A FEED FORWARD NEURAL NETWORK APPROACH FOR MATRIX COMPUTATIONS

A thesis submitted for the degree of Doctor of Philosophy

by

Ali F. Al-Mudhaf

Department of Mechanical Engineering

Brunel University

January 2001

ABSTRACT

A Feed Forward Neural Network Approach for Matrix Computations

A new neural network approach for performing matrix computations is presented. The idea of this approach is to construct a feed-forward neural network (FNN) and then train it by matching a desired set of patterns. The solution of the problem is the converged weight of the FNN. Accordingly, unlike the conventional FNN research that concentrates on external properties (mappings) of the networks, this study concentrates on the internal properties (weights) of the network. The present network is linear and its weights are usually strongly constrained; hence, complicated overlapped network needs to be construct. It should be noticed, however, that the present approach depends highly on the training algorithm of the FNN. Unfortunately, the available training methods; such as, the original Back-propagation (BP) algorithm, encounter many deficiencies when applied to matrix algebra problems; e.g., slow convergence due to improper choice of learning rates (LR). Thus, this study will focus on the development of new efficient and accurate FNN training methods. One improvement suggested to alleviate the problem of LR choice is the use of a line search with steepest descent method; namely, bracketing with golden section method. This provides an optimal LR as training progresses. Another improvement proposed in this study is the use of conjugate gradient (CG) methods to speed up the training process of the neural network.

The computational feasibility of these methods is assessed on two matrix problems; namely, the *LU*-decomposition of both band and square ill-conditioned unsymmetric matrices and the inversion of square ill-conditioned unsymmetric matrices. In this study, two performance indexes have been considered; namely, learning speed and convergence accuracy. Extensive computer simulations have been carried out using the following training methods: steepest descent with line search (SDLS) method, conventional back propagation (BP) algorithm, and conjugate gradient (CG) methods; specifically, *Fletcher-Reeves* conjugate gradient (CGFR) method and *Polak-Ribière* conjugate gradient (CGPR) method.

The performance comparisons between these minimization methods have demonstrated that the CG training methods give better convergence accuracy and are by far the superior with respect to learning time; they offer speed-ups of anything between 3 and 4 over SDLS depending on the severity of the error goal chosen and the size of the problem. Furthermore, when using *Powell*'s restart criteria with the CG methods, the problem of wrong convergence directions usually encountered in pure CG learning methods is alleviated. In general, CG methods with restarts have shown the best performance among all other methods in training the FNN for *LU*-decomposition and matrix inversion. Consequently, it is concluded that CG methods are good candidates for training FNN of matrix computations, in particular, *Polak-Ribière* conjugate gradient method with *Powell*'s restart criteria.

TABLE OF CONTENTS

Page

Acknowledgement v

List of Tables vi

List of Figures vii

Chapter

1 INTRODUCTION 1

1.1 Rationale of the problem 1

1.2 Literature Survey 4

1.2.1 Recurrent Neural Networks (RNN) for Matrix Computations 4

1.2.2 Feed-forward Neural Networks (FNN) for Matrix Computations 5

1.2.3 Training Algorithm for FNN 5

1.3 Present Work 9

2 FEED-FORWARD NEUTRAL NETWORK AND THE BACK-PROPAGATION ALGORITHM 11

2.1 Structure of Feed-forward Network 11

2.2 A Functional Viewpoint of FNN Architecture and Mappings 12

2.3 Training Algorithm 14

2.3.1 Steepest Descent Method 15

2.3.2 Pattern Presentation and Weight-Updating Strategies 17

2.3.2.1 Training by Sample 17

2.3.2.2 Training by Epoch (Batch) 18

2.3.2.3 Mixed Strategies 18

2.3.3 The Search for $\underline{w}$ 19

2.3.4 Error Surface 20

2.4 An overview of the Back-propagation (BP) Algorithm 21

2.5 Comments on the BP Training Algorithm 21

2.5.1 Error Trajectories During Training 21

2.5.2 Back-propagation Starting Points 22

2.5.3 Effect of the Error Surface on the BP Algorithm 22

2.5.4 Learning Rate (LR) 22

2.6 Design of Training Strategy 23

3 MINIMIZATION METHODS FOR TRAINING FNN................ 32
3.1 Taylor Series Expansion for the Epoch Mapping Error 32
3.2 First-Order Methods.. 33
3.2.1 Limitations of First Order (Steepest Descent)Methods 34
3.3 Steepest Descent with a Line Search Method 34
3.3.1 A Hybrid Line Search Method 36
3.3.1.1 Golden Section Search Algoritm 36
3.3.1.2 Bracketing Search Algoritm 38
3.4 BP Training Procedure with Momentum 38
3.4.1 Significant Aspects of Momentum 39
3.5 Second-Order Methods 40
3.5.1 Approximation of Local Shape of the Epoch Mapping Error... 40
3.5.2 Second-order Equations and the Hessian Matrix................ 40
3.5.3 Conjugate Gradient (CG) Methods 41
3.5.4 Comments on CG Methods 44
3.5.5 Relation between CG Method and Momentum Formulations.. 46
4 FNNs FOR LARGE-SCALE MATRIX ALGEBRA PROBLEMS 50
4.1 FNN for *LU*-Decomposition Problem 50
4.1.1 Training Formulation for *LU*-Decomposition FNN under SDLS Method.. 52
4.1.2 Training Procedure for *LU*-Decomposition FNN under SDLS Method 53
4.1.3 Training Formulation for *LU*-Decomposition FNN under CG Methods 54
4.1.4 Training Procedure for *LU*-Decomposition FNN under CG Methods 55
4.2 FNN for the Inversion of Large Scale Matrix 56
4.2.1 Training Formulation for Matrix Inversion FNN under SDLS Method.. 57
4.2.2 Training Procedure for Matrix Inversion FNN under SDLS Method 58

4.2.3 Training Formulation for Matrix Inversion FNN under CG Methods 59
4.2.4 Training Procedure for Matrix Inversion FNN under CG Methods 59
4.3 Potential for Parallelisation 61
4.3.1 Batch (Epoch) Gradient Calculation 61
4.3.2 Step Size Calculation 62
5 **SIMULATION RESULTS AND DISCUSSION** **70**
5.1 An Assessment of Conjugate Gradient Minimization Methods 71
5.2 The *LU*-Decomposition Problem 73
5.2.1 *LU*-Decomposition of Large Square Matrices 74
5.2.2 *LU*-Decomposition of Large Band Matrices 77
5.2.2.1 *LU*-Decomposition for Band Matrices of Same Bandwidth 79
5.2.2.2 *LU*-Decomposition for Band Matrices of different Bandwidth 83
5.3 The Matrix Inversion Problem 86
6 **CONCLUSIONS AND RECOMMENDATIONS** **121**
REFERENCES **126**

ACKNOWLEDGEMENT

I would like to express my sincerest appreciation to Professor Ibrahim Esat, for his notable contribution to this thesis. His clear and incisive view of the progress of this research was a great help to me in assessing and reviewing the work undertaken.

The successful completion of this research work much owes to Dr. Humood F. Al-Mudhaf, Director General of the Public Authority of Applied Education and Training, for his encouragement, scholarship and moral support.

Last but not least, I wish to express my gratitude to my father, my wife Nahed, and the rest of my family for their continued support and encouragement throughout this research work.

LIST OF TABLES

Table		Page
5.1	A Definition of training tests of LU-decomposition of square matrices	74
5.2	Detailed simulation results for Case 1 under four training methods	75
5.3	Detailed simulation results for Case 2 under four training methods	75
5.4	Detailed simulation results for Case 3 under four training methods	75
5.5	Detailed simulation results for Case 4 under four training methods	76
5.6	A Definition of training tests of LU-decomposition of band matrices	78
5.7	Detailed simulation results for Case 5 under four training methods	81
5.8	Detailed simulation results for Case 6 under four training methods	81
5.9	Detailed simulation results for Case 7 under four training methods	81
5.10	Detailed simulation results for Case 8 under four training methods	82
5.11	Detailed simulation results for Case 9 under four training methods	85
5.12	Detailed simulation results for Case 10 under four training methods	85
5.13	Detailed simulation results for Case 11 under four training methods	85
5.14	A Definition of training tests for the matrix inversion problem	86
5.15	Detailed simulations results for Case 12 under four training methods	88
5.16	Detailed simulations results for Case 13 under four training methods	88
5.17	Detailed simulations results for Case 14 under four training methods	89
5.18	Detailed simulations results for Case 15 under four training methods	89
5.19	Detailed simulations results for Case 16 under four training methods	89
5.20	Detailed simulations results for Case 17 under four training methods	90

LIST OF FIGURES

Figure		Page
2.1	Some problem-dependent design issues	25
2.2	Overall FNN-based strategy (implementation and training)	26
2.3	Structure of a feed forward neural network	27
2.4	A three layered neural network	27
2.5	Illustration of the update equation for the output layer weights	28
2.6	Illustration of the update equation for the hidden layer weights	29
2.7	The overall procedure for backpropagation(BP) algorithm	30
2.8	Possible minima of $E(w)$ found in training	31
2.9	Possible weight trajectories during training	31
2.10	Possible error evolution during training	31
3.1	Two very different trajectories for weight correction	47
3.2	Conjugate versus gradient directions weight correction: (a) Gradient guided search with constant step size; (b) Conjugate directions leading directly to the minimum	47
3.3	A typical of unimodal function	48
3.4	Evaluating the objective function at two intermediate points	48
3.5	The case when $E(a_1) < E(b_1)$; the minimizer $\underline{w}^* \in [a_o, b_1]$	49
4.1	Structured network for *LU*-decomposition	64
4.2	Flow chart of the SDLS method in case of *LU*-decomposition	65
4.3	Flow chart of the CG methods in case of *LU*-decomposition	66
4.4	Structured network for matrix inversion	67
4.5	Flow chart of the SDLS method in case of matrix inversion	68
4.6	Flow chart of the CG methods in case of matrix inversion	69
5.1	Performance of the conjugate gradient training for various reset values.	92
5.2	Optimal step size behavior for the *LU*-decomposition of a band matrix (19 x 195) under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.	93
5.3	Performance comparison of training methods for the *LU*-decomposition of square matrices having different dimensions.	94

5.4	Performance of training methods as a function of time for the *LU*-decomposition of square matrices having different dimensions.	95
5.5	Illustration of optimal step size behavior for Case 1 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.	96
5.6	Effect of matrix size on the performance of training methods used for *LU*-decomposition of square matrices.	97
5.7	Illustration of optimal step size behavior for Case 2 under different training methods: (a) CCPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.	98
5.8	Illustration of optimal step size behavior for Case 3 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.	99
5.9	Illustration of optimal step size behavior for Case 4 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.	100
5.10	Performance of training methods as a function of time for the *LU*-decomposition of band matrices having the same band-width.	101
5.11	Performance of training methods as a function of time for the *LU*-decomposition of band matrices having different band-width.	102
5.12	Effect of matrix size on the performance of four training methods used for the *LU*-decomposition of band matrices of same band-width.	103
5.13	Illustration of optimal step size behavior for Case 5 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.	104
5.14	Illustration of optimal step size behavior for Case 6 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.	105
5.15	Illustration of optimal step size behavior for Case 7 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.	106
5.16	Illustration of optimal step size behavior for Case 8 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.	107
5.17	Effect of matrix size of the performance of training methods used for the *LU*-decomposition of band matrices of different bandwidth.	108
5.18	Illustration of optimal step size behavior for Case 9 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.	109

5.19 Illustration of optimal step size behavior for Case 10 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS. 110

5.20 Illustration of optimal step size behavior for Case 11 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS. 111

5.21 Performance of training methods as a function of time for the inversion of square matrices having different dimensions. 112

5.22 Illustration of optimal step size behavior for Case 12 under different training methods:(a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS. 113

5.23 Illustration of optimal step size behavior for Case 13 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS. 114

5.24 Illustration of optimal step size behavior for Case 14 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS. 115

5.25 Illustration of optimal step size behavior for Case 15 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS. 116

5.26 Illustration of optimal step size behavior for Case 16 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS. 117

5.27 Illustration of optimal step size behavior for Case 17 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS. 118

5.28 Effect of matrix size on the performance of training methods used for the inversion of square matrices. 119

5.29 Effect of matrix size on the convergence history of training methods used for the inversion of square matrices. 120

CHAPTER 1

Introduction

1.1 Rationale of The Problem

Matrix computations are very important and basic problems in science and engineering. The existing technologies for these problems can be classified into: (1) traditional algorithm-based methods [1-3]; and, (2) parallel processing approaches, based on (very large scale integration) VLSI-inspired systolic and wave-front array processors [4-6]. Although these methods are successful in solving a lot of practical problems, they do have some limitations and disadvantages.

The main disadvantages of the traditional algorithm-based approaches are: (1) they are not parallel algorithms and cannot be realized directly by VLSI hardwares, so that the speed of processing is quite limited; and, (2) divisions are usually involved in the calculations, so that correct solutions cannot be obtained for ill-conditioned matrices.

For the parallel processing approaches, although the first disadvantage of the algorithm-based approaches is overcome, the second disadvantage still exists. This is because the key procedure of the parallel processing approaches is to modify and decompose the traditional algorithms to make them suitable for VLSI architectures and realizations [4-6]. Consequently, the calculations performed by these two approaches are the same when viewed from a higher conceptual level, i.e., the difference is only in the specific ways to realize the calculations. For example, the parallel approach for matrix inversion described in [6] is still based on Gaussian elimination; hence, the parallel processing approaches still do not work for ill-conditioned matrices.

The parallel processing approaches have another weak point; they do not give a general framework for matrix algebra problems, i.e., they are quite problem dependent. This is because the first step of the parallel approaches is to obtain a set of parallel equations for the specific problem, and then to use parallel hardware to perform the calculations of these

equations. Hence, from a general conceptual point of view, the parallel processing approach is not a unified method for matrix algebra problems.

Furthermore, it is often required to compute the inverse of a matrix as fast as possible for real time applications such as adaptive signal processing [7] and adaptive spectral estimation [8]. From a practical viewpoint, it is difficult to deliver the desired real time performance if conventional digital and sequential computations methods are used, especially for a matrix with large dimension, because the inversion of a matrix is computationally intensive.

Therefore, it is desirable to construct a computation model for matrix algebra problems that enjoys the following features: asynchronous parallel processing and continuous-time dynamics. Fortunately, neural networks have been found [13-24] to be good candidates for such models. This is because they provide a novel type of parallel processing that has powerful capabilities and potential for creative hardware implementations, meet the demand for fast computing hardware, and provide the potential for solving matrix application problems. Neural networks utilize a parallel processing structure that has large numbers of processors and many interconnections between them. These processors are much simpler than typical central processing units (CPUs). In a neural network each processor is linked to many of its neighbors (typically hundreds or thousands) so that there are many more interconnects than processors. Hence, the power of the neural network lies in the tremendous number of interconnections.

