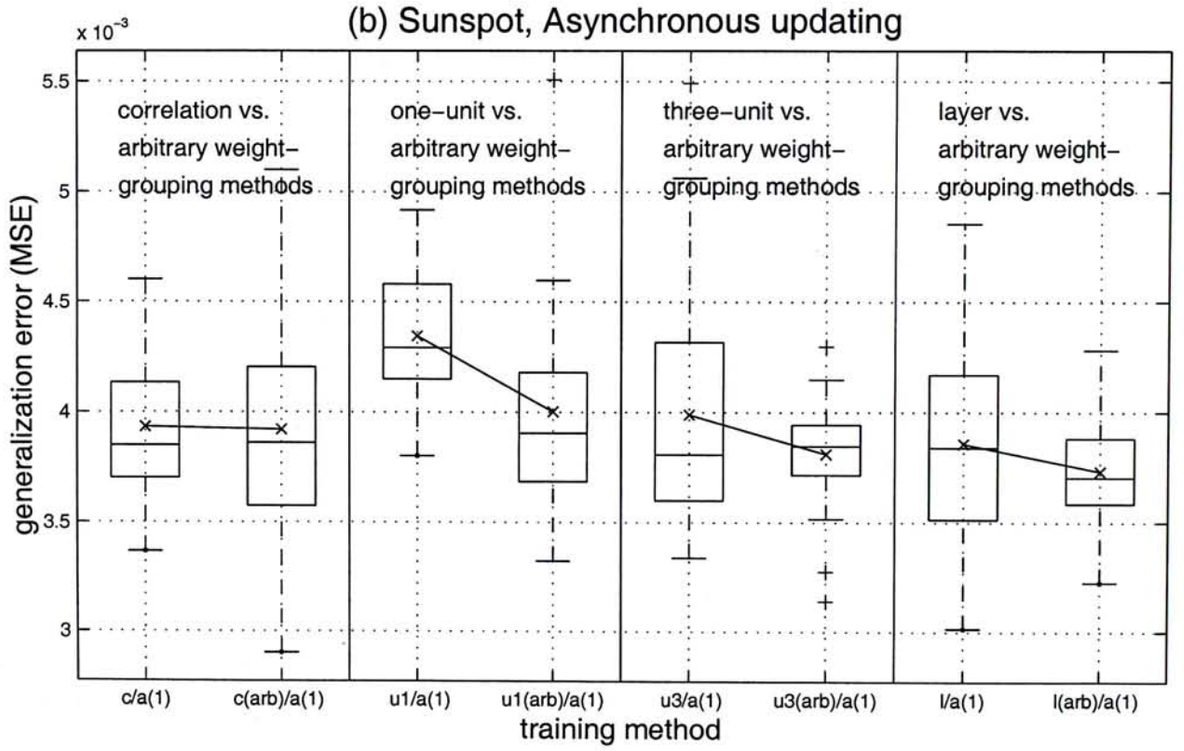


Figure 6-6 (a) Final training time measured in terms of number of flops and (b) generalization errors measured in terms of mean squared error. The x-axes show the training methods using the asynchronous updating method and different block-diagonal Hessian matrices. The labels of the x-axes are explained in Table 6-3. The sunspot training data was used in these experiments.

Label	Descriptions			
	block-diagonal Hessian matrix		updating method	maximum allowed weight change
	weight-grouping method	number and sizes of blocks		
c/s	correlation	the same under the same network size	synchronous	N/A
c(arb)/s	arbitrary			
c/a(0.5)	correlation		asynchronous	0.5
c(arb)/a(0.5)	arbitrary			
c/a(1)	correlation			1
c(arb)/a(1)	arbitrary			
u1/s	one-unit	the same under the same network size	synchronous	N/A
u1(arb)/s	arbitrary			
u1/a(0.5)	one-unit		asynchronous	0.5
u1(arb)/a(0.5)	arbitrary			
u1/a(1)	one-unit			1
u1(arb)/a(1)	arbitrary			
u3/s	three-unit	the same under the same network size	synchronous	N/A
u3(arb)/s	arbitrary			
u3/a(0.5)	three-unit		asynchronous	0.5
u3(arb)/a(0.5)	arbitrary			
u3/a(1)	three-unit			1
u3(arb)/a(1)	arbitrary			
l/s	layer	the same under the same network size	synchronous	N/A
l(arb)/s	arbitrary			
l/a(0.5)	layer		asynchronous	0.5
l(arb)/a(0.5)	arbitrary			
l/a(1)	layer			1
l(arb)/a(1)	arbitrary			

Table 6-3 Descriptions of labels shown in Figures 6-1 to 6-6.

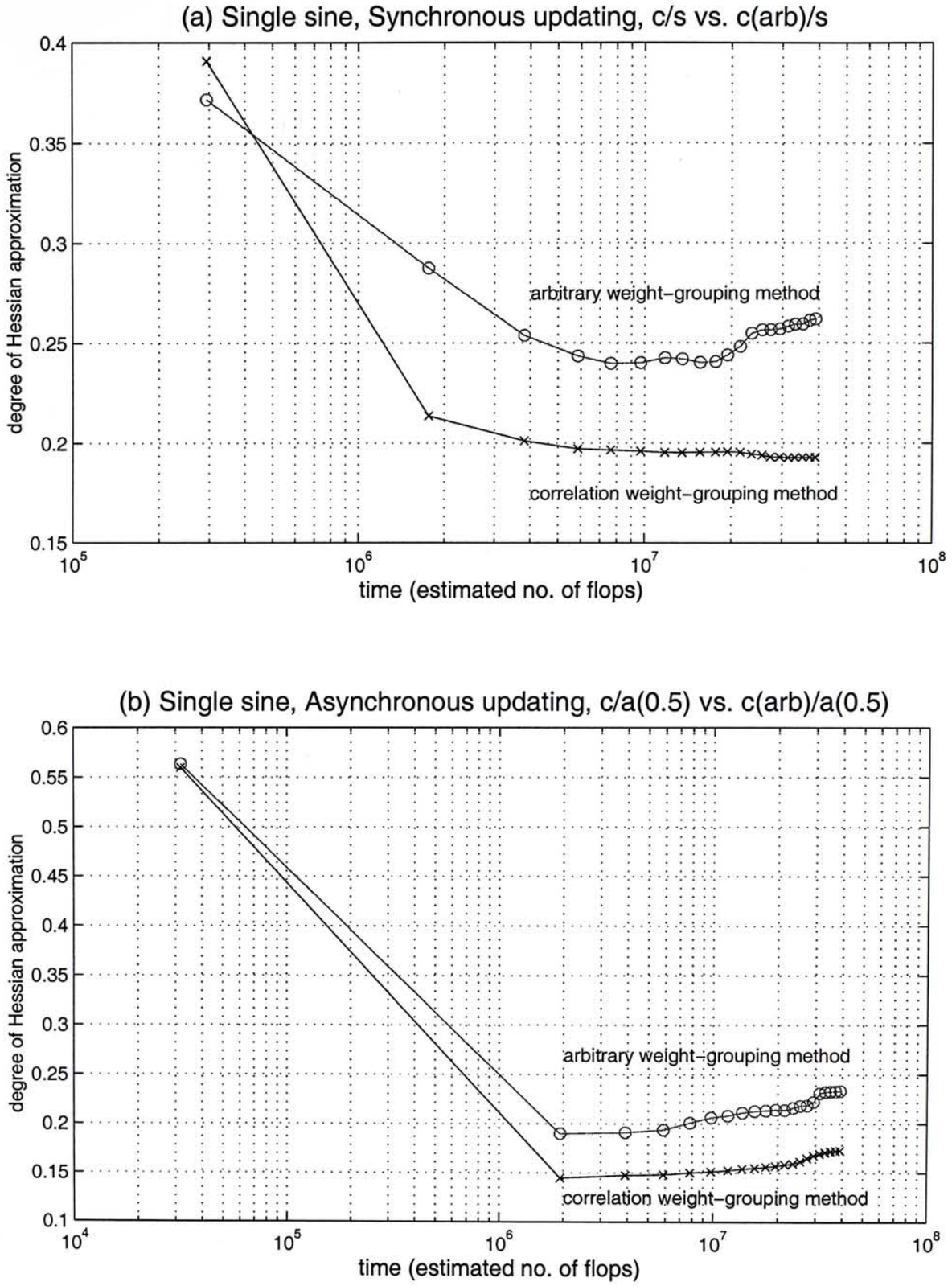


Figure 6-7 Degree of Hessian approximation during the whole training period. It is measured in terms of sum of the absolute values of all the Hessian elements relative to that of the full Hessian matrix. The (a) synchronous updating method, (b) asynchronous updating method and the single sine training data were used in these experiments. The correlation and arbitrary weight-grouping methods are represented by the line types '—x—' and '—o—' respectively.

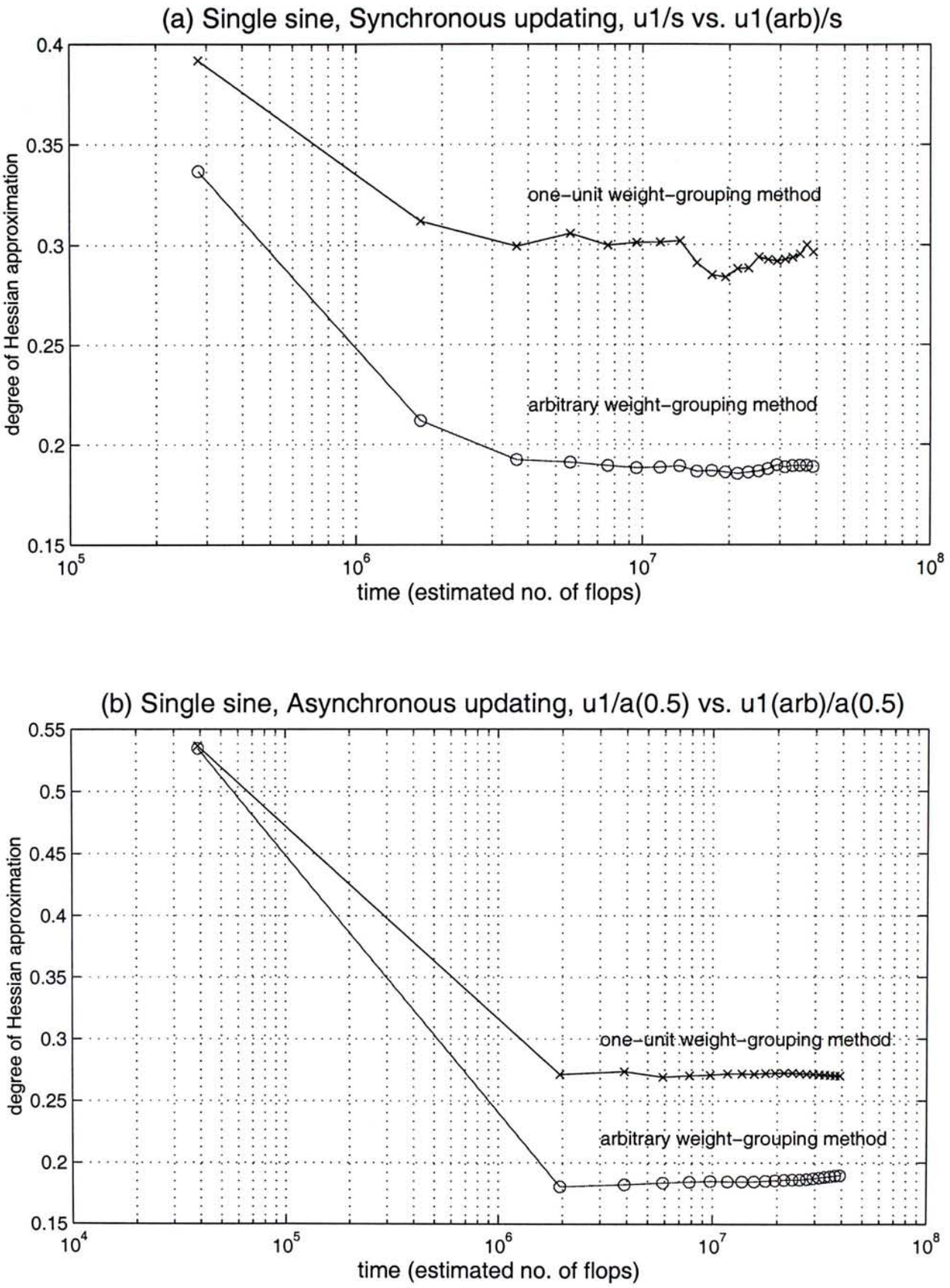


Figure 6-8 Degree of Hessian approximation during the whole training period. It is measured in terms of sum of the absolute values of all the Hessian elements relative to that of the full Hessian matrix. The (a) synchronous updating method, (b) asynchronous updating method and the single sine training data were used in these experiments. The one-unit and arbitrary weight-grouping methods are represented by the line types '—x—' and '—o—' respectively.