Two main approaches are found in the literature to perform the above matrix computations. The first approach is based on feed-forward neural network (FNN), which is the most popular neural network model that has been playing a central role in applications of neural networks. The second approach is based on recurrent neural networks now called Hopfield models. In the latter approach, the neural networks are considered as dynamic systems that have a mapping error function to be minimized as time evolves. The steady state solution of the system gives the solution to the matrix problem.

On the other hand, the basic idea of the first approach is to represent a given matrix algebra problem by a linear FNN so that if the network matches a set of desired patterns, the weights of the linear neurons give the solution to the problem. Hence, the matrix problem solution is a byproduct of the training process and is obtained once the training of the networks is performed successfully. In other words, the matrix computation in this approach depends highly on the success and efficiency of the training algorithm of FNN. This approach will be adopted in the present study.

Back-propagation (BP) algorithm is currently the most widely used training algorithm in FNN applications. Its popularity can be attributed primarily to the fact that this algorithm, in conjunction with a three layer feed-forward architecture, is capable of approximating to any degree of accuracy, any reasonable arbitrary nonlinear input-output mapping, provided that the neural network has a sufficient number of hidden neurons. Basically the BP algorithm is nothing but a descent algorithm, which attempts to minimize the difference (or error) between the desired and actual outputs of the network in an iterative manner. For each iteration, the algorithm adjusts the weights involved in the network so as to make the error decreasing along a descent direction. In doing so, many parameters are introduced in the literature for controlling the size of weight adjustment along the descent direction and for dampening oscillations of the iterations.

Despite its popularity, the BP algorithm has some drawbacks, for example, converging to local minima instead of the global minimum, temporal instability and poor scaling properties. However, its main difficulty is slow convergence, which is a typical problem for simple gradient descent methods. Therefore, there is a need for developing an efficient training method that could overcome the aforementioned drawbacks when applying the algorithm to matrix computations.

In the next section, available models for matrix computations via neural networks are presented along with various existing improvements to the BP algorithm.

1.2 Literature Survey

As mentioned in the previous section, the neural networks that have been used recently for matrix computations are the recurrent networks (RNN) and the feed-forward networks (FNN). The available literature on these models as well as the back propagation training algorithm of FNN are reviewed below:

1.2.1 Recurrent Neural Networks (RNN) for Matrix Computations

In a breakthrough paper published in 1982, John Hopfield [9] introduced the network architecture that has come to be known as the Hopfield model. In his work, he also introduced outer product rules as well as equivalent approaches based on the early work of Hebb [10] for training a class of recurrent networks [11]. Hopfield and Tank have applied neural network for solving optimization problems [12]. Since their work, recurrent neural networks have been developed for solving numerous optimization and constraint satisfaction problems.

Recently, recurrent neural networks have been proposed for solving a wide variety of matrix algebra problems. Wang and Wu [13] presented two recurrent neural networks for *LU*-decomposition and Cholesky factorization. These networks consist of two bi-directionally connected layers and each layer consists of an array of neurons. They proved that these networks are asymptotically stable in the large and capable of *LU*-decomposition and Cholesky factorization.

Later, Jang et al [14] and Luo and Zheng [15] used modified Hopfield networks for matrix inversion. They showed both analytically and by simulations that this network is guaranteed to be stable and could perform the inversion within an elapsed time of a few characteristic constants of the network. However, their examples cannot be generalized. Differing from Luo and Zheng, Wang [16] developed a linear recurrent neural network for matrix inversion, which can be decomposed into n independent sub-networks and can be easily implemented in an electronic circuit.

Recurrent Neural Networks have also been proposed for solving real-time problems involve simultaneous linear equations [17]. Wang [18] proposed an electronic neural

network that was able to generate real time solutions to large-scale simultaneous linear equations. Its application includes image processing, experimentation of the neural network to image processing, experimentation of the neural network using IC chips and discrete components, and the implementation of the neural network in analogue VLSI circuits. Carvalho and Barbosa [19] proposed a neural network model comprising linear neurons, tailored to solving linear system of algebraic equations by the minimization of a quadratic error function. Simulations showed that the neural system had excellent stability properties and that consistent or inconsistent linear systems can be solved fast. Along the same line, Cichocki and Unbehauden [20] proposed various circuit architectures of simple neuron-like analog processors for an on-line solving of a system of linear equations with real constant.

1.2.2 Feed-forward Neural Networks (FNN) for Matrix Computations

Recently, a so-called structured network was developed for solving a wide variety of matrix algebra problems in massively parallel fashion. Structured networks are feed-forward neural networks with linear neurons that have a special training algorithm. Wang and Mendel [21-23] used two-dimensional structured networks for solving linear equations. Because of the network being 2-D, the massively parallel processing advantage of it wasn't fully utilized. Wang and Mendel [24] extended the method of [21-23] to three-dimensional (3-D) structured networks. The proposed structure introduced more parallelism into the processing, but they were more complicated than 2-D structured networks. In their study, Wang and Mendel [23-24] introduced a modified version of the BP algorithm for training structured networks. Unfortunately, this method has been developed based on a heuristic argument in a somewhat *ad hoc* manner, and consequently, it cannot be generalized for different matrix computations.

1.2.3 Training Algorithm For FNN

As pointed out earlier, the training algorithm is one of the key elements when solving matrix algebra problems using FNN. Currently, Back propagation algorithm is the most widely used training algorithm in FNN applications. Park [25] rediscovered the Back propagation training technique after Werbos [26] developed the work of Madaline. In 1985,

Parker published a report on this algorithm at MIT [27]. Not long after Parker published his findings, Rumelhart, Hinton and Williams [28] modified the technique and succeeded in making it widely known. Despite these modifications, the BP algorithm still encounters two difficulties in practice; firstly, the convergence tends to be extremely slow; and secondly, convergence to the global minimum is not guaranteed. Although these two seem to be closely related, existing improvements to overcome these drawbacks are summarized into two aspects below:

For the former problem (slow convergence), various acceleration techniques have been proposed and discussed below:

1. ***Dynamically modifying learning parameters.*** Many researchers have pointed out that a constant learning rate is not suitable for a complex error surface. Research into dynamic change of the LR and the momentum factor (MF) of the BP algorithm has been extensively carried out by many authors. Jacobs [29] presented a simple method for updating the LR and MF. He suggested dynamically increasing or decreasing the LR and MF by a fixed factor based on the observations of error signals. Variations of Jacob's method were reported by Vogel et al [30] and compared in detail by Allred and Kelly [31]. Hush and Salas [32] have also proposed a method for learning rate (LR) adaptation based on a heuristic technique. On the other hand, Weir [33] considered an optimum step length and established a method for self-determination of the adaptive learning rate. Most of these techniques can be considered as a kind of line search. Although this kind of method has been proven to work well for many cases, it may lead to overadjustment of the weights, resulting in dramatic divergence. As a matter of fact, the optimal LR varies almost randomly from iteration to iteration, as observed by Yu et al. [34]. This fact implies the difficulty in adjustment of the LR based on observations of previous LR and MF and error signals.

   In 1997, Yu et al [35] proposed another version of the dynamic LR optimization where they used the first two order derivative information of the objective function with respect to LR. Since this approach does not involve explicit calculation of derivatives in weight space, but rather uses the information gathered from the forward and backward

propagation, the computational and storage burden scales with the network size exactly like the conventional BP algorithm. This makes the BP learning accelerated with a remarkable reduction in the running time. However, there is still a room for speeding-up of the convergence, since dynamic optimization of the MF has not been thoroughly considered there.

2. ***Second-order method.*** Another kind of accelerating method that can be treated as a generalized version of dynamic variation of the LR is of the second-order type, including the Newton method suggested by Becker and leCun [36] and Battiti [37]; the Broyden-Fletcher-Goldgarb-Shanno method; the Levenberg-Marquardt method reported by Webb et al. [38], and others. These methods converge rapidly as compared to the conventional BP algorithm; however, the computational and storage burden is increased quadratically with the number of weights because they have to calculate and store the Hessian matrix. It would be too costly to implement these methods even for intermediate scale applications (e.g., network training with 10^3 weights). The extended Kalman learning algorithm developed by Singhal and Wu [39] and Pushkorius and Feldkamp [40], and the nonlinear recursive least-squares learning algorithm suggested by Kollias and Anastrassiou [41] also roughly belong to this category. A simplified version of this kind algorithm has recently been reported by Monhandes et al. [42], in which they used the estimate of the Hessian matrix norm to update the LR.

Although the above techniques have been successful in speeding up learning for some problems, there is not enough discussion or experiments about their abilities to avoid local minima. Moreover, they usually introduce additional parameters that are problem-sensitive [43].

The second type of drawback of the BP method (local minima) is very common among nonlinear optimization techniques; and the following techniques have been proposed to overcome this drawback:

1. ***On-line weight updating.*** Two different schemes of updating weights can be found in the literature. One approach is to accumulate partial derivatives of the error function with respect to weights over all the input-output patterns before updating weights (batch-

mode or epoch learning). In this mode, a more accurate estimate of the true gradient is achieved. The other is to update weights after every input-output pattern. This is called on-line mode learning or learning by sample. By choosing each pattern randomly, it will produce small fluctuations that sometimes deteriorate the error mapping function. Consequently, it can get out of shallow local minima as reported by Xu et al. [44].

2. ***Starting with appropriate weights.*** Kolen and Pollack [45] have shown that the BP algorithm is quite sensitive to initial weights. Weights are usually initialized with small random values. However, starting with inappropriate weights is one reason for getting stuck in local minima or slow learning progress. For example, initial weights that are too large easily cause premature saturation as reported by Lee et al. [46]. Nguyen and Widrow [47] have shown that the learning progress can be accelerated by initializing weights in such a way that all hidden units are scattered uniformly in the input pattern space. Wessels and Barnard [48] have proposed a similar technique to avoid local minima for neural net classifiers. They refer to the local minima that do not satisfy the criterion imposed for stopping a learning process as false local minima (FLM). Hence, it is always desirable to start the training process with appropriate weights that avoid both FLM and slow learning progress.

From an optimization theory point of view, the difference between the desired output and the actual output of FNN produces an error value that can be expressed as a function of the network weights. Hence, training the network becomes an optimization problem to minimize the error mapping function, which may also be considered as an objective or cost function. One possibility for improving the convergence behavior of such problem is to modify the procedure by which the objective function is optimized. Alpsan et al. [49] have suggested several modifications on this procedure but none of them was examined on matrix problems.

1.3 Present Work

A new approach to overcome all the disadvantages of the existing training methods mentioned in section 1.2.3 is presented. From a general conceptual viewpoint, the approach here views matrix algebra problems as special pattern recognition problems. Since these types of problems are amenable to the solution via feed-forward neural networks (FNN), matrix computations are represented by a linear FNN such that when the network matches a set of desired patterns, the weights of the linear neurons give the solution to the problem. In other words, the matrix problem solution is considered as a byproduct of the training process and is obtained once the training of the networks is performed successfully. Accordingly, the matrix computations in the present approach depend highly on the success and efficiency of the training algorithm for FNN.

The present approach could be summarized as follows: (1) represent a given problem by a structured network architecture (FNN); this is the "construction phase"; (2) train the structured network to match some desired patterns; this is the "training phase"; and, (3) obtain the solution to the problem from the weights of the resulting structured network; this is the "application phase". The neuron used to construct such structured network is a linear multi-input single-output weighted-summer, i.e. a neuron whose output equals the weighted sum of its inputs. The weights are adjusted by a training procedure during the "training phase".

In the present study, several training algorithms for structured networks are investigated and compared as of the avoidance of local minima and as of their simplicity when performing the matrix *LU* decomposition and matrix inversion. The goal is to train the FNN and, consequently, to get the solution of matrix problem in a reasonable time. On the basis of three performance indexes, learning time, accuracy and stability, it will be demonstrated that some of the BP algorithm modifications mentioned in the previous section may not be worth the effort for those matrix problems that do not often require finding the exact global minimum of the objective function. A thorough discussion of these modifications will be given in this study.

This dissertation is organized as follows: chapter 2 deals with the architecture and theoretical analyses of the FNN considered in this study. Also, this chapter is devoted to an explanation of the steepest descent method and an overview of the original BP algorithm. To overcome the problems of the standard BP algorithm when applied to matrix computations, all possible extensions of the first derivative-based back-propagation are investigated in chapter 3. Presentations of the feedforward neural networks used for computing matrix decompositions and inversions are included in chapter 4. The training procedures for these FNNs under the minimization methods discussed in chapter 3 are also included in chapter 4. Furthermore, the potential for parallelisation of these procedures is discussed in this chapter. The simulation results are presented and discussed in chapter 5. Finally, the conclusions of this study as well as the possible future studies and recommendations are given in chapter 6.

CHAPTER 2

Feed-Forward Neural Network and the Back-propagation Algorithm

As mentioned in chapter 1, the approach adopted in this study views matrix algebra problems as special pattern recognition problems. Usually, this type of problems appears amenable to solution with feed-forward neural networks (FNN) through which desired patterns are matched. Customarily, the design of FNN for a specific application involves many issues, most of which require problem-dependent solutions. Some of such issues are shown in figure 2.1. Of course, the essence of these neural networks lies in the connection weights between neurons. The selection of such weights is referred to as training or learning, and consequently, the weights are referred to as the learning parameters. Presently, the most popular algorithm for training FNN is called the *back-propagation* algorithm. It is based on unconstrained optimization problem and on the associated gradient algorithm applied to the problem.

The overall computational approach used in the present study for exploring the FNN and training algorithm is shown in figure 2.2. The situation may be viewed as comprising two parts: feed-forward (implementation) of the learned mapping and training of the network. The training algorithm will use the feed-forward implementation as part of training; in this sense they are coupled.

This chapter is devoted to present the theoretical analysis of feed-forward neural network that are used for matrix computations. Included also is an illustration of their major components. The existing algorithm for training these networks is also discussed and its drawbacks when applied to matrix computations are pointed out.

2.1 Structure of Feed-forward Network

The *feed-forward network* [50-51] is composed of a hierarchy of processing units (neurons), organized in a series of two or more mutually exclusive sets of neurons (or layers). The first, or input, layer serves as a holding site for the inputs applied to the network. The last, or output, layer is the point at which the overall mapping of the network input is

available. Between these two extremes lie zero or more layers of *hidden neurons,* it is in these internal layers that additional remapping or computing takes place.

Links, or weights, connect each neuron in one layer only to those in the next higher layer. There is an implied directionality in these connections, in that the output of a neuron, scaled by the value of a connecting weight, is fed forward to provide a portion of the activation for the neurons in the next higher layer. Figure 2.3 illustrates the structure of a typical feed-forward network.

In figure 2.3, note that the information flow in the network is restricted to flow layer by layer from the input to the output. For a given input, the computation of the output is achieved through the collective effect of individual input-output characteristic of each neuron. Thus, from architecture viewpoint, FNN allows parallelism (parallel processing) within each layer, but the flow of interlayer information is necessarily serial. Hence, the neural network can be regarded as a *parallel* computation device [52].

2.2 A Functional Viewpoint of FNN Architecture and Mappings

The mathematical characterization of the neuron of the present FNN is a linear multi-input single-output weighted-summer, i.e. a neuron whose output equals the weighted sum of its inputs. The weights are adjusted by the training procedure during the "training phase".

For the present study, a neural network structure consisting of three layers, as depicted in figure 2.4, is considered. As mentioned earlier, the three layers are referred to as the input, hidden, and output layers. There are n inputs x_i, where $i = 1, \ldots, n$. There are m output y_s, $s = 1, \ldots, m$. There are l neurons in the hidden layer. The outputs of the neurons in the hidden layer are z_j, where $j = 1, \ldots, l$. The role of the input layer is somewhat fictitious, in that input layer neurons are used only to hold input values and to distribute these values to neurons in the next layer. Thus, the input layer neurons do not implement a separate mapping or conversion of the input data, and their weights, strictly speaking, do not exit. In other words, the input layer is static and performs linear branching only and, thus, the inputs $x_1, \ldots, x_n$ are distributed only to the neurons in the hidden layer. The neurons in the hidden and output layers are thought of as multi-input-single-output linear elements, with each activation function being the identity map. In Figure 2.4, the neurons are not explicitly depicted in the input layer; instead, they are illustrated as signal splitters.

The weights for inputs into the hidden layer are denoted by w^h_{ji}, i =1,....., n, j = 1,...., l. The weights for inputs from the hidden layer into the output layers are denoted by w^o_{sj}, j = 1,....., l; s = 1,...., m. Given the weights w^h_{ji} and w^o_{sj}, the neural network implements a map from R^n to R^m. To find an explicit formula for this map, the input to the jth neuron in the hidden layers is denoted by v_j. In case of linear hidden neurons, the output from the jth neuron in the hidden layer denoted by z_j is equal to v_j. That is,

$$v_j = \sum_{i=1}^{n} w^h_{ji} x_i \tag{2.1}$$

$$z_j = \sum_{i=1}^{n} w^h_{ji} x_i \tag{2.2}$$

Similarly, in case of linear output neurons, the input to s^{th} neuron in the output layer is equal to the output y_s from the sth neuron of the output layer. Mathematically, this gives

$$y_s = \sum_{j=1}^{l} w^o_{sj} z_j \tag{2.3}$$

Therefore, the relationship between the inputs x_i, i=1,....., n, and the sth output y_s is given by

$$y_s = \sum_{j=1}^{l} w^o_{sj} \sum_{i=1}^{n} w^h_{ji} x_i \tag{2.4}$$

$$= F_s(x_1, \ldots\ldots\ldots, x_n) \tag{2.5}$$

Thus, the overall mapping that the neural network implements is given by the following function relating FNN input to output:

$$y_1 = F_1(x_1, \ldots\ldots\ldots, x_n) \tag{2.6}$$

$$\begin{aligned} y_2 &= F_2(x_1, \ldots\ldots\ldots, x_n) \\ . &= . \\ . &= . \\ . &= . \\ . &= . \\ . &= . \\ . &= . \\ . &= . \end{aligned}$$

$$y_m = F_m(x_1, \ldots\ldots\ldots, x_n) \tag{2.7}$$

2.3 Training Algorithm

After choosing an appropriate network structure, much of the effort in designing the FNN for a specific application will involve the design of a training algorithm. The objective of the training process is finding the set of weights that will minimize the mismatch between desired values and network's output. In other words, the FNN training involves the adjustment of the network weights such that the actual or computed output generated by the network y for a given input $x_d = [x_{d1}, \ldots\ldots, x_{dn}]^T$ is as "close" as possible to a desired or target output y_d.

Mathematically, the training process can be formulated as an optimization problem where the objective function (mapping error function, E^p) is typically chosen as the sum squared-error over the outputs of the output neurons (training-by-sample or pattern). That is,

$$\text{Minimize } E^p \tag{2.8}$$

where $E^p = \frac{1}{2}\sum_{s=1}^{m}(y_{ds}^p - y_s^p)^2$, and m is the number of the output neurons of the FNN. Here, the superscript p indicates that the mapping error function E^p is based on the pth input/output pair of the training set (patterns) H. The training set (patterns) for FNN often consists of ordered pairs of vectors and is denoted by

$$H = \{(x_d^p, y_d^p)\} \qquad p = 1, 2, \ldots, N \tag{2.9}$$

where N is the number of patterns or samples. From equation (2.4), the computed output y_s for the given input x_d can be written as

$$y_s = \sum_{j=1}^{l} w_{sj}^o \left(\sum_{i=1}^{n} w_{ji}^h x_{di} \right) \tag{2.10}$$

The network weights to be estimated via optimization may be arranged into a vector, denoted $\underline{w} = [\,\underline{w}^o, \underline{w}^h\,]^T$. Thus, equation (2.8) can be rewritten as

$$\text{Minimize } E^p\,(\underline{w}) \tag{2.11}$$

All possible weights for the network that satisfy this equation lie in a vector space having the same dimension as $\underline{w}$. Since the dimensionality of $\underline{w}$ is often large, visualization of this space is often difficult.

2.3.1 Steepest Descent Method

The problem of weight estimation in the FNN is one of the parameter estimation using nonlinear programming schemes. The values of the weight vector are chosen to minimize the objective function given by equation (2.11). Hence, beginning with an initial guess $\underline{w}_o$, an iterative method to solve this equation is to update $\underline{w}$ using,

$$\underline{w}^{(k+1)} = \underline{w}^{(k)} + \Delta\underline{w}^{(k)} \tag{2.12}$$

In case of steepest descent method, $\Delta\underline{w}^{(k)}$ is chosen to be

$$\Delta\underline{w}^{(k)} = -\eta\, g\left(\underline{w}^{(k)}\right) \tag{2.13}$$

In this expression, g is the error gradient vector and η is a small positive scalar referred to as the learning rate (LR) or the step size (usually taken as a fixed value).

Accordingly, to formulate this algorithm, it is necessary to compute the gradients of the mapping error function (2.11) in weight space or the partial derivatives of E (the objective function to be minimized) with respect to each component of w^o_{sj} and w^h_{ji}. For this, the objective function of equation (2.8) is rewritten as

$$E^p = \frac{1}{2}\sum_{s=1}^{m}\left(y^p_{ds} - \left(\sum_{j=1}^{l} w^o_{sj} z_j\right)\right)^2 \tag{2.14}$$

where, for each $j = 1,\ldots,l$,

$$z_j = \sum_{i=1}^{n} w^h_{ji} x^p_{di} \tag{2.15}$$

Using the chain rule, the error gradient for the output layer can be computed as,

$$\frac{\partial E^p}{\partial w^o_{sj}} = -\delta_s z_j \tag{2.16}$$

where δ_s is the output error $(y_{ds} - y_s)$.

Next, the partial derivative of E^p with respect to w^h_{ji} is computed. For this, we start with the equation

$$E^p = \frac{1}{2}\sum_{s=1}^{m}\left(y^p_{ds} - \left(\sum_{j=1}^{l} w^o_{sj}\left(\sum_{i=1}^{n} w^h_{ji} x^p_{di}\right)\right)\right)^2 \tag{2.17}$$

Using the chain rule once again, the error gradient for the hidden layer can be computed as

$$\frac{\partial E^p}{\partial w_{ji}^h} = -\sum_{s=1}^{m}(y_{ds}^p - y_s^p)\, w_{sj}^o x_{di}^p \tag{2.18}$$

Simplifying the above yields

$$\frac{\partial E^p}{\partial w_{ji}^h} = -\sum_{s=1}^{m}\delta_s w_{sj}^o x_{di}^p \tag{2.19}$$

Now, it is ready to formulate the gradient algorithm for updating the weights of the present FNN. The update equations for the two sets of weights w_{sj}^o and w_{ji}^h are written separately as follows:

$$w_{sj}^{o(k+1)} = w_{sj}^{o(k)} + \Delta w_{sj}^{o(k)}, \tag{2.20}$$

where the weight adjustment $\Delta w_{sj}^{o(k)}$ is set proportional to the gradient as follows:

$$\Delta w_{sj}^{o(k)} = \eta \delta_s^{(k)} z_j^{(k)}, \tag{2.21}$$

and for the hidden layer

$$w_{ji}^{h(k+1)} = w_{ji}^{h(k)} + \Delta w_{ji}^{h(k)}, \tag{2.22}$$

where the weight adjustment $\Delta w_{ji}^{h(k)}$ is set proportional to the gradient as follows:

$$\Delta w_{ji}^{h(k)} = \eta\, f_j^{(k)} x_{di}^p, \tag{2.23}$$

where

$$f_j^{(k)} = \sum_{s=1}^{m}\delta_s^{(k)} w_{sj}^{o(k)}, \tag{2.24}$$

$$z_j^{(k)} = \sum_{i=1}^{n} w_{ji}^{h(k)} x_{di}^p, \tag{2.25}$$

$$y_s^{p(k)} = \sum_{j=1}^{l} w_{sj}^{o(k)} z_j^{(k)}, \tag{2.26}$$

$$\delta_s^{(k)} = (y_{ds}^p - y_s^{p(k)}). \tag{2.27}$$

The update equation for the weights w_{sj}^o of the output layer neurons is illustrated in figure (2.5), whereas the update equation for the weights w_{ji}^h of the hidden layer neurons is illustrated in figure (2.6).

2.3.2 Pattern Presentation and Weight-Updating Strategies

Given the power of the weight correction strategy developed in section 2.3.1, numerous options are possible. Part of the rationale for these strategies is explored below. Most important, training by sample versus training by epoch is considered.

2.3.2.1 Training by Sample

Equations (2.21) and (2.23) give *product forms* for individual weight correction, or update, based on the difference between y_s^p, y_{ds}^p, for a pre-specified x_d^p. Equation (2.21) can be rewritten as:

$$\Delta^p w_{sj}^{o(k)} = \eta \delta_s^{(k)} \sum_{i=1}^{n} w_{ji}^{h(k)} x_{di}^p, \tag{2.28}$$

where η could be viewed as an adjustment or scaling parameter and the error of the sth output neuron is defined by Equation (2.27). The superscript p in $\Delta^p w_{sj}^{o(k)}$ indicates that the weight correction is based on the pth input/output pair of H. The network weights are corrected for the pth training pair using Equations [(2.20) and (2.22)] for all i, j and s. This is called training *by sample,* or *pattern-based training.* A closer look to this weight correction technique (Equation (2.28)) suggests the following two potential deficiencies:

1. If $x_d^p = 0$, there is no correction, even for nonzero δ_s^k.
2. The effect of this weight correction on the network response to other pattern pairs in the training process has not been considered.

The above deficiencies may lead to three serious phenomena: "premature saturation", "occurrence of saddle points", and "entrapment in a local error minimum". To overcome these phenomena, the pattern mode of training is usually coupled with a random pattern selection strategy. This allows a weight correction that is somewhat random in nature. A "good" weight selection (initialization) procedure would be to initialize neurons (with random weights) such that the expected value of neuron activation is zero [48]. This, in turn, helps the neurons to be in their "active regions" to obtain maximum weight correction, i.e., where the derivative of the activation function is not too small or not equals zero. Of course, this makes back-propagation training more efficient. Fortunately, there is no need to use this

random pattern selection strategy in the present study where the neurons are linear since the derivative of the activation function in this case is always equal to one.

2.3.2.2 Training by Epoch (Batch)

Weight corrections based on individual input patterns usually guide the gradient descent procedure. If it is desired to use the total epoch error, E^N, to guide the gradient descent procedure, the gradient $\frac{\partial E^N}{\partial \underline{w}}$ is required [53]. Fortunately, this does not require a separate derivation since the total epoch error can be formed by summing the mapping error function E^p, given by equations (2.14), over all the pattern pairs in the training set (epoch) as follows:

$$E^N = \sum_{p=1}^{N} E^p, \tag{2.29}$$

Hence, the gradient is given by

$$\frac{\partial E^N}{\partial \underline{w}} = \sum_{p=1}^{N} \frac{\partial E^p}{\partial \underline{w}} , \tag{2.30}$$

Thus, an alternative is training by epoch, where the following form of weight correction based on equation (2.28) is formed

$$\Delta^N w_{sj}^{o(k)} = \sum_{p=1}^{N} \Delta^p w_{sj}^{o(k)}, \tag{2.31}$$

This represents an overall or accumulated correction to the weight set after each sweep of all pattern pairs in the training set, or training epoch. This approach is also referred to as *batch training*. Epoch-based training represents a smoothing of the weight corrections. Therefore, this approach will be adopted in the present study.

2.3.2.3 Mixed Strategies

Intermediate or hybrid training methods are also possible. In one such approach, suggested by Munro [54], a pattern is presented repeatedly until its error is reduced to a certain preselected value. A generalization of this [55] is a set of *educational* pattern presentation strategies. These strategies have the common property that the frequency of presentation of an individual pattern is related to its mapping error. Patterns that have high

errors, or are "difficult to train," may be presented more frequently. Of course, care must be taken not to undo prior training.

2.3.3 The Search for $\underline{w}$

The overall mapping implemented by the FNN (Eq. 2.7) may also be characterized as

$$y = f(x, \underline{w}, H) \tag{2.32}$$

Assuming *a fixed topology, a fixed training set,* and *weight-independent neuron characteristics,* Equation (2.32) may be viewed in several ways. One of such ways is that, for fixed x and y, it may be viewed as a constraint on the allowable values of $\underline{w}$. Specifically, there may be no solutions, a unique solution, or many solutions for $\underline{w}$, depending upon the given values of x and y.

Assuming there is at least one solution to Equation (2.32), this *solution space* may be visualized as the subspace of weight space in which Equation (2.32) is satisfied. As shown below, this space has several properties. Most importantly, however, *this is the space where solutions should be searched.* The network training or learning algorithm guides this search. It is noted that

- Some algorithms may only explore a limited region of weight space, denoted the *algorithm search space,* thereby missing the solution if the solution and algorithm search spaces are non-overlapping.
- Some algorithms may be more efficient at this search than others.
- The computational complexity of this search is worth investigating.