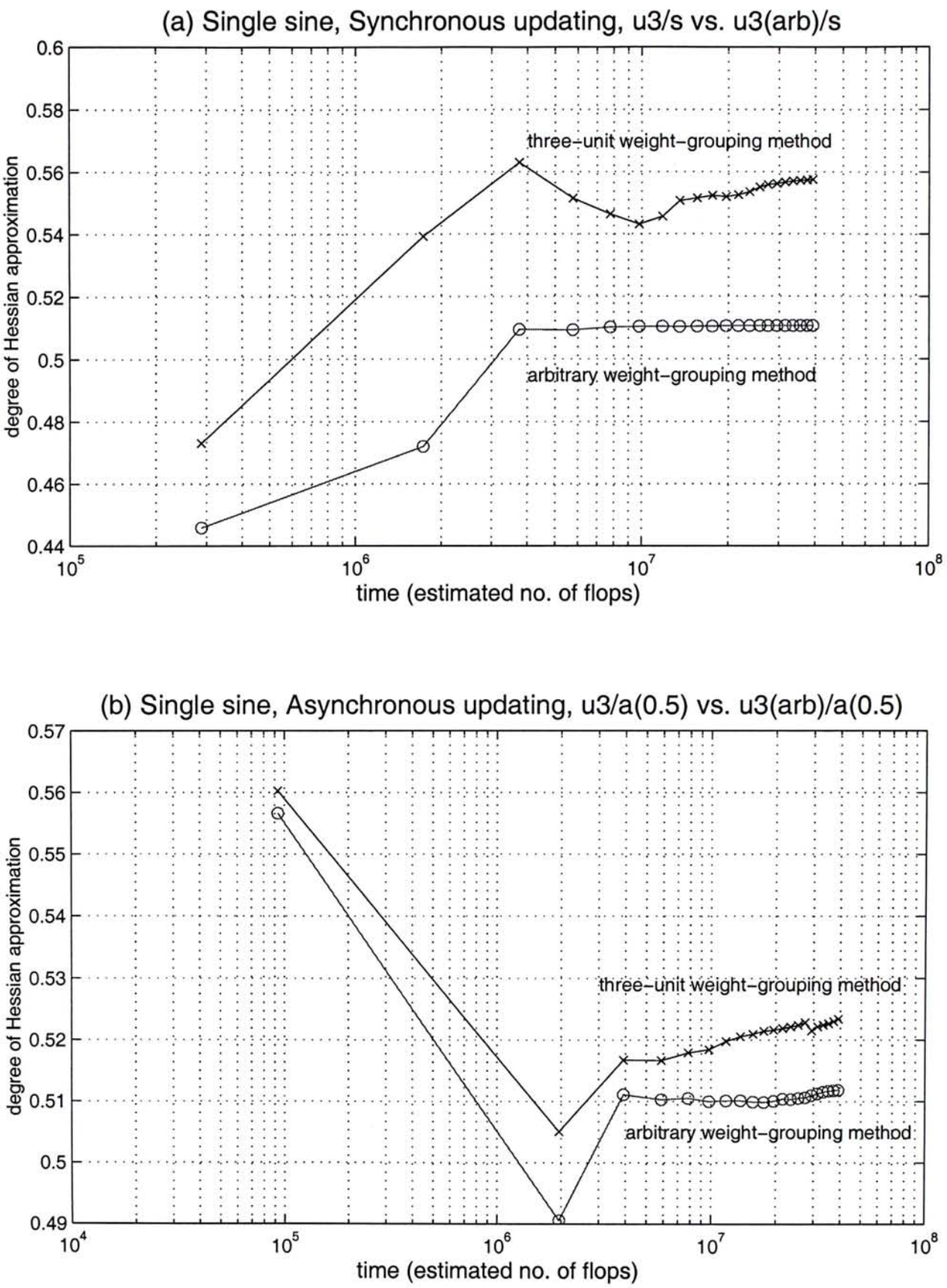


Figure 6-9 Degree of Hessian approximation during the whole training period. It is measured in terms of sum of the absolute values of all the Hessian elements relative to that of the full Hessian matrix. The (a) synchronous updating method, (b) asynchronous updating method and the single sine training data were used in these experiments. The three-unit and arbitrary weight-grouping methods are represented by the line types '—x—' and '—o—' respectively.

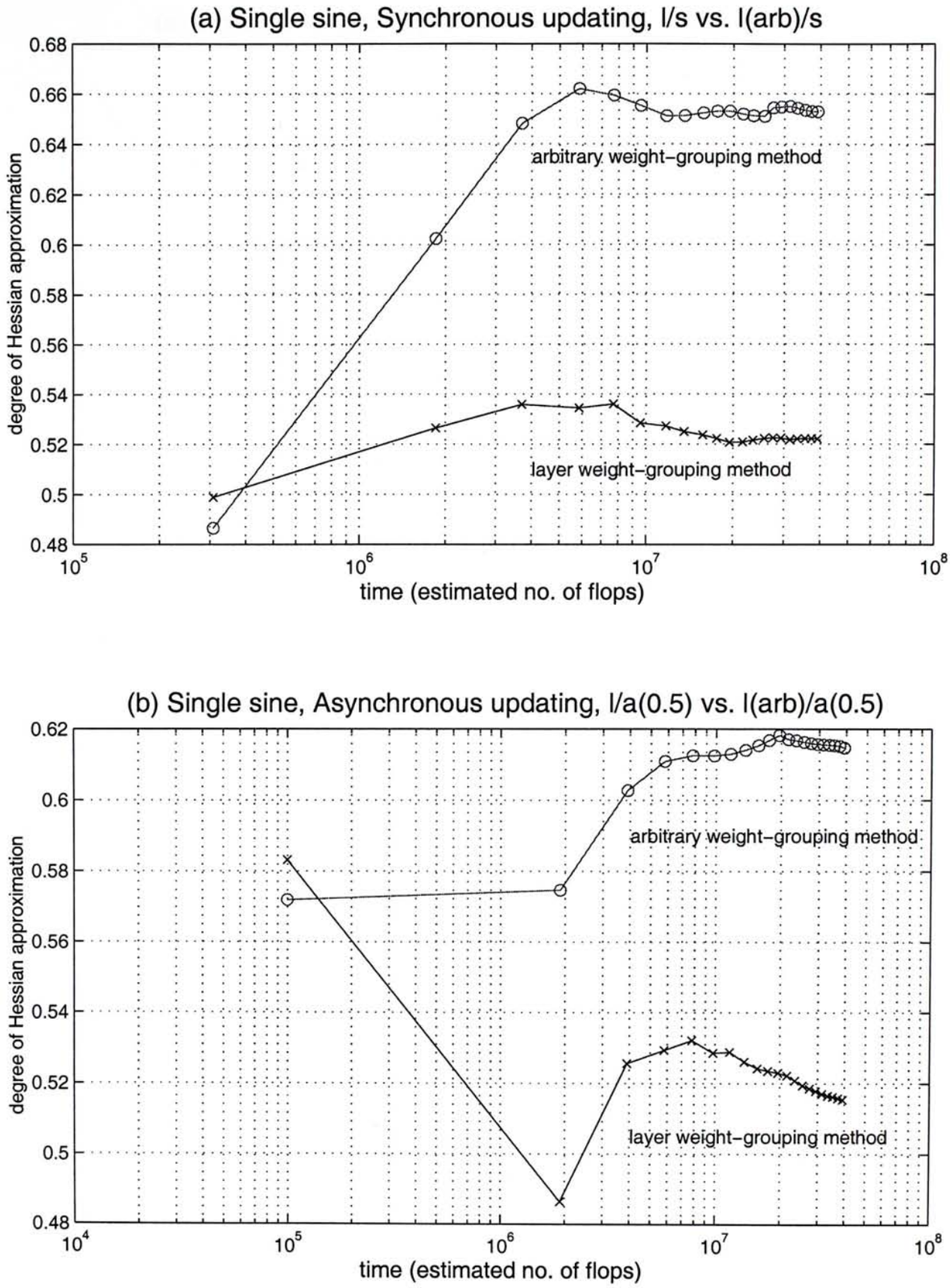
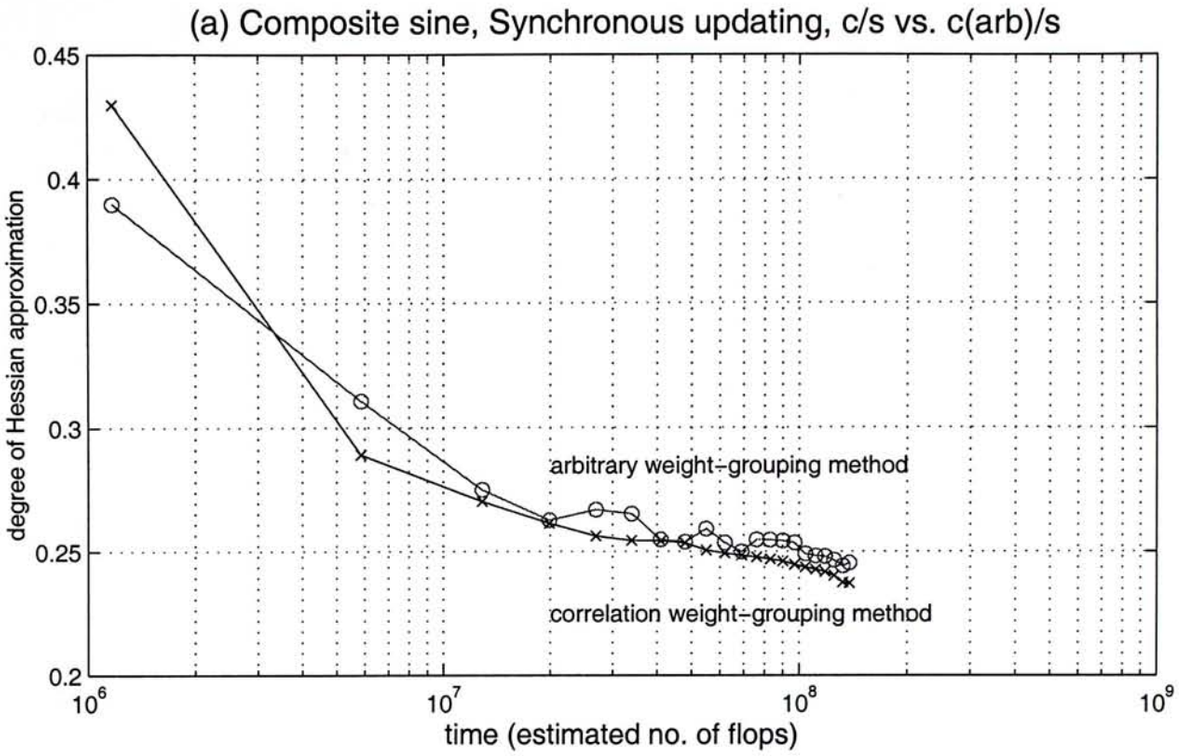


Figure 6-10 Degree of Hessian approximation during the whole training period. It is measured in terms of sum of the absolute values of all the Hessian elements relative to that of the full Hessian matrix. The (a) synchronous updating method, (b) asynchronous updating method and the single sine training data were used in these experiments. The layer and arbitrary weight-grouping methods are represented by the line types '—x—' and '—o—' respectively.