The formulation of Equation (2.32) is based on H, which is given by equation (2.9). Requiring Equation (2.32) to be satisfied for *all* elements of H is thus a more critical constraint. Specifically, it is the conjunction of each of the N constraints given by Equation (2.32). Equations (2.9) and (2.32) may then be combined to define an epoch mapping error, $E^N(\underline{w}, H)$, as follows:

$$E^N(\underline{w}, H) = \frac{1}{2}\sum_{p=1}^{N}\sum_{s=1}^{m}(y_{ds}^p - y_s^p)^2 \tag{2.33}$$

where the training objective is to minimize E^N. Notice that Equation (2.33) does not require the gradient descent, or any particular procedure. It merely stipulates that there exists a

problem-specific function of $\underline{w}$ and H that is to be minimized. Most often this is done by some guided search over weight space. For example, the search could be guided by gradient descent, where $\underline{w}$ is iteratively updated as follows:

$$\underline{w}^{(k+1)} = \underline{w}^{(k)} + \eta \left(- \frac{\partial E^N}{\partial \underline{w}} \right) \tag{2.34}$$

As a search procedure for $\underline{w}$ Equation (2.34), however, may be "fooled" into a local minimum and may be inefficient. A limiting argument applied to Equation (2.34) shows that the temporal derivative of $\underline{w}$ during training, (i.e., $\partial \underline{w} / \partial t$), and the gradient, $\partial E^N / \partial \underline{w}$, are related. Using Equation (2.34) yields

$$\frac{\partial \underline{w}}{\partial t} = fn(\underline{w}, H) \tag{2.35}$$

This shows that learning is characterized by a (usually) nonlinear differential equation.

2.3.4 Error Surface

Equation (2.33) is even more important in that it *defines a training set- (and architecture-) specific surface in weight space.* For a fixed architecture this surface will change with changes in H. Although it is impossible to visualize this surface when the dimension of $\underline{w}$ exceeds 3, it nonetheless is one of the most important tools in characterizing training. The search for a reasonable or optimal $\underline{w}$ is highly dependent on the shape of E^N; for some H the surface shape (coupled with a choice of training or search algorithm) may yield miserable results. Another H, with the same parameters, may yield impressive learning algorithm performance [28].

2.4 An Overview of the Back-propagation (BP) Algorithm

The back-propagation algorithm [28] is a product-learning rule for a feed-forward, multiple-layer, neural network that uses steepest descent method with *fixed* step size to achieve training or learning by error correction. This algorithm consists of the following steps:

1) Initialize all neuron weights in the network.
2) Apply an input (stimulus) vector to the network.
3) Feed forward or propagate the input vector to determine neuron outputs.
4) Compare neuron responses in the output layer with the desired or target response.
5) Compute and propagate an error sensitivity measure backward (starting at the output layer) through the network, using this measure as the basis for the weight correction.
6) Minimize the overall error at each stage through neuron weight adjustments.

Based on the steepest descent method mentioned in section (2.3.1), this algorithm can be summarized qualitatively as follows: Using the inputs x_{di} and the current set of weights, we first compute the quantities z_j^k, y_s^k, and δ_s^k, in turn. This called the *forward* pass of the algorithm, because it involves propagating the input forward from the input layer to the output layer. Next, we compute the update weights using the quantities computed in the forward pass. This is called the *reverse* pass of the algorithm, because it involves propagating the computed output errors δ_s^k backward through the network. This is shown in figure 2.7.

2.5 Comments on the BP Training Algorithm

Several problems may be encountered when using the BP algorithm in matrix computations and the probable reasons of these problems are explored below:

2.5.1 Error Trajectories During Training

The BP procedure described before is based on first-order gradient descent and therefore *may find a local minimum in E.* Unfortunately, as shown in Figure 2.8, local minima found by the solution procedure may correspond to sub-optimal solutions with respect to the global minima. In addition, the particular minimum found, and corresponding network weights, is a function of many parameters, including $\underline{w}_0$. This is shown in figure 2.9.

A valuable tool for monitoring training is to plot E as a function of the iteration. This may take many forms, including quick, direct convergence, oscillatory behavior (both stable and unstable), and "plateauing," as shown in Figure 2.10.

2.5.2 Back-propagation Starting Points

As mentioned above, the error surfaces of most mapping problems are subject to local minima, which result in inferior mappings. In the next chapter several approaches that attempt to avoid local minima are introduced. However, many algorithms that rely on a sequential search over the error surface may become caught in local minima. As will be seen in Section 3.4, adding a momentum term to the back-propagation equations may help to overcome this problem. Another important concept that attempts to overcome local minima and make back-propagation-based training more efficient is proper initialization of weights.

2.5.3 Effect of the Error Surface on the BP Algorithm

In addition to the problem of local minima, which may be encountered in the BP algorithm, at least two problems can occur when $\partial E^N / \partial \underline{w} = 0$ is used in BP algorithm to guide the search for $\underline{w}$. These problems are premature saturation and saddle points that are mentioned in section 2.3.2.1.

2.5.4 Learning Rate (LR)

Usually, the learning rate determines what amount of the calculated error gradients will be used for the weight correction [56]. The "best" value of learning rate depends on the characteristics of the error surface, i.e., a plot of E versus $\underline{w}$. If the surface changes rapidly, the gradient calculated only on local information will give a poor indication of the true "right path." In this case, a smaller rate is desirable. On the other hand, if the surface is relatively smooth, a larger learning rate will speed convergence. This rationale, however, is based on knowledge of the shape of the error surface, which is rarely available. Some indication may be given by calculation of E at each iteration and by observation of the impact of previous weight corrections. A general rule might be to use the largest learning rate that works and does not cause oscillation. A rate that is too large may cause the system to oscillate and thereby slow or prevent the network's convergence.

To overcome the above deficiencies, the use of line search method (bracketing with golden section method [57]) is investigated in the next chapter to obtain the optimum learning rate η as training progresses. This will not only increase the rate of convergence but it will also have a great influence on the stability of the method.

To increase the rate of convergence in the above approach, and consequently, to speed up the training of the network, the use of the conjugate gradient (CG) techniques [36] as well as the use of momentum terms [58] are investigated in the next chapter.

2.6 Design of Training Strategy

The design of a reasonable training strategy necessitates some engineering judgment as to the following training concerns:

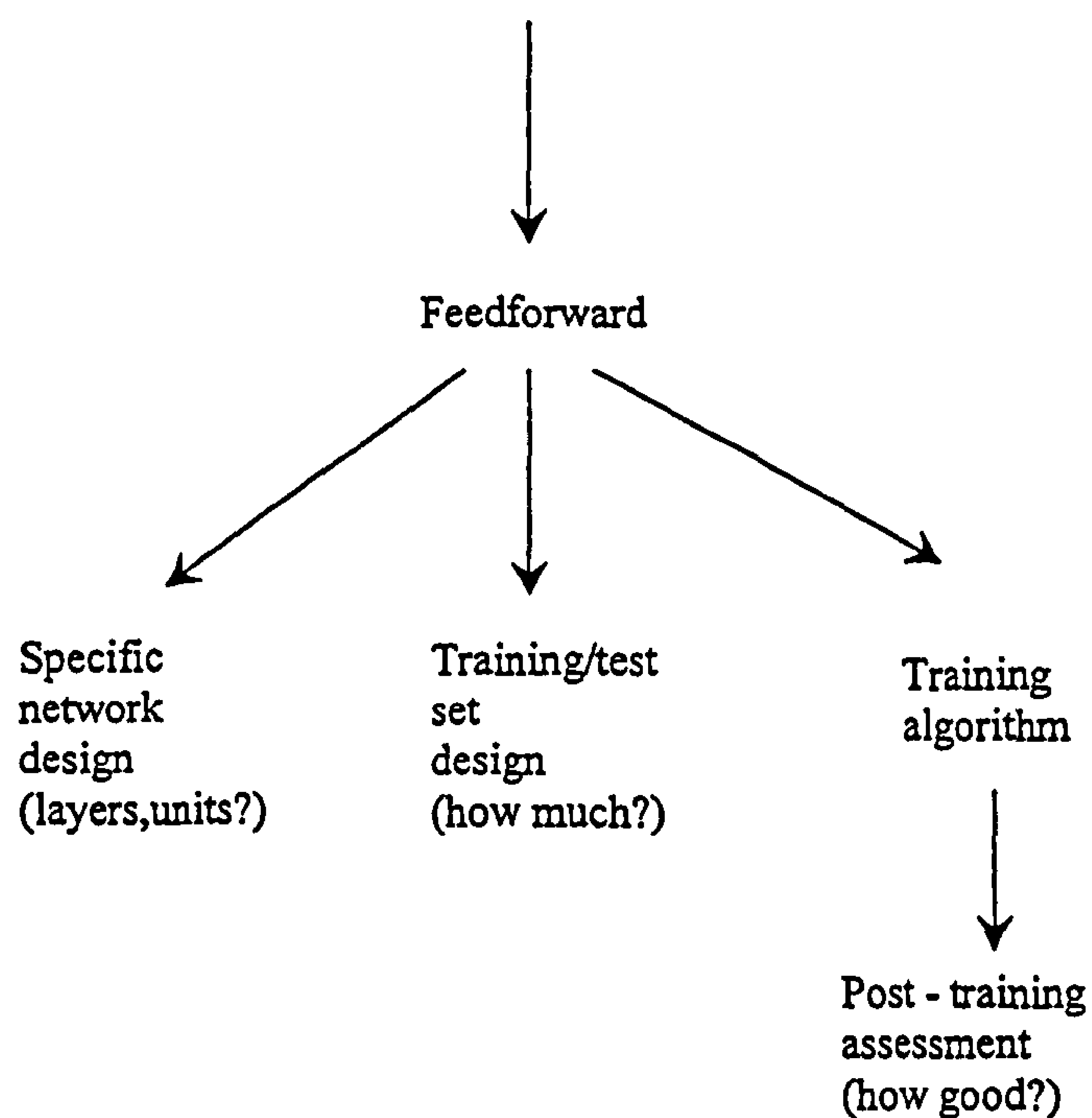
1) Whether to train by sample or by an epoch (a single pass through the training set).
2) Sequential versus random ordering of the training set data.
3) Determining whether the training algorithm has diverged, has converged, is oscillating, or is stuck at a local error minimum.
4) Determining appropriate initial conditions on weights, etc.
5) Whether to use first-order algorithm or higher-order algorithm.
6) Whether to use momentum or not.
7) When using momentum, what would be the appropriate value for momentum to use?

Accordingly, FNN training algorithm is judged by a set of often-conflicting requirements, namely; simplicity, flexibility, and efficiency. Simplicity is a measure of the effort required to apply the algorithm, including computational complexity. Flexibility relates the extendibility of the algorithm to training different architectures, and efficiency relates the computational requirements for training and the success (i.e., resulting network performance) of the training phase.

Since the matrix problem solution is considered as a byproduct of the training process and is obtained once the training of the networks is performed successfully, a "good" design of the training algorithm pertaining to simplicity, flexibility, and efficiency is very crucial to

the matrix computation via FNN. Hence, in this study an effort is devoted to reach a "good" design for training strategy.

- I/O representation
- Architecture of network (for desired application), e.g. :



- Training of network
- Validation of network

Figure 2.1 Some problem – dependent design issues.

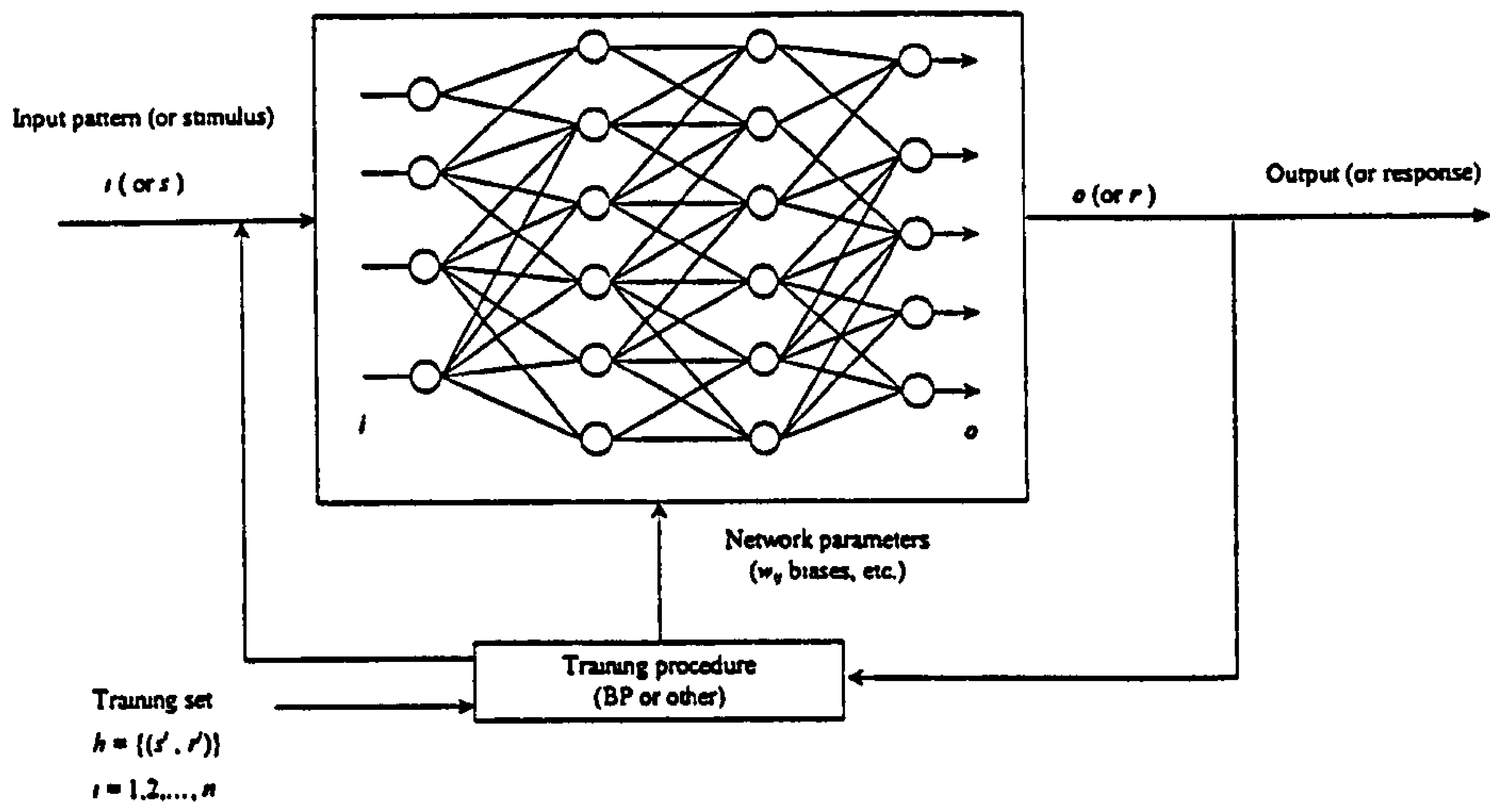


Figure 2.2 Overall FNN–based strategy (implementation and training).