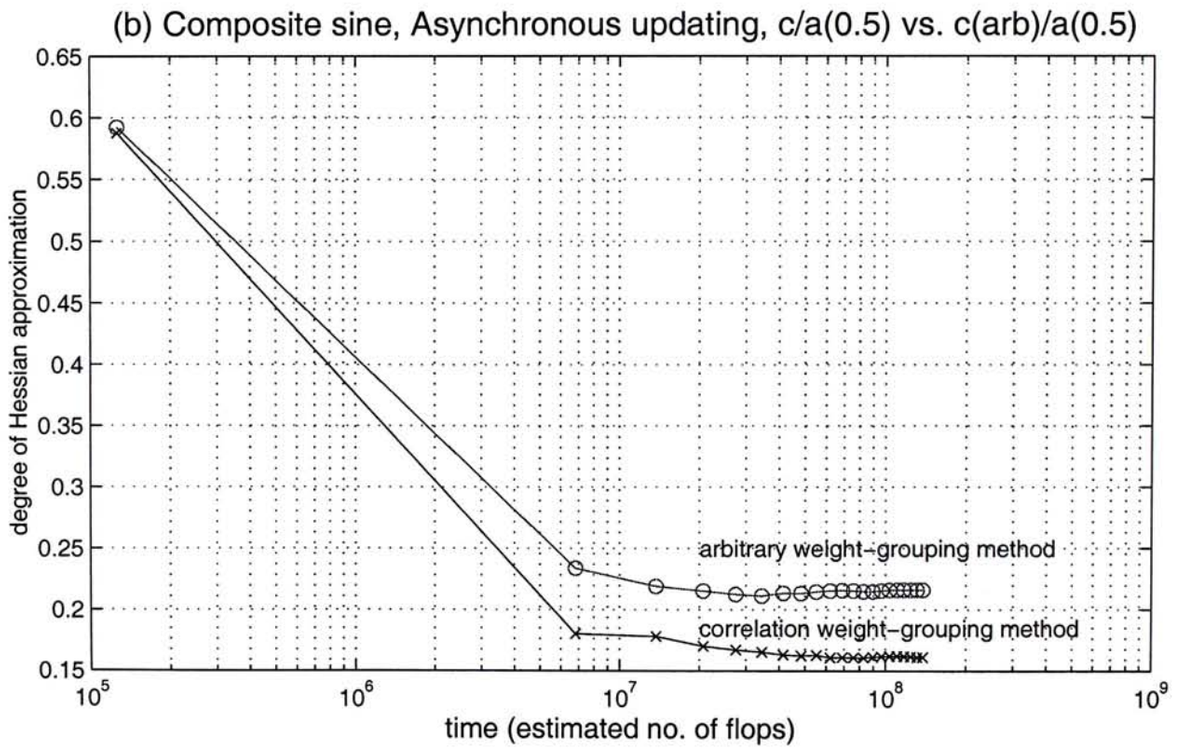
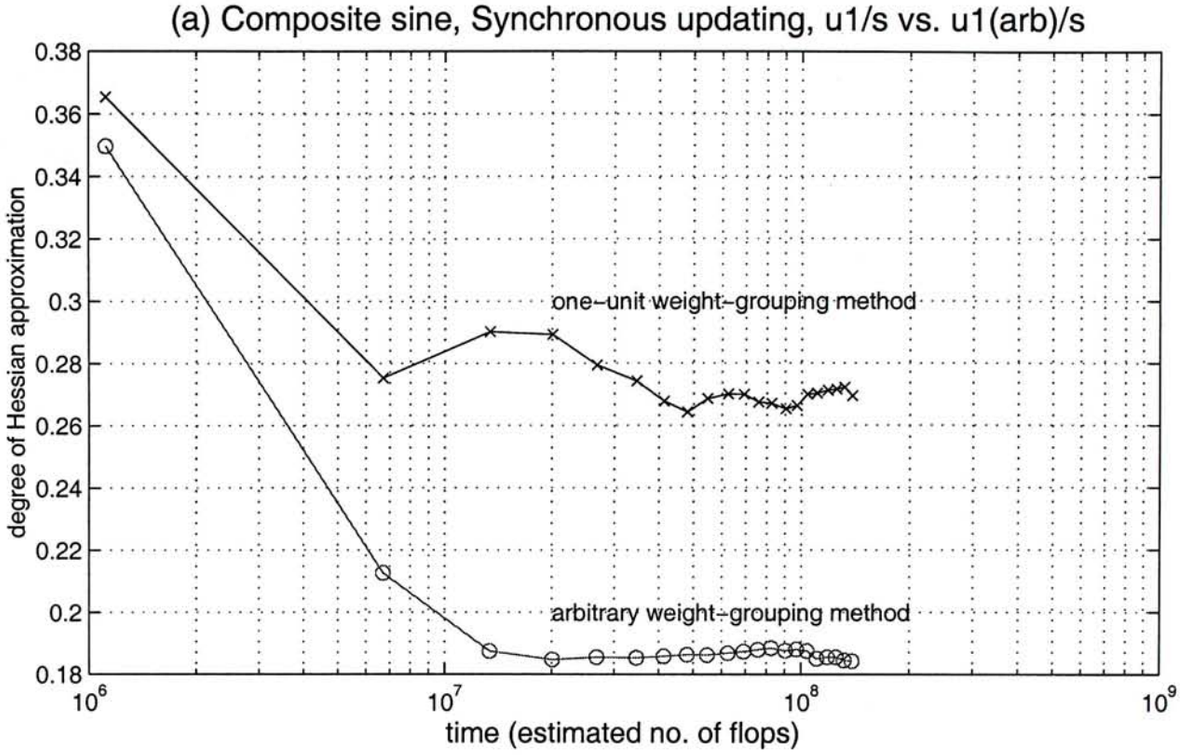


Figure 6-11 Degree of Hessian approximation during the whole training period. It is measured in terms of sum of the absolute values of all the Hessian elements relative to that of the full Hessian matrix. The (a) synchronous updating method, (b) asynchronous updating method and the composite sine training data were used in these experiments. The correlation and arbitrary weight-grouping methods are represented by the line types '—x—' and '—o—' respectively.



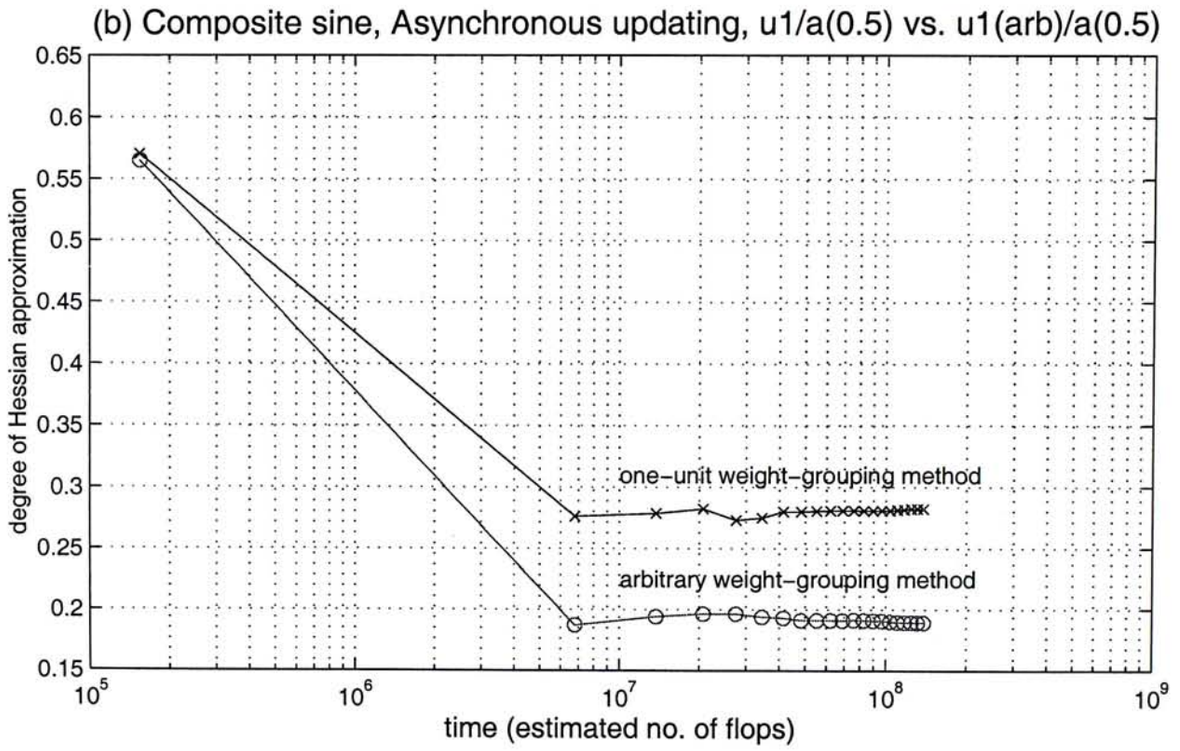
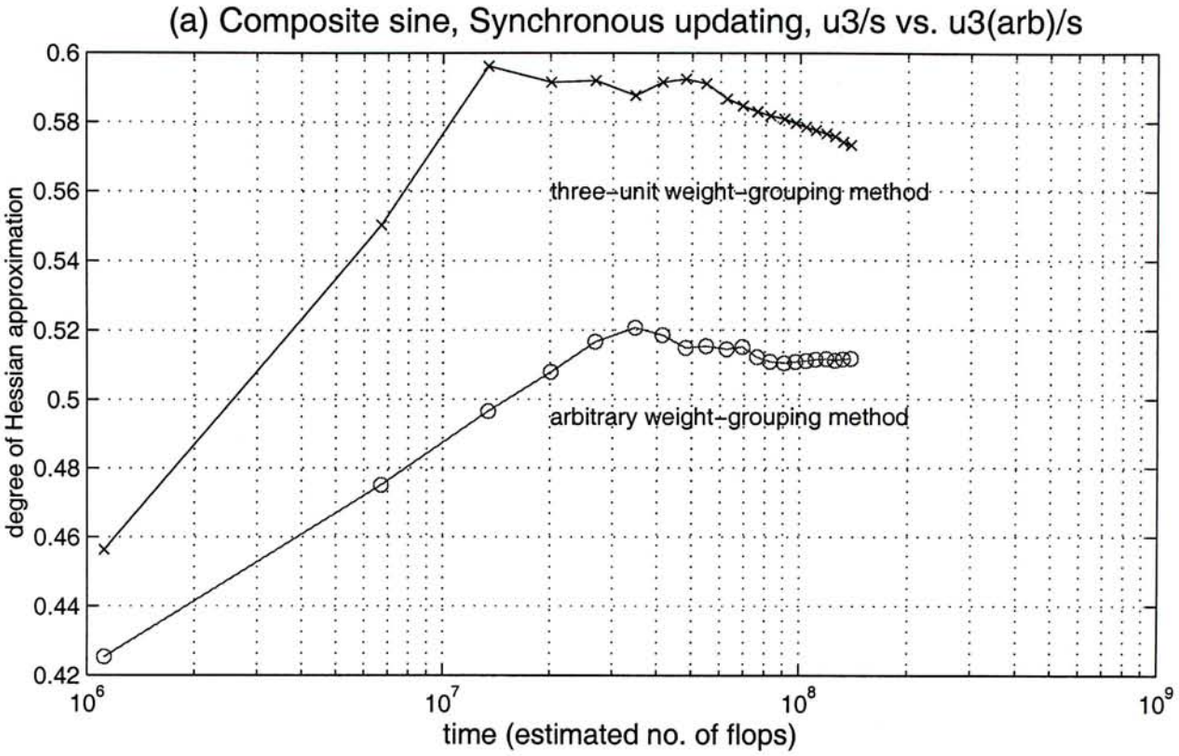


Figure 6-12 Degree of Hessian approximation during the whole training period. It is measured in terms of sum of the absolute values of all the Hessian elements relative to that of the full Hessian matrix. The (a) synchronous updating method, (b) asynchronous updating method and the composite sine training data were used in these experiments. The one-unit and arbitrary weight-grouping methods are represented by the line types '—x—' and '—o—' respectively.



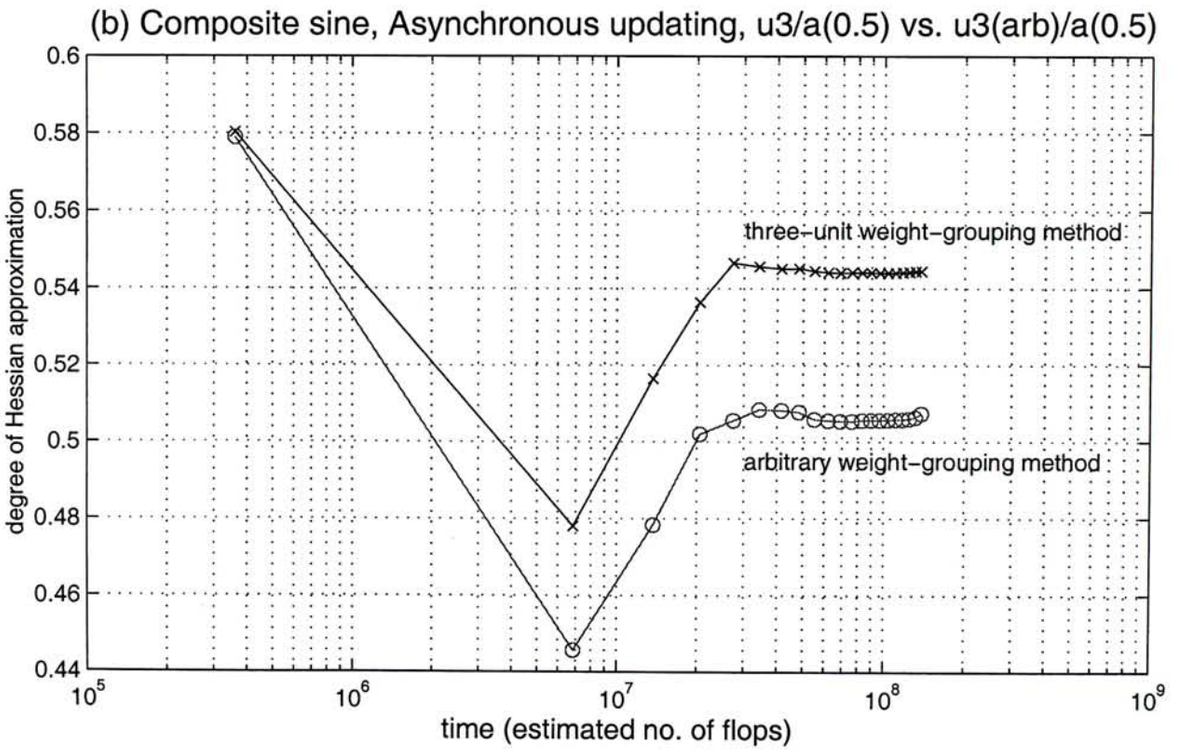
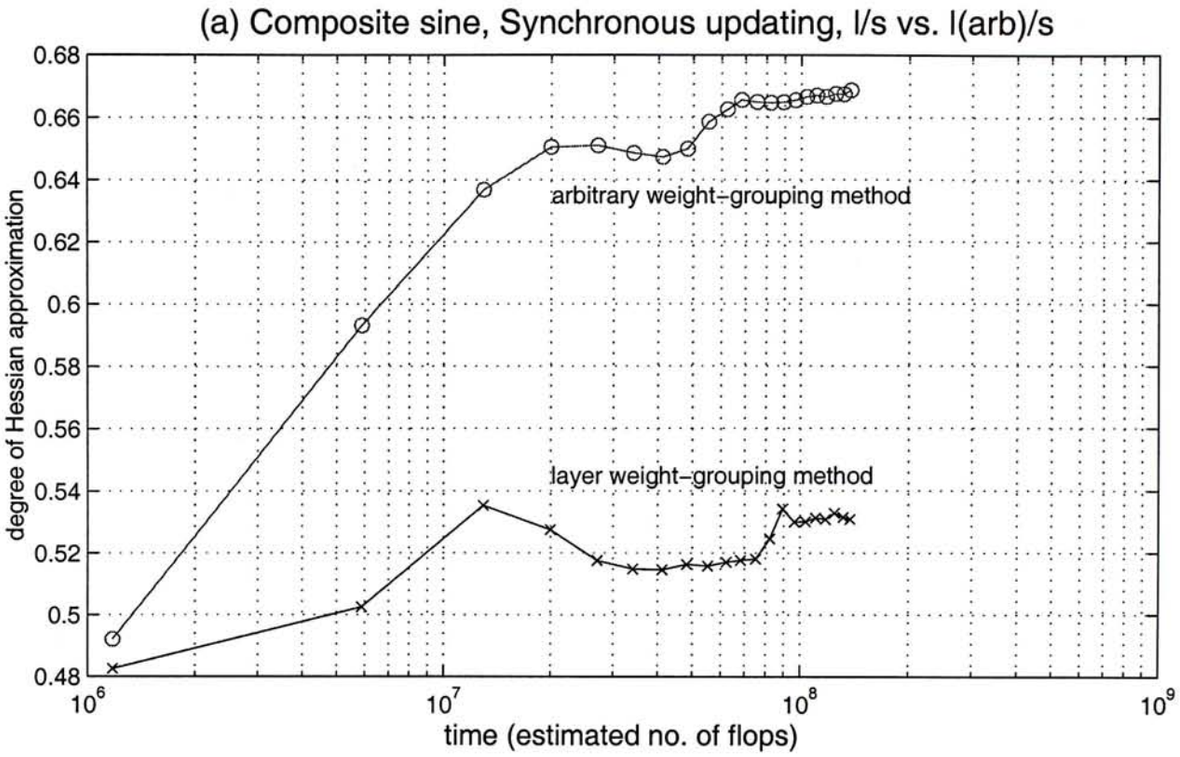


Figure 6-13 Degree of Hessian approximation during the whole training period. It is measured in terms of sum of the absolute values of all the Hessian elements relative to that of the full Hessian matrix. The (a) synchronous updating method, (b) asynchronous updating method and the composite sine training data were used in these experiments. The three-unit and arbitrary weight-grouping methods are represented by the line types '—x—' and '—o—' respectively.



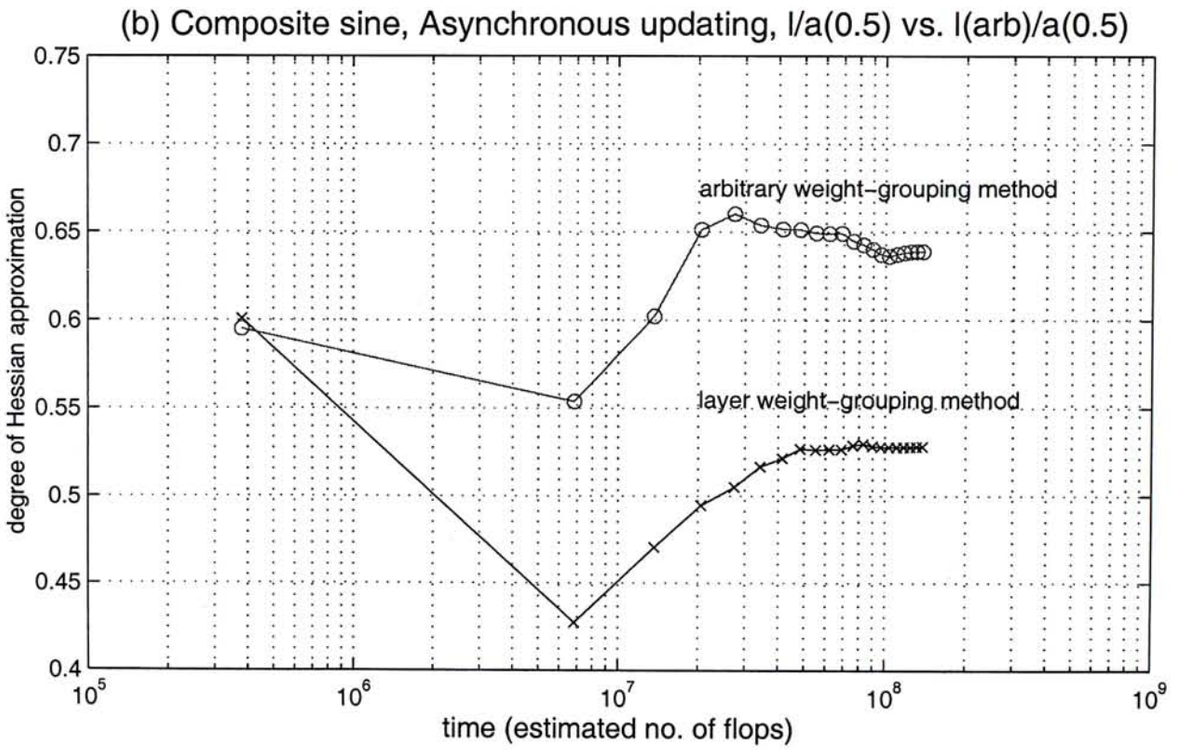
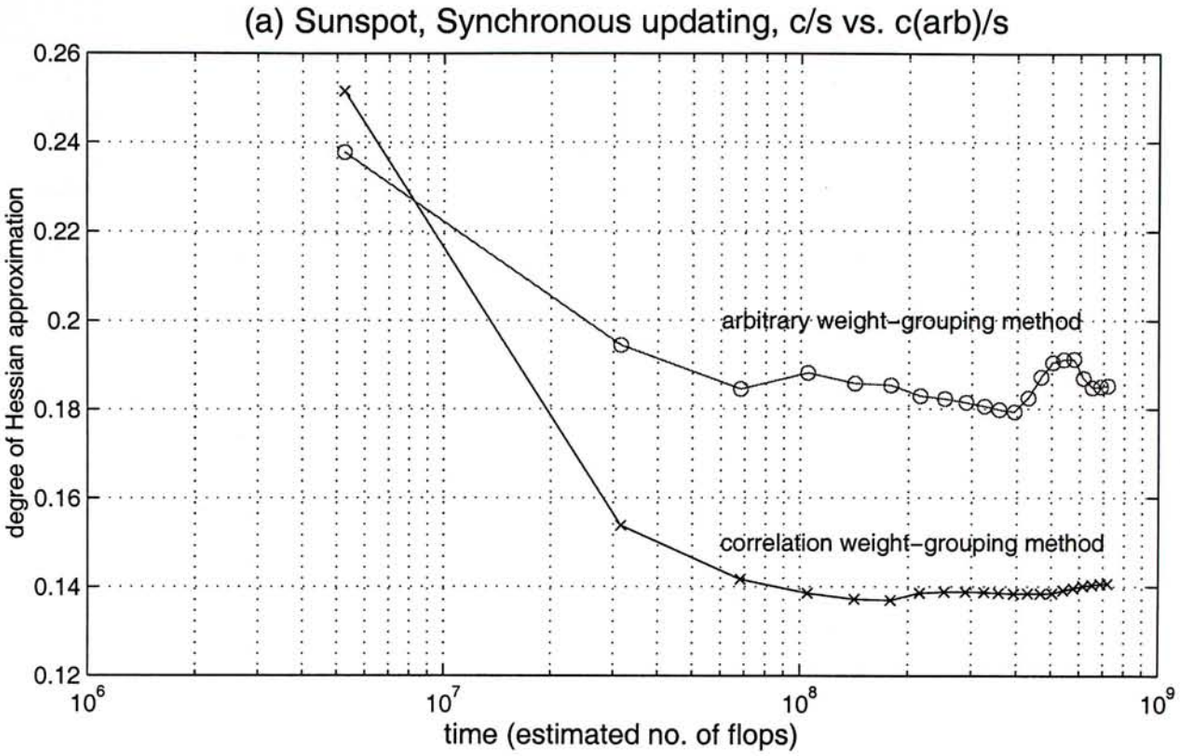


Figure 6-14 Degree of Hessian approximation during the whole training period. It is measured in terms of sum of the absolute values of all the Hessian elements relative to that of the full Hessian matrix. The (a) synchronous updating method, (b) asynchronous updating method and the composite sine training data were used in these experiments. The layer and arbitrary weight-grouping methods are represented by the line types '—x—' and '—o—' respectively.