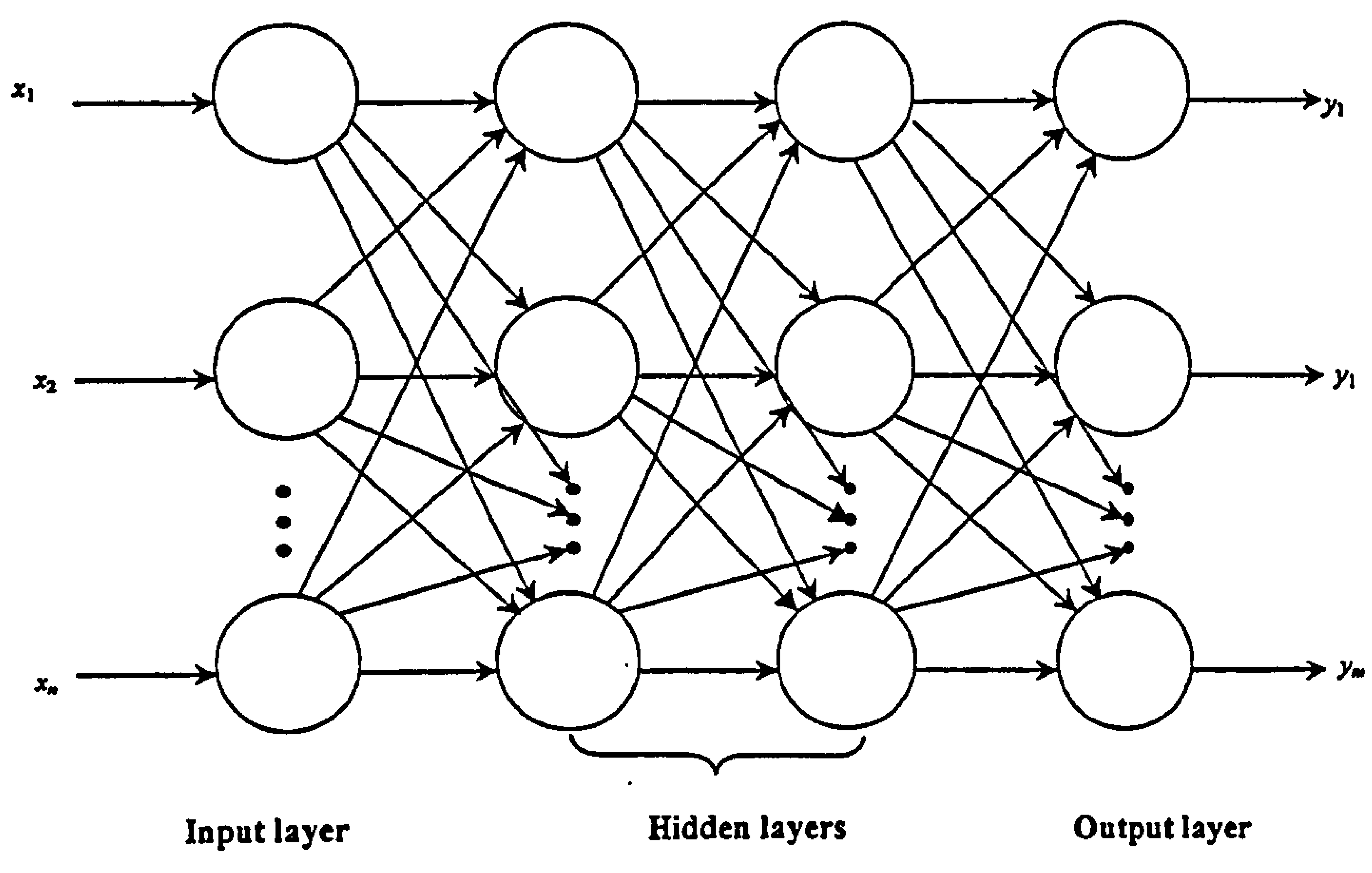


Figure 2.3 Structure of a feed forward neural network.

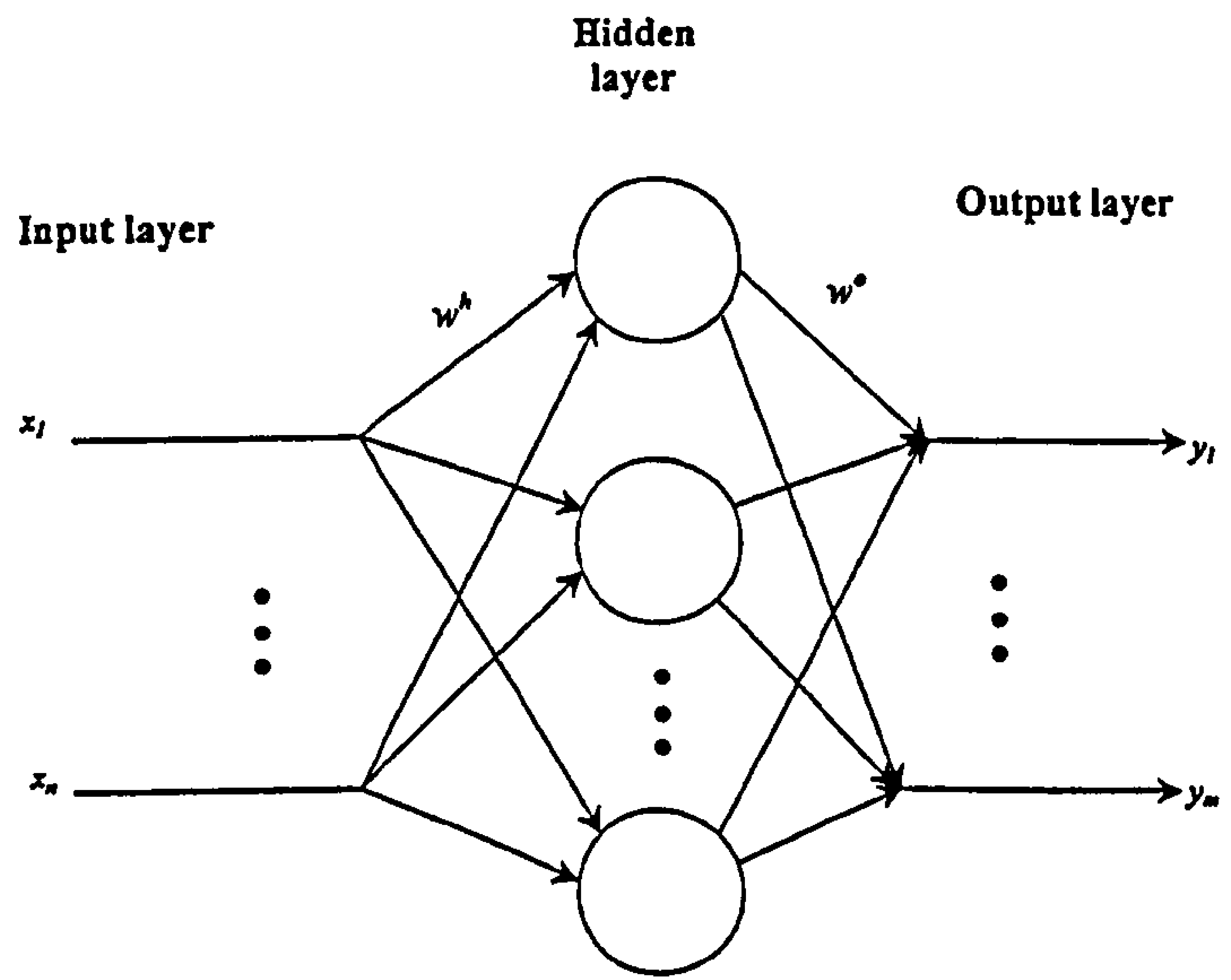


Figure 2.4. A three layered neural network.

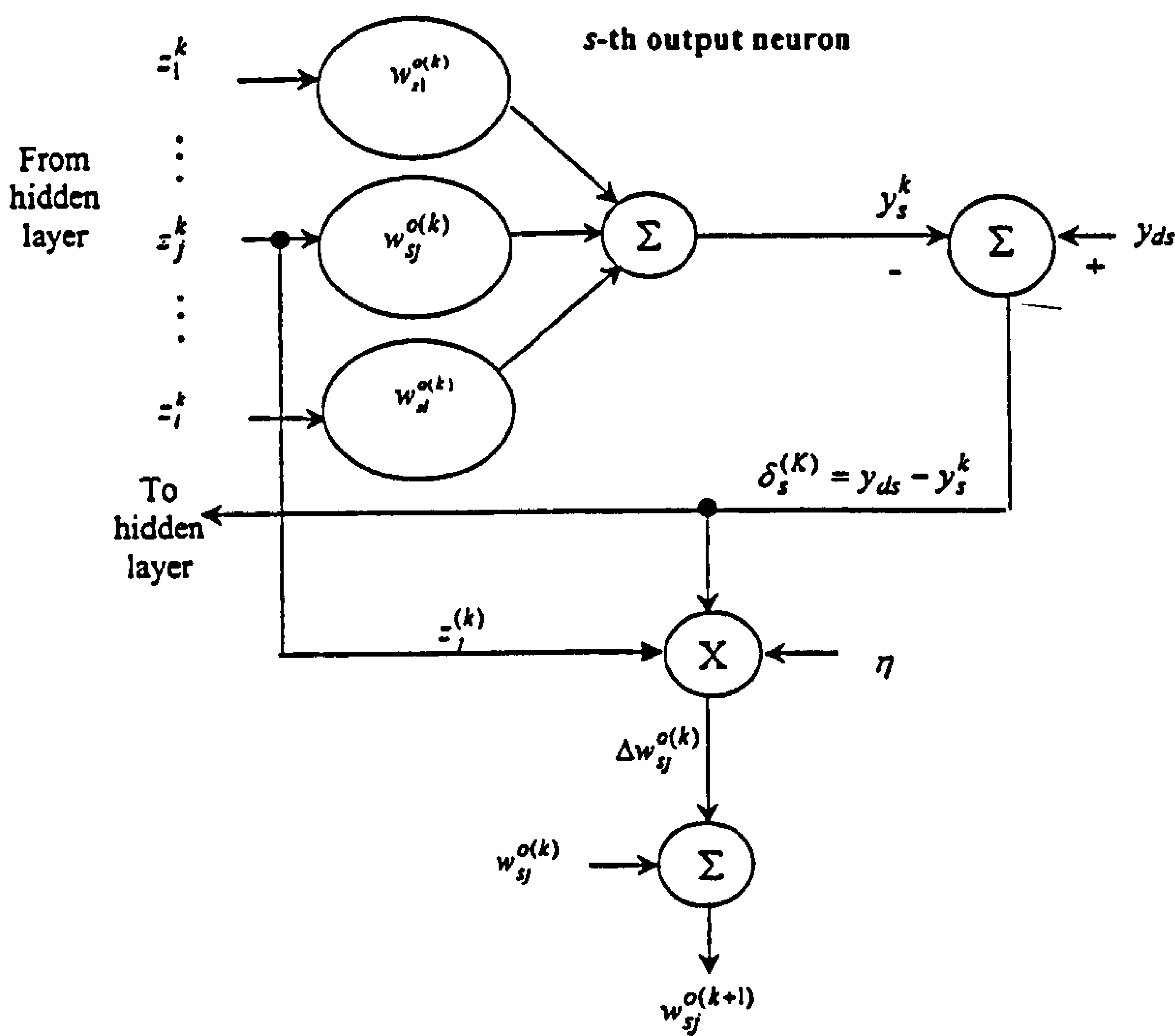


Figure 2.5 Illustration of the update equation for the output layer weights.

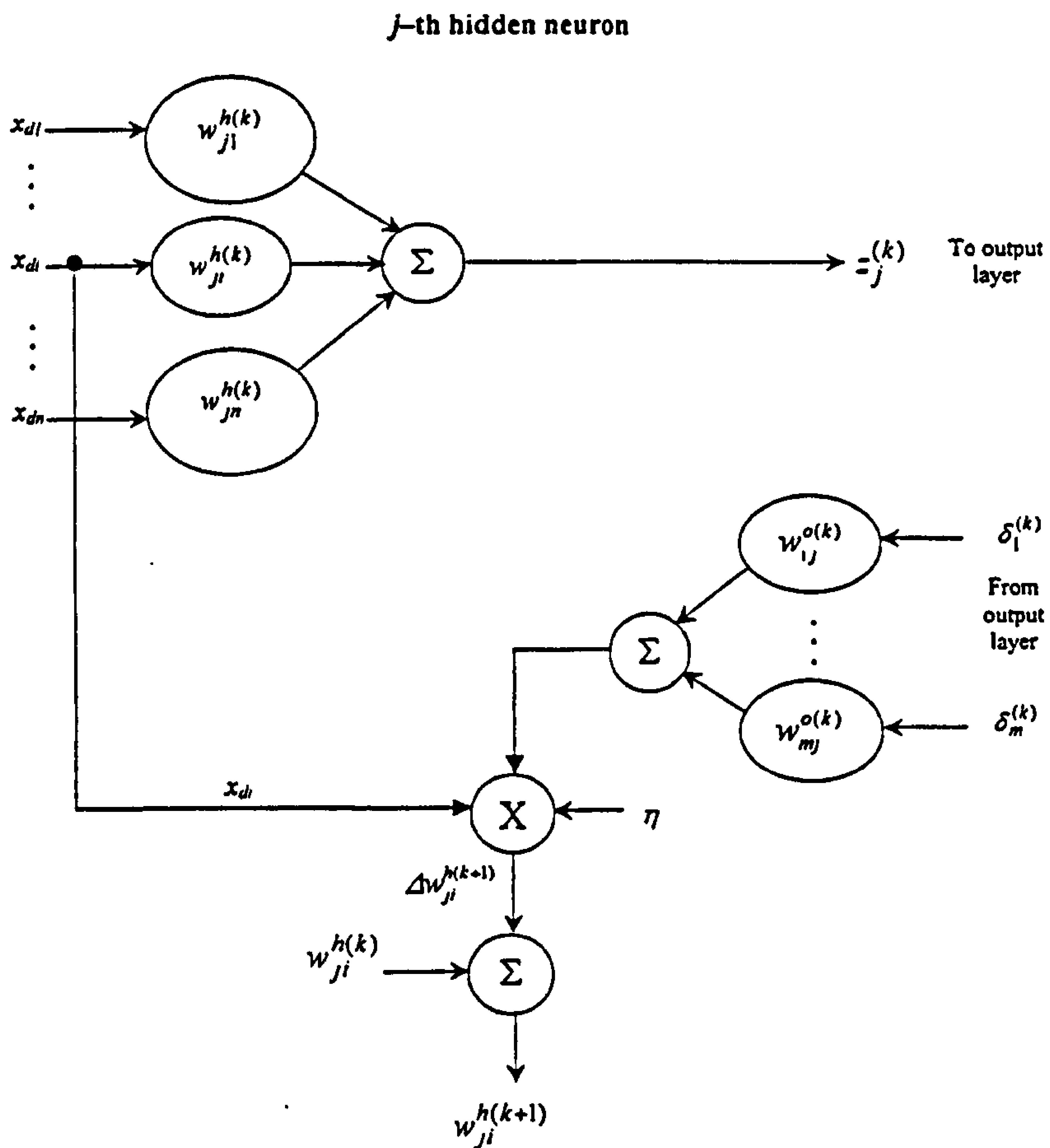


Figure 2.6 Illustration of the update equation for the hidden layer weights.