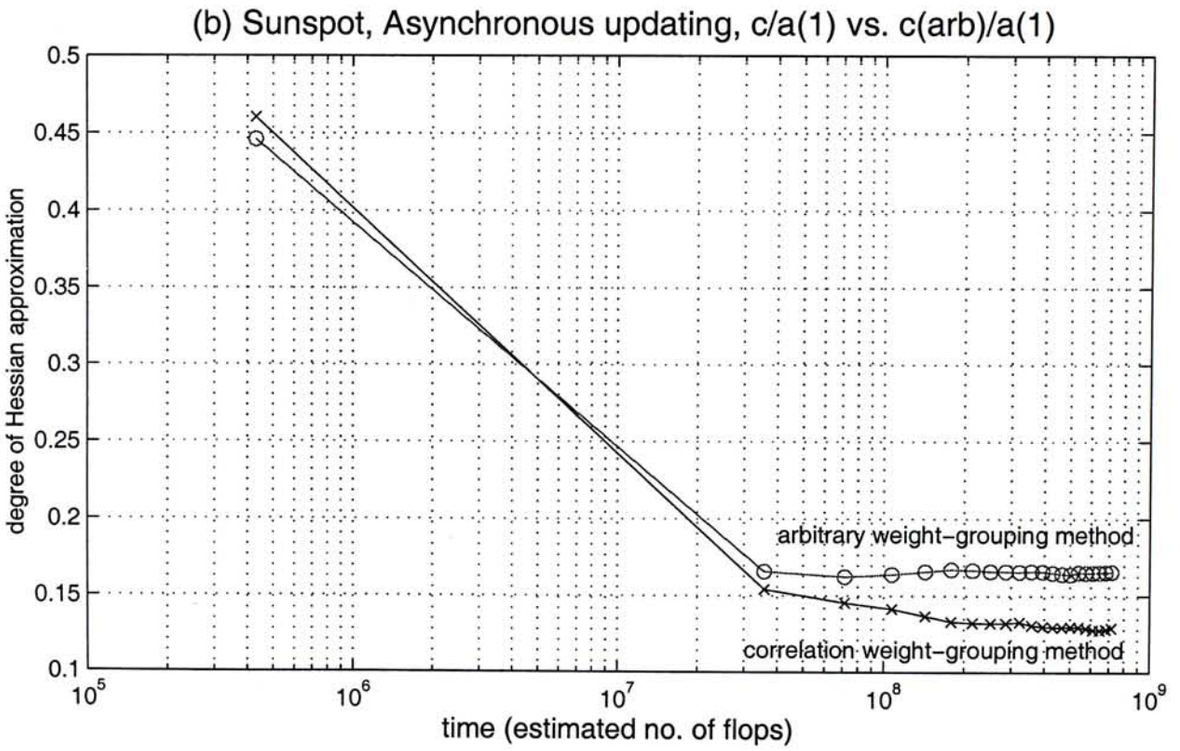
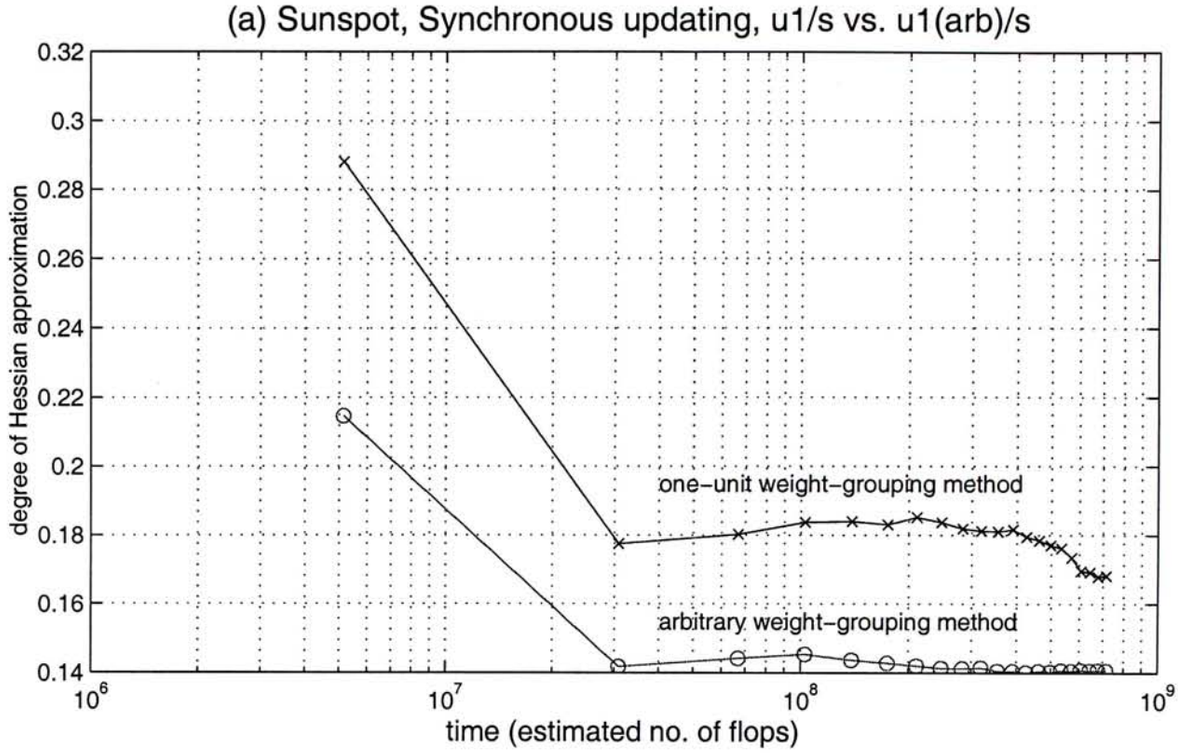


Figure 6-15 Degree of Hessian approximation during the whole training period. It is measured in terms of sum of the absolute values of all the Hessian elements relative to that of the full Hessian matrix. The (a) synchronous updating method, (b) asynchronous updating method and the sunspot training data were used in these experiments. The correlation and arbitrary weight-grouping methods are represented by the line types '—x—' and '—o—' respectively.



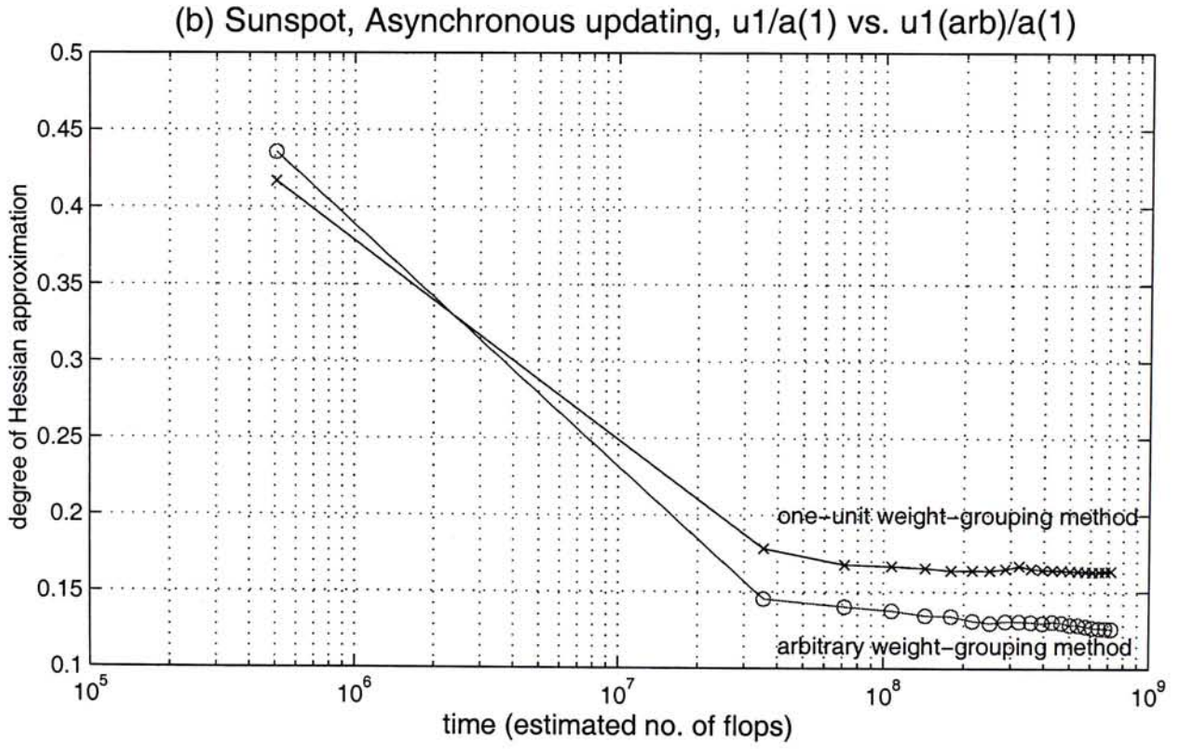


Figure 6-16 Degree of Hessian approximation during the whole training period. It is measured in terms of sum of the absolute values of all the Hessian elements relative to that of the full Hessian matrix. The (a) synchronous updating method, (b) asynchronous updating method and the sunspot training data were used in these experiments. The one-unit and arbitrary weight-grouping methods are represented by the line types '—x—' and '—o—' respectively.