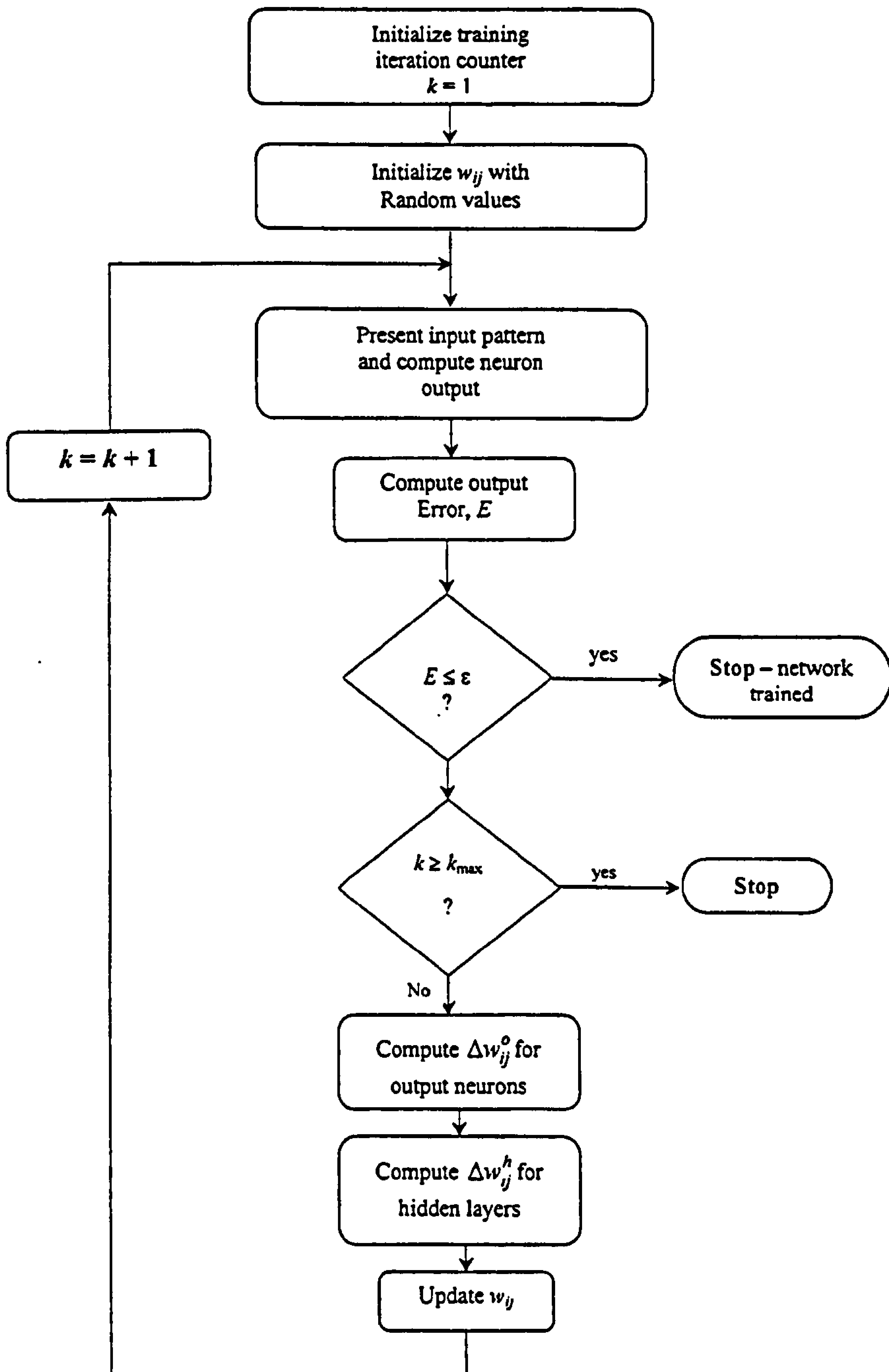


Figure 2.7 The overall procedure for Back propagation (BP) algorithm.

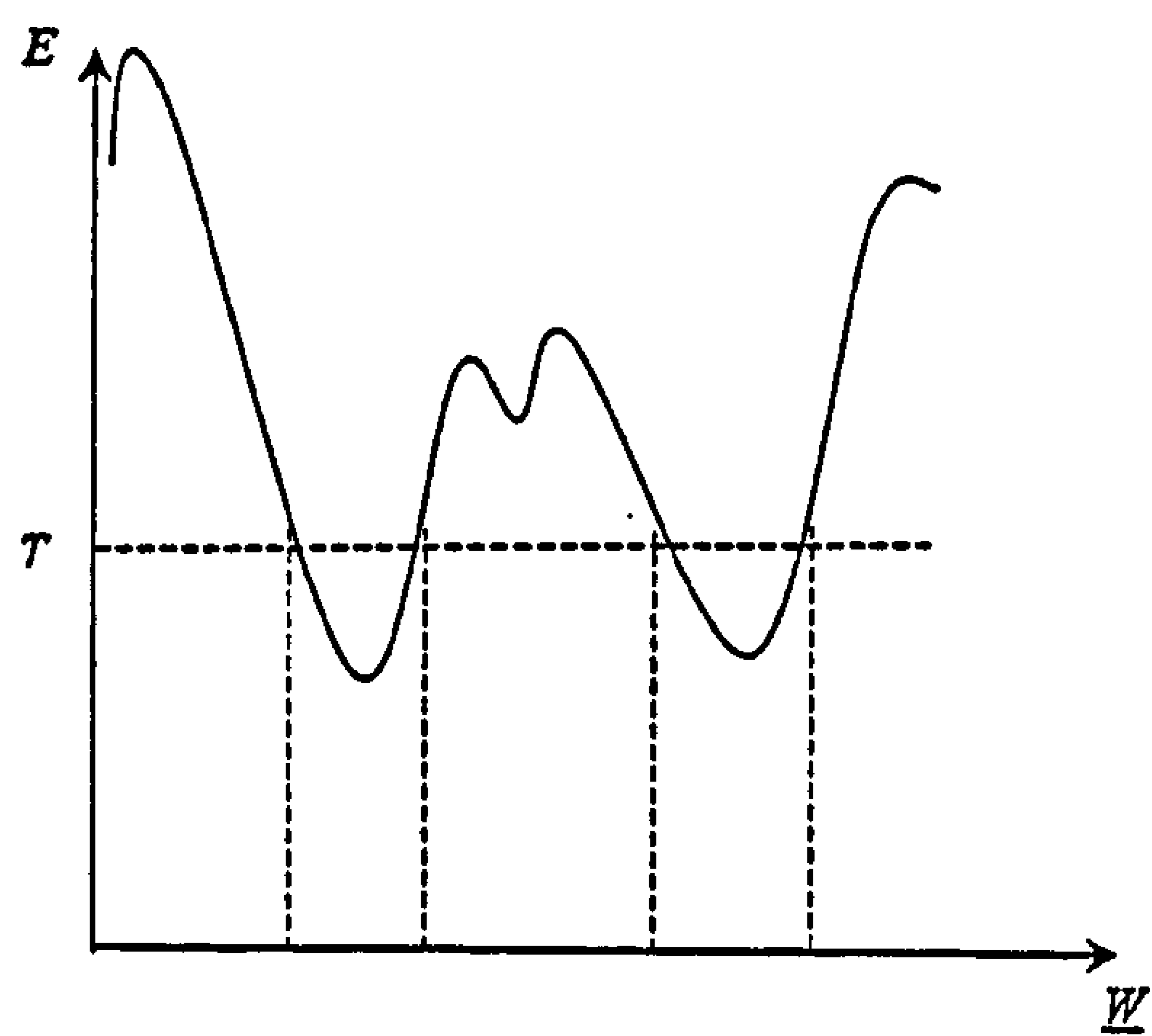


Figure 2.8 Possible minima of E (w) found in training.

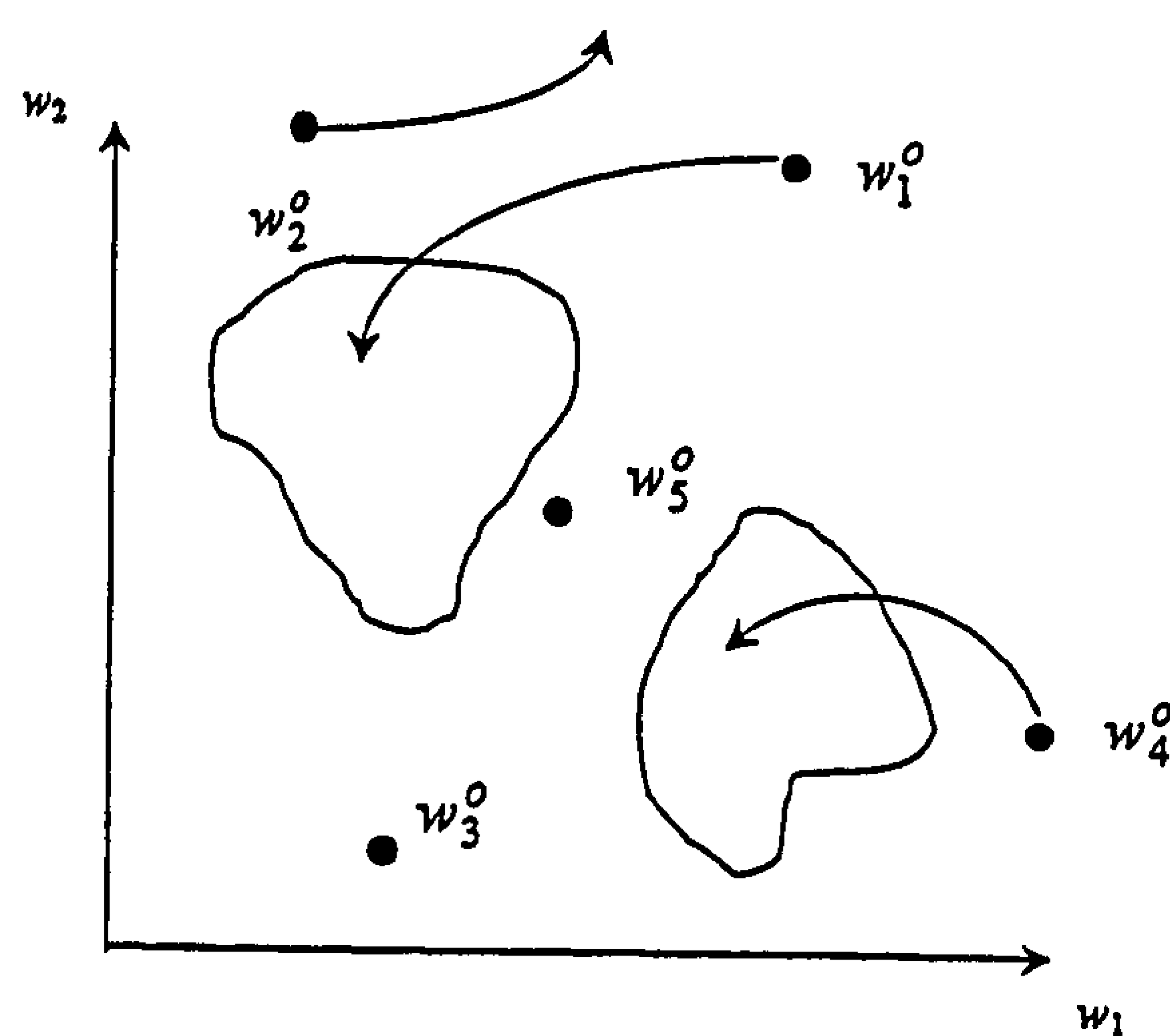


Figure 2.9 Possible weight trajectories during training.