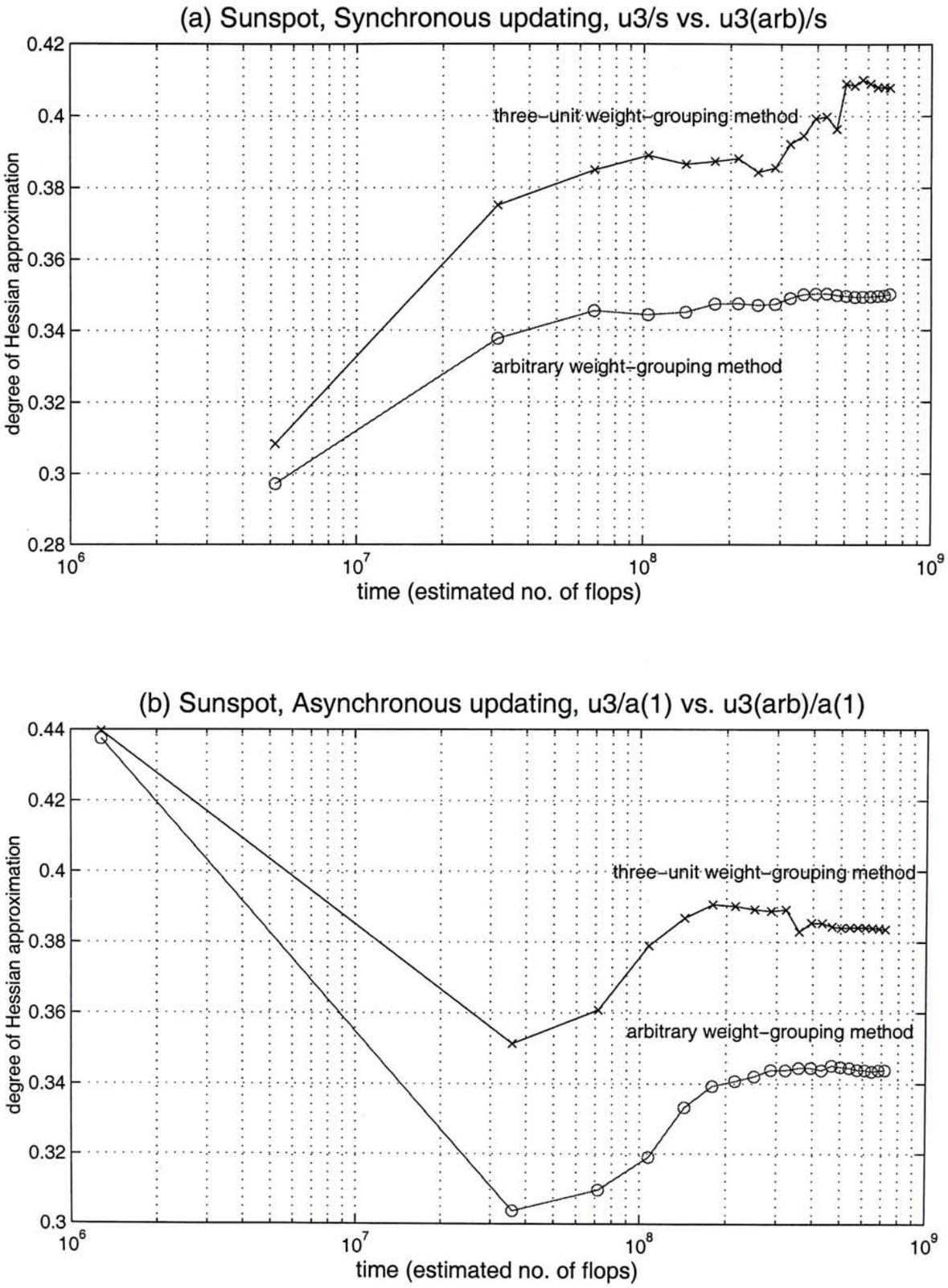
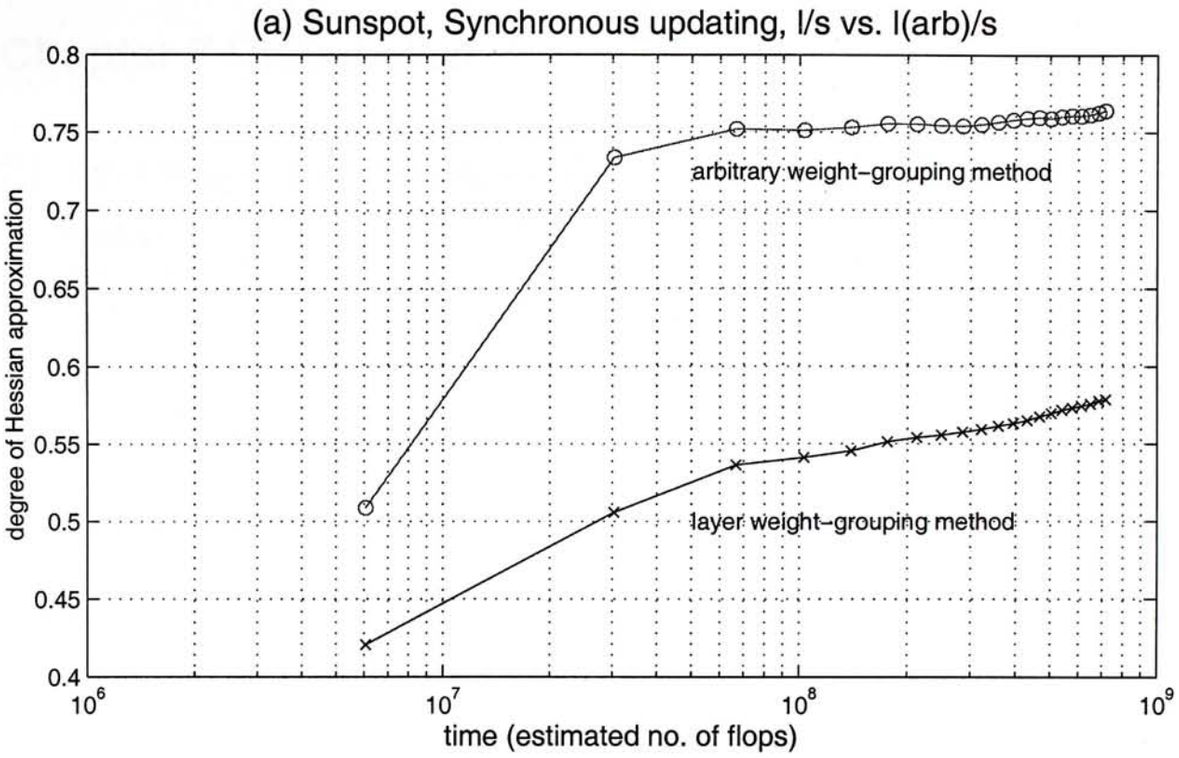


Figure 6-17 Degree of Hessian approximation during the whole training period. It is measured in terms of sum of the absolute values of all the Hessian elements relative to that of the full Hessian matrix. The (a) synchronous updating method, (b) asynchronous updating method and the sunspot training data were used in these experiments. The three-unit and arbitrary weight-grouping methods are represented by the line types '—x—' and '—o—' respectively.



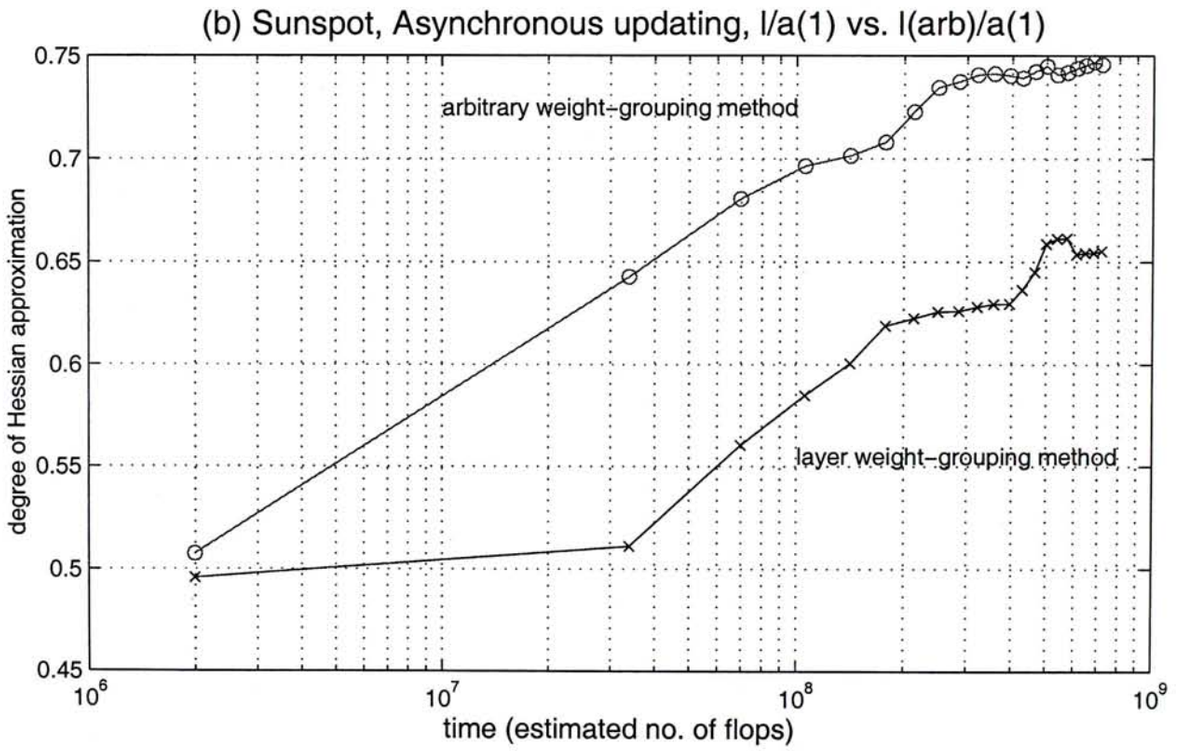


Figure 6-18 Degree of Hessian approximation during the whole training period. It is measured in terms of sum of the absolute values of all the Hessian elements relative to that of the full Hessian matrix. The (a) synchronous updating method, (b) asynchronous updating method and the sunspot training data were used in these experiments. The layer and arbitrary weight-grouping methods are represented by the line types '—x—' and '—o—' respectively.

Chapter 7 Discussion

7.1 Advantages and disadvantages of using block-diagonal Hessian matrix

In the following, the advantages and disadvantages of using block-diagonal Hessian matrix in terms of the computation complexity, storage complexity, training time performance and ease of use are listed.

7.1.1 Advantages

The advantages of using block-diagonal Hessian matrix are as follows.

i. First, the computation complexity is reduced. The computation load of calculating a block-diagonal Hessian matrix is less than that of calculating a full Hessian matrix. Moreover, if the block-diagonal Hessian matrix is used, the original system of updating equations can be decomposed into a number of smaller systems of updating equations. The computation load of solving this set of systems of updating equations is also less than that of solving the original system of updating equations. A detailed analysis is given in Section 7.2

ii. Second, the storage complexity is reduced. The Hessian matrix requires most space. Its storage complexity is $O(N^4)$. If we only store the non-zero Hessian elements, the storage complexity of a block-diagonal Hessian matrix ranges from $O(N^4)$ to $O(N^2)$. The details are explained in Section 7.3

iii. Third, the weights of different blocks can be calculated and updated in parallel if the synchronous updating method is used. The implementation is detailed in Section 7.4.

iv. Last, our experiments showed that the training time performance of the asynchronous updating with constraint methods under various block-diagonal Hessian matrices was better than that of the original method with full Hessian matrix. Moreover, when the networks were trained to predict the single sine or sunspot data, the training time performance of the asynchronous updating with constraint method was significantly better.

7.1.2 Disadvantages

The disadvantages of using block-diagonal Hessian matrix are as follows.

i. The first disadvantage is the choice of the value of maximum allowed weight change. This choice affects both the training time and generalization performance. One may spend some time tuning this parameter. In our experiments, the half of the 10 to 90 percentile range of the weights was close to the best choice.

   An alternative implementation of weight change constraint is given in Section 7.5.

ii. The second disadvantage is the choice of block-diagonal Hessian matrices. There are many different block-diagonal Hessian matrices and we lack prior knowledge on the choice of these matrices. Finding the one with the best performance takes us an unmanageable amount of time. Our experiments showed that block-diagonal Hessian matrices with small number of blocks had better training time performance. Regarding the weight-grouping methods, sub-network weight-grouping method was the best. However, we had only studied a few of them and many of them were ignored.

7.2 Analysis of computation complexity

In this section, the computation complexities of the original Levenberg-Marquardt method with full Hessian matrix described in Section 1.3.2.2, asynchronous updating method with the block-diagonal Hessian matrix described in Section 4.3 and synchronous updating method with the block-diagonal Hessian matrix described in Section 4.4.2 are analyzed.

The computation-intensive steps of these training methods are the calculations of Jacobian matrix **J**, Hessian matrix **H**, cost function E and weight update vector $\Delta\mathbf{w}$ (solving systems of updating equations)

The computation complexities of these computation-intensive steps are shown in Tables 7-1 and 7-2. We assume that the number of hidden units is much larger than the numbers of input and output units. The computation complexities are expressed in terms of the number of hidden units N and the length of the time series

T. Among the four steps, a tick is placed beside the step with the largest computation complexity, which is the computation complexity of the whole algorithm.

Mode of updating	Calculation	Computation complexity						
		Original method with full Hessian matrix			Asynchronous or synchronous updating method with block-diagonal Hessian matrix			
					the block-diagonal matrix is partitioned into O(N) blocks		the block-diagonal matrix is partitioned into $O(N^2)$ blocks	
			$T \le N^2$	$T \ge N^2$				
Batch	Jacobian matrix	$O(N^4T)$		✓	$O(N^4T)$	✓	$O(N^4T)$	✓
	Hessian matrix	$O(N^4T)$		✓	$O(N^3T)$		$O(N^2T)$	
	Cost function	$O(N^2T)$			$O(N^3T)$		$O(N^4T)$	✓
	Weight update vector by solving systems of equations	$O(N^6)$	✓		$O(N^4)$		$O(N)$	
Sequential	Jacobian matrix	$O(N^4)$			$O(N^4)$	✓	$O(N^4)$	✓
	Hessian matrix	$O(N^4)$			$O(N^3)$		$O(N^2)$	
	Cost function	$O(N^2)$			$O(N^3)$		$O(N^4)$	✓
	Weight update vector by solving systems of equations	$O(N^6)$	✓		$O(N^4)$	✓	$O(N)$	

Table 7-1 Computation complexities of the original method, asynchronous updating method and synchronous updating method. A tick is placed beside the step with the largest computation complexity. Jacobian matrix calculation that requires $O(N^4)$ computation per time step is used.