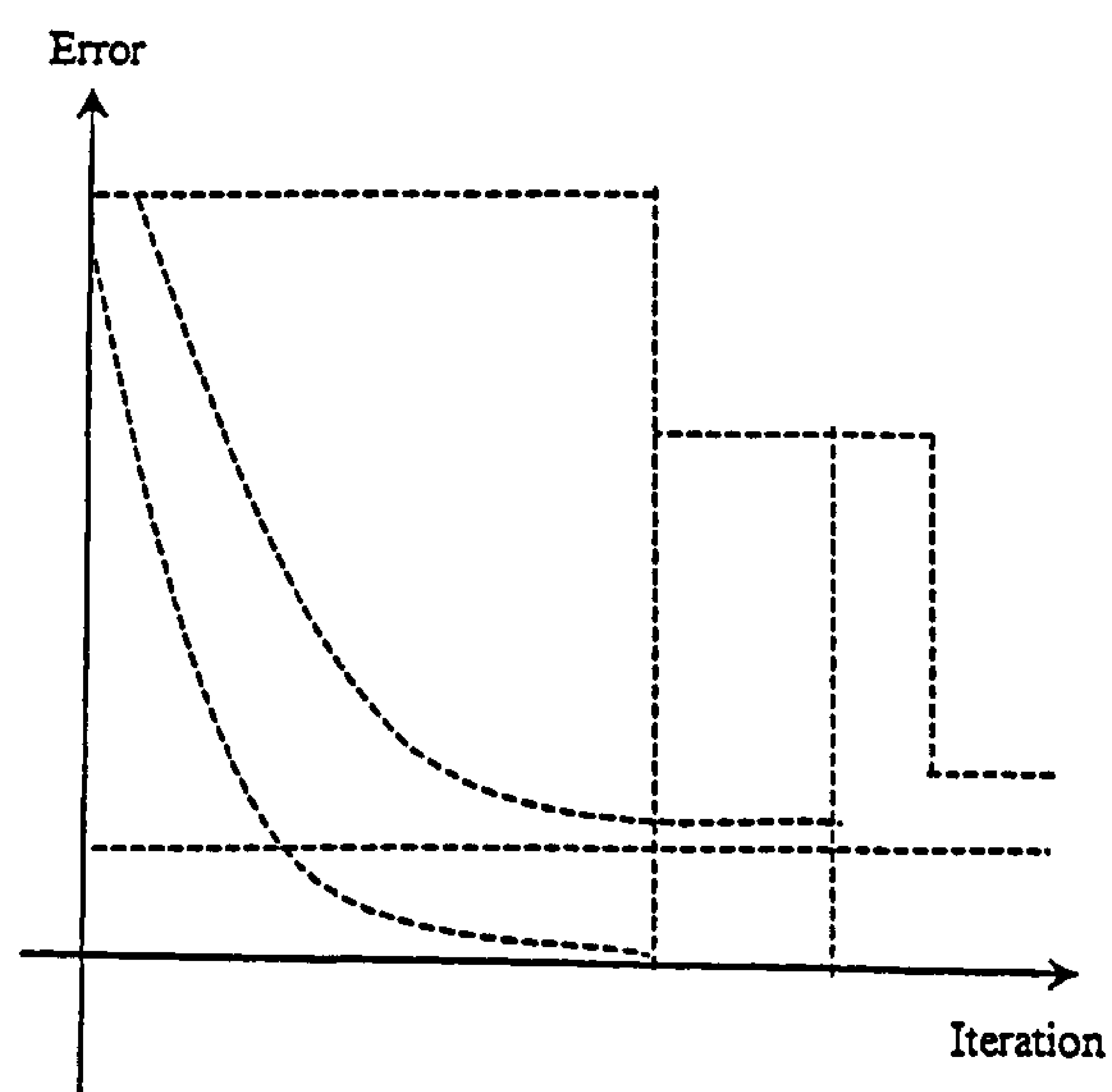


Figure 2.10 Possible error evolution during training.

CHAPTER 3

Minimization Methods for Training FNN

The back-propagation training algorithm, as shown in Chapter 2, employs the back propagation of error gradients and gradient descent. Numerous improvements and extensions have been proposed, the most popular of which is probably momentum. A large number of other, mostly heuristic, modifications also exist [48]. For example, adaptive learning rates have been investigated by a number of researchers [29].

Conventional numerical analysis and optimization theory, on the other hand, provide a rich source of alternative and more sophisticated approaches. Especially noteworthy is the use of second-derivative information. For example, when the objective function (total mapping error function, E^N) is represented by a second-order Taylor series, the global minimum can be found in V iterations, where V is the number of degrees of freedom of the system [59].

The problem of minimizing E^N is one of *unconstrained optimization;* that is, only an objective function is specified and no other constraints are specified. Hence, all variants of optimization can be understood as ways of solving the problems of the BP algorithm; such as, the temporal instability, the poor convergence, and scaling properties.

In this chapter, in an effort to overcome the above problems of standard BP algorithm when applied to matrix computations, all possible extensions of the first derivative-based back-propagation are investigated below.

3.1 Taylor Series Expansion for the Epoch Mapping Error

From Section 2.3.3, the epoch mapping error $E^N(\underline{w})$ for weight values $\underline{w}$ is defined as the total sum of square errors and has the form

$$E^N(\underline{w}, H) = \frac{1}{2}\sum_{p=1}^{N}\sum_{s=1}^{m}(y_{ds}^p - y_s^p)^2 \tag{3.1}$$

Equation (3.1) defines hyper-surfaces in $\underline{w}$ space, which are rarely expressible in closed form, and cannot be visualized (except in low-dimension spaces). E^N may be expanded in a Taylor series about $\underline{w}_o$, yielding:

$$E^N(\underline{w}) = E^N(\underline{w}_o) + \left.\frac{\partial E^N}{\partial \underline{w}}\right|_{w=w_o} (\underline{w} - \underline{w}_o)$$

$$+\frac{1}{2!}(\underline{w} - \underline{w}_o)^T \left.\frac{\partial^2 E^N}{\partial \underline{w}^2}\right|_{w=w_o} (\underline{w} - \underline{w}_o) + \text{higher order terms} \tag{3.2}$$

It is worth noting that the third term in Equation (3.2) contains Hessian matrix, $\frac{\partial^2 E^N}{\partial \underline{w}^2}$, which is often computationally very significant. This is because $\frac{\partial^2 E^N}{\partial \underline{w}^2}$ is a matrix of dimension n_w x n_w, where n_w is the total number of weights in the network (i.e., the dimension of $\underline{w}$).

3.2 First-Order Methods

As shown in figure 3.1, even when a local minimum in E^N is found by the search algorithm, the trajectory in weight space is not unique, or perhaps even direct or efficient. Figure 3.1 shows two somewhat extreme cases. In one, convergence to the local minimum is achieved in two iterations, whereas the other requires a series of somewhat oscillatory weight changes. In most cases, the former is preferred.

In first-order methods only the first two terms on the right-hand side of Equation (3.2) are used. The second term; namely, $\frac{\partial E^N}{\partial \underline{w}}$ is used as the basis for the weight correction:

$$\Delta^N \underline{w} = -\eta \frac{\partial E^N}{\partial \underline{w}} \tag{3.3}$$

This equation was derived in Chapter 2. Here, we seek a point in weight space, denoted $\underline{w}$, for which $\frac{\partial E^N}{\partial \underline{w}} = 0$. This also requires the second derivative matrix, $\frac{\partial^2 E^N}{\partial \underline{w}^2}$ to be positive definite; that is,

$$(\underline{w} - \underline{w}_o)^T \left.\frac{\partial^2 E^N}{\partial \underline{w}^2}\right|_{\underline{w}=\underline{w}_o} (\underline{w} - \underline{w}_o) > 0 \qquad \text{for } \|\underline{w} - \underline{w}_o\| \neq 0 \tag{3.4}$$

3.2.1 Limitations of First Order (Steepest Descent) Methods

In addition to the fundamental problem of finding local minima in E^N, several other limitations in first derivative-based (first-order) methods exist if constant step size is used:

- Step sizes (denoted by η) must be carefully chosen.
- The direction of minimization at each step is orthogonal to the contour of constant E^N (i.e., in a direction opposite to the gradient at that point), and therefore the trajectory of $\underline{w}$ may zigzag toward a minimum. One of the most significant aspects of this, however, is that *subsequent weight corrections are not orthogonal to those previously applied* and thus may undo some previous training. Figure 3.*2a* illustrates this concept.

For example, in Figure *3.2a* the weight vector $\underline{w}_1$, computed after $\underline{w}_o$ is used, has the undesirable property that some of the correction from $\underline{w}_o$ is undone. Thus, a desirable feature of any multi-step algorithm is that new search directions should not interfere with the minima found in previous search directions. This is one of the aims of higher-order methods, especially conjugate gradient methods (figure 3.2*b*) as will be seen in Section 3.5.3.

3.3 Steepest Descent with a Line Search Method

One extension to gradient descent is to constrain the direction of the search to be along a line determined by the gradient, with length determined by adjusting the learning rate (LR) $\eta^{(k)}$ at each step:

$$\underline{w}^{(k+1)} = \underline{w}^{(k)} - \eta^{(k)} \left. \frac{\partial E^N}{\partial \underline{w}} \right|_{\underline{w}=\underline{w}^{(k)}} \qquad \eta^{(k)} > 0 \tag{3.5}$$

The rate $\eta^{(k)}$ is chosen such that *the weight correction moves along a line (in the negative gradient direction) until a minimum is reached.* Extra computation is required to determine the "length" of this line. A more general form of Equation (3.5) leads to line minimization techniques, which are related to the conjugate gradient strategies that will be discussed later in this chapter. The basic formulation is

$$\underline{w}^{(k+1)} = \underline{w}^{(k)} + \eta^{(k)} \underline{d}^{(k)} \tag{3.6}$$

where $\underline{d}^{(k)}$ is the direction of the search in weight space at iteration k. The strategy is to choose both $\underline{d}^{(k)}$ and $\eta^{(k)}$ to minimize $E^N(\underline{w}^{(k+1)})$. Suppose $\underline{d}^{(k)}$ has already been chosen using gradient descent, i.e.,

$$\underline{d}^{(k)} = -\frac{\partial E^N\left(\underline{w}^{(k)}\right)}{\partial \underline{w}} \tag{3.7}$$

then $\eta^{(k)}$ must be found such that

$$\frac{\partial E^N\left(\underline{w}^{(k+1)}\right)}{\partial \eta} = 0 \tag{3.8}$$

Notice that no specific technique for the determination of $\eta^{(k)}$ has been chosen yet to achieve the objective of Equation (3.8). The conceptually simplest strategy is to try $\eta^{(k)}$ for a sequence of values; compute $E^N\left(\underline{w}^{(k+1)}\right)$ along $\underline{d}^{(k)}$ for each $\eta^{(k)}$; and stop when $\eta^{(k)}$ becomes large enough to cause $E^N\left(\underline{w}^{(k+1)}\right)$ to increase; in this case $\eta^{(k)}$ will be the optimal learning rate (LR) at this direction. The most important thing to note about this strategy is that successive weight correction steps are orthogonal to the gradient; therefore this strategy has the quadratic convergence properties of the conjugate gradient approach. To verify this property, observe that

$$\begin{aligned}
\frac{\partial E^N\left(\underline{w}^{(k+1)}\right)}{\partial \eta} &= \frac{\partial E^N\left(\underline{w}^{(k)} + \eta^{(k)}\underline{d}^{(k)}\right)}{\partial \eta} \\
&= \left[\frac{\partial E^N\left(\underline{w}^{(k+1)}\right)}{\partial \underline{w}^{(k+1)}}\right]^T \left[\frac{\partial\left(\underline{w}^{(k)} + \eta^{k)}\underline{d}^{(k)}\right)}{\partial \eta}\right] \\
&= \left[\frac{\partial E^N\left(\underline{w}^{(k+1)}\right)}{\partial \underline{w}^{(k+1)}}\right]^T \underline{d}^{(k)} \\
&= \nabla_{\underline{w}^{(k+1)}} E^N\left(\underline{w}^{(k+1)}\right) \bullet \nabla_{\underline{w}^{(k)}} E^N\left(\underline{w}^{(k)}\right) \\
&= 0
\end{aligned} \tag{3. 9}$$

In fact, a common way to develop the conjugate gradient methods, as will be seen later, is to begin with a choice of a direction $\underline{d}$ and then require that the length of the weight correction be such that the minimum of E^N in this direction is obtained [60].

3.3.1 A Hybrid Line Search Method

As pointed out, the speed of the first order and conjugate gradient methods depends to a large extent on the efficiency of the line-search. The search methods discussed in this section allow us to determine the minimizer of a function E^N over a closed interval, say $[a_o, b_o]$. The only property assumed for the objective function E^N is that it is *unimodal*, which means that E^N has only one local minimizer. An example of such a function is depicted in figure 3.3. The methods discussed are based on evaluating the objective function at different points in the interval $[a_o, b_o]$. These points are chosen in such a way that an approximation to the minimizer of E^N may be achieved in as few evaluations as possible. Hence, the goal is to progressively narrow the range until the minimizer is boxed in with sufficient accuracy.

In this study, to control the degree of inexactness of the line search, a hybrid line-search method is examined. This method combines two line search algorithms; namely, a bracketing algorithm (using function evaluations only) with golden section search algorithm [57]. As the name suggests, this method before using the golden section search, it uses the bracketing algorithm described below to bracket the minimum of the function to be minimized between two points, through a series of function evaluations. Then, these two end points of the bracket are used as the initial points for the golden section search algorithm described below to locate the minimum to a desired degree of accuracy.

3.3.1.1 Golden Section Search Algorithm

Consider a unimodal function E of one variable and the interval $[a_o, b_o]$. If E is evaluated at only one intermediate point of the interval, the range within which the minimizer is located cannot then be narrowed. The function E has to be evaluated at two intermediate points, as illustrated in Figure 3.4. The intermediate points will be chosen in such a way that the reduction in the range is symmetric, in the sense that

$$a_1 - a_0 = b_0 - b_1 = \rho(b_0 - a_0),$$

where

$$\rho < \frac{1}{2}$$

The function E is then evaluated at the intermediate points. If $E(a_1) < E(b_1)$, then the minimizer must lie in the range $[a_0, b_1]$. If, on the other hand, $E(a_1) \geq E(b_1)$, then the minimizer is located in the range $[a_1, b_0]$ (see figure 3.5).

Starting with the reduced range of uncertainty the process can be repeated and similarly two new points, say a_2 and b_2, using the same value of $\rho < ½$ as before can be found. However, it is desired to minimize the number of the objective function evaluations while reducing the width of the uncertainty range. Suppose, for example, that $E(a_1) < E(b_1)$, as in figure 3.5, then, it is known that $\underline{w}^* \varepsilon [a_0, b_1]$. Since a_1 is already in the uncertainty range and $E(a_1)$ is already known, a_1 can be made to coincide with b_2. Thus, only one new evaluation of E at a_2 would be necessary.

The question that addresses itself is what value of ρ would be that results in only one evaluation of E. The answer to this question is found in reference [57] where it is proven that when ρ is equal to 0.382, only one evaluation of E is required. Hence, using the Golden section algorithm means that at every stage of the uncertainty range reduction (except the first one), the objective function E need only be evaluated at one new point. The uncertainty range is reduced by the ratio $(1 - \rho) = 0.618$ at every stage. Hence, Ng steps of reduction using the Golden section method reduce the range by the factor

$$(1-\rho)^{Ng} = 0.618^{Ng}$$

Consequently, the following recurrent relations are used to determine the optimal location of the points in the golden section search algorithm:

$$\eta_1 = a_0 + 0.382 l_0$$

$$\eta_2 = b_0 - 0.382 l_0$$

and

$$\begin{aligned} \eta_{k+1} &= a_k + 0.382 l_k \\ &= b_k - 0.382 l_k \end{aligned} \qquad (3.10)$$

where l_k is the length of the kth interval (a_k, b_k).