Mode of updating	Calculation	Computation complexity							
		Original method with full Hessian matrix			Asynchronous or synchronous updating method with block-diagonal Hessian matrix				
					the block-diagonal matrix is partitioned into O(N) blocks			the block-diagonal matrix is partitioned into $O(N^2)$ blocks	
			$T \le N^2$	$T \ge N^2$		$T \le N$	$T \ge N$		
Batch	Jacobian matrix	$O(N^3T)$			$O(N^3T)$		✓	$O(N^3T)$	
	Hessian matrix	$O(N^4T)$		✓	$O(N^3T)$		✓	$O(N^2T)$	
	Cost function	$O(N^2T)$			$O(N^3T)$		✓	$O(N^4T)$	✓
	Weight update vector by solving systems of equations	$O(N^6)$	✓		$O(N^4)$	✓		$O(N)$	
Sequential	Jacobian matrix	$O(N^3)$			$O(N^3)$			$O(N^3)$	
	Hessian matrix	$O(N^4)$			$O(N^3)$			$O(N^2)$	
	Cost function	$O(N^2)$			$O(N^3)$			$O(N^4)$	✓
	Weight update vector by solving systems of equations	$O(N^6)$	✓		$O(N^4)$	✓		$O(N)$	

Table 7-2 Computation complexities of the original method, asynchronous updating method and synchronous updating method. A tick is placed beside the step with the largest computation complexity. Jacobian matrix calculation that requires $O(N^3)$ computation per time step is used.

The difference between Tables 7-1 and 7-2 is the calculations of Jacobian matrix. The computation complexities per time step of Jacobian matrix calculations shown in Tables 7-1 and 7-2 are $O(N^4)$ and $O(N^3)$ respectively. The RTRL-like calculation described in Section 1.4.2 is an example of Jacobian matrix calculation that requires $O(N^4)$ computation per time step. Examples of Jacobian matrix calculations that require $O(N^3)$ computation per time step are as follows.

i. One is the Green's function method [Sun92]. But, [Logar93] reported that the prediction error of using the Green's function method was larger than that of using the RTRL algorithm. Moreover, the difference of errors between these two methods increased as the prediction horizon increased.

ii. The other is the truncated back-propagation through time algorithm described in Section 1.5.2.1. Information older than O(N) time steps is ignored. This is suitable for time series with short time correlation.

The computation complexities of the asynchronous and synchronous updating methods depend on the number of blocks of the block-diagonal Hessian matrix. So, two conditions of the block-diagonal Hessian matrix are considered.

i. One is that the block-diagonal matrix is partitioned into O(N) blocks.

ii. The other one is that the block-diagonal matrix is partitioned into $O(N^2)$ blocks.

To simplify the analysis, we assume the block sizes of the block-diagonal Hessian matrix are approximately equal.

7.2.1 Trend of computation complexity of each calculation

If Tables 7-1 and 7-2 are read in row, these tables show the trend of computation complexity of each calculation as a function of number of blocks. As the number of blocks increases, both the computation complexities of Hessian matrix and weight update vector calculations decrease. However, the computation complexity of cost function calculation increases because the number of cost function calculations increases. Finally, the computation complexity of Jacobian matrix calculation is not affected by the number of blocks and remains the same.

7.2.2 Batch mode of updating

We first discuss the case of batch mode of updating, which is shown in the upper halves of Tables 7-1 and 7-2. In the original method with full Hessian matrix, the computation complexity of weight update vector calculation dominates when T is small relative to the number of weights ($\approx N^2$). If T is large, the computation complexity of Hessian matrix calculation dominates. This computation complexity is the same as that of gradient based method such as the RTRL algorithm. (The BPTT algorithm requires only $O(N^2T)$ operations. However, it is not suitable for large T since its memory requirement increases with T).

In the asynchronous or synchronous updating method with the block-diagonal Hessian matrix, the computation complexities of Jacobian matrix and cost function calculations become dominant as the number of blocks increases. These two calculations have influential effect on the computation complexity of the whole algorithm. Take the case when the block-diagonal Hessian matrix is partitioned into O(N) blocks as an example. If the computation complexity of Jacobian matrix calculation is $O(N^3T)$ instead of $O(N^4T)$ and $T \geq N$, the computation complexity of the whole algorithm is reduced from $O(N^4T)$ to $O(N^3T)$. The condition of $T \geq N$ is often satisfied in practice since T is normally much larger than N.

As the number of blocks increases, the computation complexity of the whole algorithm does not change on the condition that the computation complexity of the Jacobian matrix calculation is $O(N^4T)$. If the computation complexity of the Jacobian matrix calculation is $O(N^3T)$, the computation complexity of the whole algorithm first decreases and then increases. The lowest computation complexity is $O(N^3T)$ when the block-diagonal Hessian matrix is partitioned into O(N) block and $T \geq N$.

7.2.3 Sequential mode of updating

We then discuss the case of sequential mode of updating, which is shown in the lower halves of Tables 7-1 and 7-2. In the original method with full Hessian matrix, the computation complexity of weight update vector calculation dominates. This computation complexity is greater than that of the gradient based method such as the RTRL algorithm. If the block-diagonal Hessian matrix is used, the

computation complexity of the whole algorithm ranges from $O(N^6)$ to $O(N^4)$, which is comparable to that of the gradient based method.

The computation complexity of weight update vector calculation dominates when the number of blocks is small. When the number of blocks is large, the computation complexity of cost function calculation becomes dominant. The computation complexity of Jacobian matrix calculation is not dominant. The computation complexity of the whole algorithm remains the same whether the computation complexity of Jacobian matrix calculation is $O(N^3)$ or $O(N^4)$.

7.3 Analysis of storage complexity

In this section, the storage complexities of the original Levenberg-Marquardt method with full Hessian matrix described in Section 1.3.2.2, asynchronous updating method with the block-diagonal Hessian matrix described in Section 4.3 and synchronous updating method with the block-diagonal Hessian matrix described in Section 4.4.2 are analyzed.

Two sets of variables that require most space are the Hessian matrix **H** and the triply indexed set of values $\left\{\frac{\partial V_j(t)}{\partial R_{j'r'}}\right\}$. This set of values $\left\{\frac{\partial V_j(t)}{\partial R_{j'r'}}\right\}$ is used in the calculation of the Jacobian matrix **J** with respect to the recurrent weights. The calculation is shown in Equation 1.33.

The storage complexities of these two sets of variables are shown in Table 7-3. They are expressed in terms of the number of hidden units N. The table shows the cases of using the original method, asynchronous and synchronous updating methods. It also shows the cases of using full Hessian matrix and block-diagonal Hessian matrices which are partitioned into O(N) or $O(N^2)$ blocks. In this analysis, two assumptions have been made.

i. The number of hidden units is much larger than the numbers of input and output units.
ii. The block sizes of a block-diagonal Hessian matrix are approximately equal.

Storage	Storage complexity				
	Original method with full Hessian matrix	Asynchronous updating method with block-diagonal Hessian matrix	Synchronous updating method with block-diagonal Hessian matrix	Asynchronous updating method with block-diagonal Hessian matrix	Synchronous updating method with block-diagonal Hessian matrix
		the block-diagonal matrix is partitioned into O(N) blocks		the block-diagonal matrix is partitioned into $O(N^2)$ blocks	
$\left\{\frac{\partial V_j(t)}{\partial R_{j'r'}}\right\}$	$O(N^3)$	$O(N^2)$	$O(N^3)$	O(N)	$O(N^3)$
Hessian matrix	$O(N^4)$	$O(N^3)$		$O(N^2)$	

Table 7-3 Storage complexities of the original method, asynchronous updating method and synchronous updating method.

7.3.1 Trend of storage complexity of each set of variables

If Table 7-3 is read in row, this table shows the trend of storage complexity as a function of number of blocks. As the number of blocks increases, the storage complexity of Hessian matrix decreases. The storage complexity of $\left\{\frac{\partial V_j(t)}{\partial R_{j'r'}}\right\}$ also decreases if the asynchronous updating method is used. However, if the synchronous updating method is used, the storage complexity of $\left\{\frac{\partial V_j(t)}{\partial R_{j'r'}}\right\}$ remains the same.

7.3.2 Trend of overall storage complexity

As the number of blocks increases, the storage complexity of the whole algorithm decreases from $O(N^4)$ to $O(N^2)$ on the condition that the asynchronous updating method is used. If the synchronous updating method is used, the storage complexity decreases from $O(N^4)$ to $O(N^3)$.

7.4 Parallel implementation

The comparisons described in Section 4.5 showed that the synchronous updating method converged slower than the asynchronous updating method. However, it possesses a useful property that the asynchronous updating method does not have. This property is that the algorithm of the synchronous updating method is inherently parallel. For example, hidden states can be computed simultaneously during the cost function calculation. The elements of the Jacobian matrix can be computed in parallel if the RTRL-like calculation described in Section 1.4.2 is used. The Hessian elements can also be computed at the same time. Finally, weight updates of different blocks can be computed simultaneously. Therefore, the synchronous updating method is readily implemented in parallel processing hardware. Substantial training time can then be saved.

7.5 Alternative implementation of weight change constraint

In the asynchronous updating with constraint method described in Section 4.3, constraint is imposed on the weight update to avoid excessively large magnitude of weight update. Alternatively, this can be done by using regularizer in the form of magnitude of the change of the weight vector. Then the weight change magnitude can be controlled by the regularization parameter.

This regularization can be implemented easily in the Levenberg-Marquardt algorithm. The updating equation of Levenberg-Marquardt algorithm shown in Equation 1.24 is obtained by minimizing the modified error function $\tilde{E}$ shown in Equation 7.1 with respect to $\mathbf{w}_{new}$. (Refer to [Bishops95, Chan96] for the derivation of $\tilde{E}$.)

$$\tilde{E} = \frac{1}{2}\left\|\mathbf{e} + \mathbf{J}(\mathbf{w}_{new} - \mathbf{w}_{old})\right\|^2 + \lambda\left\|\mathbf{w}_{new} - \mathbf{w}_{old}\right\|^2 \qquad (7.1)$$

where $\mathbf{w}_{old}$ is the current weight vector

$\mathbf{w}_{new}$ is the new weight vector

$\mathbf{e}$ is the error vector

$\mathbf{J}$ is the Jacobian matrix

Equation 7.1 shows that the regularization used in the Levenberg-Marquardt algorithm is in the form of magnitude of the change of the weight vector and the

regularization parameter is λ. Therefore, the amount of weight change constraint can be controlled by the regularization parameter λ.

Chapter 8 Conclusions

The Levenberg-Marquardt method is a nonlinear least squares algorithm and can be used to train neural networks with cost functions in the form of sum-of-squares nonlinear functions. It utilizes the second-order information of the cost function to achieve fast convergence speed. However, the arithmetic operations and storage required to train recurrent networks are large. Training a recurrent network with the Levenberg-Marquardt algorithm requires $O(N^4T)$ operations per epoch in batch mode calculation (or $O(N^6)$ operations per time step in sequential mode calculation) and $O(N^4)$ storage, where N is the number of fully recurrent hidden units and T is the number of training data.

In this thesis, we proposed applying the Levenberg-Marquardt method with the block-diagonal Hessian matrix to the recurrent neural network training. This method requires less operations per epoch / time step (ranging from $O(N^3T)$ to $O(N^4T)$ in batch mode calculation or ranging from $O(N^4)$ to $O(N^6)$ in sequential mode calculation) and less storage (ranging from $O(N^2)$ to $O(N^4)$). Moreover, weight updates of different blocks can be calculated independently. This property makes the parallel processing possible.

We need to consider two factors when the block-diagonal Hessian matrix is applied. These factors are the choices of the block-diagonal Hessian matrices and weight updating methods. They are described as follows.

Regarding the first factor, the block-diagonal Hessian matrices depend on the number of blocks, sizes of the blocks and weight-grouping method. To simplify the study, we assumed that the block sizes of a block-diagonal Hessian matrix were approximately equal. In our experiments, choosing a block-diagonal Hessian matrix with small number of blocks and the sub-network weight-grouping method was the best. However, there are many possible block-diagonal Hessian matrices not evaluated in this thesis.

Regarding the second factor, two weight updating methods, namely the asynchronous and synchronous updating methods, are proposed. Asynchronous method updates weights of one block at a time while synchronous method updates weights of all blocks at a time. The asynchronous updating method converges faster than the synchronous updating method but we should impose constraint on each

weight to remedy the poor generalization problem generated by the large incoming weight magnitudes of the hidden units. Although the synchronous updating method converges slower, it is inherently parallel. It is readily implemented in parallel processing hardware. This property can save substantial training time.

References

[Angeline94] Peter J. Angeline, Gregory M. Saunders and Jordan B. Pollack, "An Evolutionary Algorithm that Constructs Recurrent Neural Networks," IEEE Trans. on Neural Networks, vol.5, no.1, p. 54-65, Jan. 1994.

[Battiti92] Roberto Battiti, "First- and Second-Order Methods for Learning: Between Steepest Descent and Newton's Method," Neural Computation 4, 141-166, 1992.

[Becker88] S. Becker and Y. Le Cun, "Improving the Convergence of Back-Propagation Learning with Second Order Methods," Proceedings of the 1988 Connectionist Models Summer School (Pittsburg 1988), eds. D. Touretzky, G. Hinton, and T. Sejnowski, 29-37. San Mateo: Morgan Kaufmann.

[Bengio89] Y. Bengio, R. Cardin, and R. De Mori, "Speaker Independent Speech Recognition with Neural Networks and Speech Knowledge," Advances in Neural Information Processing System II (Denver 1989), ed. D.S. Touretzky, 218-225. San Mateo: Morgan Kaufmann.

[Bishop95] Christopher M. Bishop, "Neural Networks for Pattern Recognition," New York: Oxford University Press, 1995.

[Catfolis93] T. Catfolis, "A Method for Improving the Real-Time Recurrent Learning Algorithm," Neural Networks, vol. 6, no. 6, p. 807-21, 1993.

[Chan87] L. W. Chan and F. Fallside, "An Adaptive Training Algorithm for Back-Propagation Networks," Computer Speech and Language, 2, 205-218, 1987.

[Chan95] L. W. Chan and Evan F. Y. Young, "Locally Connected Recurrent Networks," Technical Report CS-TR-95-10, Department of Computer Science and Engineering, The Chinese University of Hong Kong, 1995.

[Chan96] L. W. Chan, "Levenberg-Marquardt Learning and Regularization," Proceedings of the International Conference on Neural Information Processing, p. 139-44, vol. 1, Sept. 1996.

[Dennis83] J. E. Dennis and R. B. Schnabel, "Numerical Methods for Unconstrained Optimization and Nonlinear Equations," Prentice Hall, Englewood Cliffs, NJ. 1983.

[Dorffner96] G. Dorffner, "Neural Networks for Time Series Processing," Neural Network World, 6(4)447-468, 1996.

[Elman90] Jeffrey L. Elman, "Finding Structure in Time," Cognitive Science 14, 179-211, 1990.

[Fahlman88] Scott E. Fahlman, "Fast-Learning Variations on Back-Propagation: An Empirical Study," Proceedings of the 1988 Connectionist Models Summer School (Pittsburg 1988), eds. D. Touretzky, G. Hinton, and T. Sejnowski, 38-51. San Mateo: Morgan Kaufmann.

[Fahlman91] Scott E. Fahlman, "The Recurrent Cascade-Correlation Architecture," Technical Report CMU-CS-91-100, School of Computer Science, Carnegie Mellon University, 1991.

[Fogel91] D. B. Fogel, L. J. Fogel and V. W. Porto, "Evolutionary Methods for Training Neural Networks," IEEE Conference on Neural Networks for Ocean Engineering, p. 317-27, Aug. 1991.

[Frasconi92] P. Frasconi, M. Gori and G. Soda, "Local Feedback Multilayered Networks," Neural Computation 4, 120-130, 1992.

[Giles90] C. L. Giles, G. Z. Sun, H. H. Chen, Y. C. Lee and D. Chen, "Higher Order Recurrent Networks and Grammatical Inference," D. Tourezky, (Ed.), Advances in Neural Information Processing Systems 2, 380-387. San Mateo, CA: Morgan Kaufman, 1990.

[Hagan94] Martin T. Hagan and Mohammad B. Menhaj, "Training Feedforward Networks with the Marquardt Algorithm," IEEE Trans. on Neural Networks, vol. 5, no. 6, 989-993, November 1994.

[Hertz91] John Hertz, Anders Krogh and Richard G. Palmer, "Introduction to the Theory of Neural Computation," Wokingham, U. K.: Addison-Wesley Pub. Co., 1991.

[Horne95] Bill G. Horne and C. Lee Giles, 1995. "An Experimental Comparison of Recurrent Neural Networks," Neural Information Processing Systems 7, eds. G. Tesauro, D. Touretzky, T.Leen, p.697, MIT Press.

[Jordan86] Michael I. Jordan, "Attractor Dynamics and Parallelism in a Connectionist Sequential Machine," In Proceedings of the Eighth Annual Conference of the Cognitive Science Society (Amherst 1986), 531-546. Hillsdale: Erlbaum.

[Karunanithi92] N. Karunanithi and D. Whitley, "Prediction of Software Reliability Using Feedforward and Recurrent Neural Nets," IJCNN International Joint Conference on Neural Networks, p. 800-5 vol.1, June 1992.

[Koehn94] Philipp Koehn, "Combining Genetic Algorithms and Neural Networks: The Encoding Problem," MSc. Thesis, The University of Tennessee, Knoxville, December 1994.

[Kollias88] S. Kollias and D. Anastassiou, "Adaptive Training of Multilayer Neural Networks using a Least Squares Estimation Technique," IEEE International Conference on Neural Networks, vol.1, 383-90, 1988.

[Kollias89] S. Kollias and D. Anastassiou, "An Adaptive Least Squares Algorithm for the Efficient Training of Artificial Neural Networks," IEEE Transactions on Circuits and Systems, vol.36, no.8, 1092-101, 1989.

[Lawrence97] Steve Lawrence, Andrew D. Back, A. C. Tsoi and C. Lee Giles, "On the Distribution of Performance from Multiple Neural-Network Trials," IEEE Trans. on Neural Networks, vol. 8, no. 6, 1507-1517, November 1997.

[Li90] M. Li, K. Mehrotra, C. Mohan and S. Ranka, "Sunspot Numbers Forecasting Using Neural Networks," Proceedings 5th IEEE International Symposium on Intelligent Control 1990, p. 524-9 vol.1, Sept. 1990.

[Logar93] Antonette M. Logar, Edward M. Corwin and William J. B. Oldham, "A Comparison of Recurrent Neural Network Learning Algorithms," 1993 IEEE International Conference on Neural Networks, p. 1129-34, vol. 2, 1993.

[Mori93] H. Mori and T. Ogasawara, "A Recurrent Neural Network for Short-term Load Forecasting," ANNPS '93 Proceedings of the Second International Forum on Applications of Neural Networks to Power Systems, p. 395-400, April 1993.

[Mozer93] Michael C. Mozer, "Neural Net Architectures for Temporal Sequence Processing," A. Weigend & N. Gershenfeld (Eds.), Predicting the Future and Understanding the Past, Redwood City, CA: Addison-Wesley Publishing, 1993.

[Pedersen95] M. W. Pedersen and L. K. Hansen, "Recurrent Networks: Second Order Properties and Pruning," Advances in Neural Information Processing Systems 7, p. 673-680, 1995.

[Press88] William H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Wetterling, "Numerical Recipes in C: the Art of Scientific Computing," Cambridge: Cambridge University Press, 1988.

[Ricotti88] L. Prina Ricotti, S. Ragazzini and G. Martinelli, "Learning of Word Stress in a Sub-Optimal Second Order Back-Propagation Neural Network," IEEE International Conference on Neural Networks (San Diego 1988), vol. I, 355-361.

[Sarle95] Warren S. Sarle, "Stopped Training and Other Remedies for Overfitting," To appear in Proceedings of the 27th Symposium on the Interface, 1995.

[Schmidhuber92] Jürgen Schmidhuber, "A Fixed Size Storage O(n^3) Time Complexity Learning Algorithm for Fully Recurrent Continually Running Networks," Neural Computation, vol. 4, 243-248, 1992.

[Sharda90] R. Sharda and Rajendra B. Patil, "Neural Networks as Forecasting Experts: an Empirical Test," International Joint Conference on Neural Networks, vol. 2, p. 491-494, Washington, D. C., 1990.

[Shepherd97] A. J. Shepherd, "Second-Order Methods for Neural Networks: Fast and Reliable Training Methods for Multi-Layer Perceptrons," Springer, 1997.

[Shimohara88] K. Shimohara, T. Uchiyama, and Y. Tokunaga, "Back-Propagation Networks for Event-Driven Temporal Sequence Processing," IEEE International Conference on Neural Networks (San Diego 1988), vol. I, 665-672. New York: IEEE, 1988.

[Sun92] G. Z. Sun, H. H. Chen and Y. C. Lee, "Green's Function Method for Fast On-line Learning Algorithm of Recurrent Neural Networks," In Advances in Neural

Information Processing Systems 4, J. E. Moody, S. J. Hanson and R. P. Lippmann, Eds. San Mateo, CA: Morgan Kaufmann, p. 333-340, 1992.

[Tang93] Z. Tang and Paul A. Fishwick, "Feedforward Neural Nets as Models for Time Series Forecasting," ORSA Journal on Computing, vol. 5, no. 4, p. 374-85, Fall 1993.

[Tsoi94] A. C. Tsoi and Andrew D. Back, "Locally Recurrent Globally Feedforward Networks: a Critical Review of Architectures," IEEE Trans. on Neural Networks, vol. 5, no. 2, pp. 229-239, March 1994.

[Ulbricht92] C. Ulbricht, G. Dorffner, S. Canu, D. Guillemyn, G. Marijuan, J. Olarte, C. Rodriguez and I. Martin, "Mechanisms for Handling Sequences with Neural Networks," In C. H. Dagli, et al. (eds.), Intelligent Engineering Systems through Artificial Neural Networks, Proceedings of the ANNIE'92 Conference, St. Louis, Missouri, USA, Volume 2, ASME Press, New York, 1992.

[Weigend90] Andreas S. Weigend, Bernardo A. Huberman and David E. Rumelhart, "Predicting the Future: a Connectionist Approach," International Journal of Neural Systems, vol. 1, no. 3, p. 193-209, 1990.

[Wille97] J, Wille, "On the Structure of the Hessian Matrix in Feedforward Networks and Second Derivative Methods," Proceedings of the 1997 IEEE International Conference on Neural Networks, vol.3, 1851-5.

[Williams89] Ronald J. Williams, and David Zipser, "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks," Neural Computation 1, 270-280, 1989.

[Williams90] Ronald J. Williams and J. Peng, " An Efficient Gradient-based Algorithm for On-line Training of Recurrent Network Trajectories," Neural Computation 2, 490-501, 1990.

[Wilson95] William H. Wilson, "Stability of Learning in Classes of Recurrent and Feedforward Networks," Proceedings of the Sixth Australian Conference on Neural Networks, 142-145, 1995.

[Zipser89] David Zipser, "A Subgrouping Strategy that Reduces Complexity and Speeds Up Learning in Recurrent Networks," Neural Computation 1, 552-558, 1989.