3.3.1.2 Bracketing Search Algorithm

In this algorithm, the minimum of the function to be minimized is bracketed between two points, through a series of function evaluations. The algorithm begins with an initial point a_o, a function $E(a_o)$, a step size β_o, and a step expansion parameter $\gamma > 1$. The steps of the Bracketing algorithm [57] are outlined as:

1. Evaluate $E(a_o)$ and $E(a_o + \beta_o)$.
2. If $E(a_o + \beta_o) < E(a_o)$, let $a_1 = a_o + \beta_o$ and $\beta_1 = \gamma\beta_o$, and evaluate $E(a_1 + \beta_1)$. Otherwise, go to step 4.
3. If $E(a_1 + \beta_1) < E(a_1)$, let $a_2 = a_1 + \beta_1$ and $\beta_2 = \gamma\beta_1$, and continue incrementing the subscripts this way until $E(a_k + \beta_k) > E(a_k)$. Then go to step 8.
4. Let $a_1 = a_o$ and $\beta_1 = -\xi\beta_o$, where ξ is a constant that satisfies $0 < \xi < 1/\gamma$, and evaluate $E(a_1 + \beta_1)$.
5. If $E(a_1 + \beta_1) > E(a_1)$ go to 7.
6. Let $a_2 = a_1 + \beta_1$ and $\beta_2 = \gamma\beta_1$, and continue incrementing the subscripts this way until $E(a_k + \beta_k) > E(a_k)$. Then, go to step 8.
7. The minimum has been bracketed between points $(a_o - \xi\beta_o)$ and $(a_o + \beta_o)$. Go to step 9.
8. The last three points satisfy the relations $E(a_{k-2}) > E(a_{k-1})$ and $E(a_{k-1}) < E(a_k)$, and hence, the minimum is bracketed.
9. Stop.

3.4 BP Training Procedure with Momentum

BP without momentum is a first order minimization that utilizes gradient information to follow a path of steepest descent. While in theory such a procedure is guaranteed to find a minimum, in practice it depends much on appropriate scaling parameter (LR). Moreover, at points in the weight space where the Hessian matrix has a large condition number, gradient descent is extremely slow to converge because it oscillates from side to side across the minimum [61]. Often, in gradient approaches LR is adjusted as training progresses. This type

of adjustment allows for large initial corrections but does not avoid weight oscillations around the minimum near the solution. To help overcome this problem, a momentum term is added and in this case the step direction is no longer steepest descent but modified by the previous direction [58]. In effect, momentum utilizes second order information but requires only one step memory and uses only local information. This can be seen from the addition of momentum to the output weight update at the $(k+1)$th iteration which results in the following modified correction $\Delta^N w_{sj}^o$ equation:

$$\Delta^N w_{sj}^{o(k+1)} = \sum_{p=1}^{N} \left(\Delta^p w_{sj}^{o(k+1)} + \alpha\, \Delta^p w_{sj}^{o(k)} \right) \tag{3.11}$$

In this equation α is called momentum factor (MF) that needs to be carefully chosen. The second term in Equation (3.11), for positive α, yields a correction at step $(k + 1)$ that is somewhat different from the case if Equation (2.31) alone is used. As will be shown soon, this extension is related to a second-order weight correction technique.

3.4.1 Significant Aspects of Momentum

- In training formulations involving momentum, when $\frac{\partial E^N}{\partial w_{sj}^o}$ has the same algebraic sign on consecutive iterations, $\Delta^N w_{sj}^o$ grows in magnitude and so w_{sj}^o is modified by a large amount. Thus, momentum tends to accelerate descent in steady downhill directions (i.e., giving momentum to the correction).
- In training formulations involving momentum, when $\frac{\partial E^N}{\partial w_{sj}^o}$ has alternating algebraic signs on consecutive iterations, $\Delta^N w_{sj}^o$ becomes smaller and so the weight adjustment is small. Thus, momentum has a stabilizing effect on learning.

Thus, momentum may prevent oscillations in the system and help the system escape local error-function minima in the training process by making the system less sensitive to local changes. Momentum value selection, α, is much like the learning rate in that it is peculiar to specific error surface contours. Also, if the system enters a local minimum that is steep on one

side and less so on another, the momentum built up during entry into the minimum may be enough to push it back out.

Furthermore, it is noted that the use of E^P versus E^N in training causes the weight corrections to be averaged over *H*. Momentum, on the other hand, causes weight corrections to be averaged over *time (iteration)*.

3.5 Second-Order Methods

In these approaches the following term is not ignored, as in the first-order methods:

$$\frac{1}{2!}(\underline{w}-\underline{w}_o)^T \left.\frac{\partial^2 E^N}{\partial \underline{w}^2}\right|_{w=\underline{w}_o} (\underline{w}-\underline{w}_o)$$

3.5.1 Approximation of Local Shape of the Epoch Mapping Error

Assume that around $\underline{w} = \hat{\underline{w}}$

$$\left.\frac{\partial E^N}{\partial \underline{w}}\right|_{w=\hat{w}} = 0 \tag{3.12}$$

The local shape of Equation (3.2) is then dominated by the second term:

$$E^N(\underline{w}) = E^N(\hat{\underline{w}}) + \frac{1}{2!}(\underline{w}-\hat{\underline{w}})^T \left.\frac{\partial^2 E^N}{\partial \underline{w}^2}\right|_{w=\hat{w}} (\underline{w}-\hat{\underline{w}}) \tag{3.13}$$

The shape of the error surface in this region is that of a hyperparaboloid, and the loci of contours of constant E^N are hyperellipsoids.

3.5.2 Second-Order Equations and the Hessian Matrix

To begin, rewrite Equation (3.2):

$$E^N(\underline{w}) = E^N(\underline{w}_o) + g|_{\underline{w}=\underline{w}_o}(\underline{w}-\underline{w}_o)$$

$$+\frac{1}{2!}(\underline{w}-\underline{w}_o)^T Q|_{\underline{w}=\underline{w}_o}(\underline{w}-\underline{w}_o) + \text{ higher order terms} \tag{3.14}$$

where the notation $|_{\underline{w}} = \underline{w}_o$ is used as a reminder that quantities are evaluated at the point about which the linearization takes place. In addition, the following vector and matrix are defined:

$$g = \frac{\partial E^N}{\partial \underline{w}} \tag{3.15}$$

$$Q = \frac{\partial^2 E^N}{\partial \underline{w}^2} \tag{3.16}$$

The vector g in Equation (3.15) is the familiar gradient, and the matrix Q of second partial derivatives in Equation (3.16) is referred to as the *Hessian* matrix.

In the second-order formulation of Equation (3.14), minima are located where the gradient of Equation (3.14) is equal to zero. Differentiation of Equation (3.14) with respect to $\underline{w}$ yields

$$\frac{\partial E^N}{\partial \underline{w}} = g + Q\underline{w} - Q\underline{w}_o \tag{3.17}$$

where g and Q are evaluated at $\underline{w} = \underline{w}_o$. From Equations (3.17) and (3.12), we establish that

$$\hat{\underline{w}} = \underline{w}_o - Q^{-1} g \tag{3.18}$$

which shows that the optimal weight vector may be found using second-order techniques if g (the gradient) and the Hessian Q are available. Unfortunately, the calculation of Q is prohibitively expensive, and the inversion of this (typically) large matrix further complicates the issue. Therefore, we resort to approximation techniques.

3.5.3 Conjugate Gradient (CG) Methods

In conjugate gradient methods, successive steps in $\Delta\underline{w}$ are chosen such that the learning at previous steps is preserved. A direction for the weight correction is computed and line minimization in this direction is performed, generating $\underline{w}^{(k+1)}$. Successive weight corrections are constrained to be *conjugate* to those used previously. Interestingly, this is achieved without inversion of Q, as in Equation (3.18).

Denote the *weight correction direction* in weight space for minimization of E at iteration k as $\underline{d}^{(k)}$ and the gradient of $E^N(\underline{w}^{(k)})$ as $g^{(k)}$. For example, in the back-propagation algorithm $\underline{d}^{(k)}$ is equal to $-g^{(k)}$. The traditional conjugate gradient updates the weights for the kth iteration as follows:

$$\underline{w}^{(k+1)} = \underline{w}^{(k)} + \eta^{(k)} \underline{d}^{(k)} \tag{3.19}$$

where $\eta^{(k)}$ is the step size, a scalar which minimizes the objective function in the conjugate direction $\underline{d}^{(k)}$.

Up to this point we have not said how the conjugate directions $\underline{d}^{(k)}$ are found. Several techniques [62] exist for the determination of the conjugate directions $\underline{d}^{(k)}$. Suppose the initial weight correction direction is, $\underline{d}^{(0)} = -g^{(0)}$. Beginning with $k = 0$, we require subsequent gradients to be orthogonal to the previous weight correction direction; that is,

$$\underline{d}^{(k)T} g^{(k+i)} = 0 \qquad i = 1, 2, \ldots\ldots, k_{max} \tag{3.20}$$

which yields

$$\underline{d}^{(k)T} (g^{(k+2)} - g^{(k+1)}) = 0 \tag{3.21}$$

Notice that $(g^{(k+2)} - g^{(k+1)})$ is the change of gradient of E from $\underline{w}^{(k+1)}$ to $\underline{w}^{(k+2)}$. From Equation (3.17),

$$g^{(k)} = Q(\underline{w}^{(k)} - \underline{w}_o) + g\big|_{\underline{w}=\underline{w}_o} \tag{3.22}$$

and thus, Equations (3.21) and (3.22) may be manipulated to yield

$$\underline{d}^{(k)T} Q(\underline{w}^{(k+2)} - \underline{w}^{(k+1)}) = 0 \tag{3.23}$$

Since $\Delta\underline{w}^{(k+1)} = (\underline{w}^{(k+2)} - \underline{w}^{(k+1)}) = d^{(k+1)}$, we get

$$\underline{d}^{(k)T} Q\underline{d}^{(k+1)} = 0 \tag{3.24}$$

Equation (3.24) may be rewritten as

$$\left\langle \underline{d}^{(k+2)}, \left[Q\underline{d}^{(k+1)}\right]\right\rangle = 0 \tag{3.25}$$

Hence, weight correction directions $\underline{d}^{(k+2)}$ and $\underline{d}^{(k+1)}$ are said to be *conjugate to one another in the context of* Q, or Q-conjugate.

There exist several ways to find the conjugate vectors required in Equation (3.24). For an $n_w \times n_w$ matrix Q, iterative techniques are available to find the n_w solutions to Equation (3.24). However, because of the dimensions of Q and the effort involved in computing Q, we choose alternative techniques. Instead, consider the following weight space search strategy:

$$\underline{d}^{(k+1)} = -g^{(k+1)} + \beta^{(k+1)} \underline{d}^{(k)} \tag{3.26}$$

where β is a weight factor for the previous directions. Equation (3.26) defines a path or sequence of search directions $\underline{d}^{(k)}$, $k = 0, 1, \ldots, n_w$, in weight space, where each weight correction direction vector $\underline{d}^{(k)}$ is

- Computed sequentially, beginning with $\underline{d}^{(0)}$
- Different from that found using gradient descent, where $\underline{d}^{(k)} = - g^{(k)}$
- Formed as the sum of the current (negative) gradient direction and a scaled version of the previous correction.

Forcing Equation (3.26) to satisfy Equation (3.24) yields

$$\underline{d}^{(k)T} Q \underline{d}^{(k+1)} = -\underline{d}^{(k)T} Q g^{(k+1)} + \underline{d}^{(k)T} Q \beta^{(k+1)} g^{(k)} = 0 \tag{3.27}$$

from which $\beta^{(k+1)}$ is found as

$$\beta^{(k+1)} = \frac{\underline{d}^{(k)T} Q g^{(k+1)}}{\underline{d}^{(k)T} Q g^{(k)}} \tag{3.28}$$

Although Equation (3.28) seems to require Q for the computation of β, numerous algebraically equivalent forms exist. For instance, when using the *Fletcher-Reeves* formulation [57], $\beta^{(k+1)}$ is given by

$$\beta^{(k+1)} = \frac{g^{(k+1)T} g^{(k+1)}}{g^{(k)T} g^{(k)}} \tag{3.29}$$

Whereas, when using the *Polak-Ribière*'s formulation [57], $\beta^{(k+1)}$ is given by

$$\beta^{(k+1)} = \frac{g^{(k+1)T} \left(g^{(k+1)} - g^{(k)} \right)}{g^{(k)T} g^{(k)}} \tag{3.30}$$

It can be shown that if the conjugate directions just discussed may be found and the approximation of E^N by a second-order Taylor series is precise, then the minimum will be found by this method in n_w steps, where n_w is the number of weights. The proof is relatively simple, and makes use of the fact that the search directions, $\underline{d}^{(k)}$, are linearly independent. Starting with $- g^{(0)}$ as the first weight correction direction, after n_w applications of Equations (3.26) and (3.29) or (3.30), n_w conjugate directions would be found. Referring to Equation (3.20), the next gradient must be orthogonal to the n_w previously computed $\underline{d}^{(k)}$, or at least satisfy this equation. In a vector space of dimension n_w, there are at most n_w linearly independent vectors. Therefore, the only way for Equation (3.20) to be satisfied is for

$g^{(n_w+1)} = 0$; i.e., the minimum is reached. This shows that the conjugate gradient approach has quadratic convergence. It should be kept in mind that the conjugate gradient strategy developed in this section was based upon the assumption that the mapping error function was quadratic; i.e., the higher-order terms were neglected.

3.5.4 Comments on CG Methods

Performance comparisons of the standard BP and the traditional CG methods seem to be task dependent. For example, in comparing percentage of trials that converged on a global minimum, Van der Smagt [59] reported that the *Fletcher-Reeves* CG method [57] was not as good as standard BP on the XOR task but was better than standard BP on two function estimation tasks. A somewhat similar result is reported by Aylward *et al.* [63]. They reported that faster learning is obtained by CG on a function estimation problem but a slower learning (compared to standard BP) is obtained on a task involving classification of handwritten numerals. Since the performance of the CG methods when applied to matrix computations is not known, it is desirable to investigate this issue in the present study.

Another point of comparison between methods is their ability to reduce error on learning of the training set. De Groot and Wurtz [64] reported that CG was able to reduce error on a function estimation problem some 1000 times smaller than standard BP in 10s of CPU time. Barnard [65] compared the CG method and the standard BP method without momentum on three different classification tasks. He found that the CG method is able to reduce the error more rapidly and to a lower value than the BP method for a given number of iterations. In another study by Barnard and Holm [66] the CG method was compared with a first order adaptive step size algorithm. They found that the CG method achieves lower errors and higher generalization in the early iterations but the adaptive algorithm gradually approached the same performance with extra iterations. Hence, in the present study, a comparison between algorithms is necessary to determine their ability to achieve small errors on matrix algebra problems.

Traditional CG method restarts the search direction every n_w iterations (where n_w is the number of weights in the network) by setting β=0, that is, resetting the search direction to the negative gradient. It also assumes exact line searches in order to calculate step size. Since this

restart rule is inefficient because a step in the steepest descent direction seldom reduces the error by much and since exact line searches are computationally expensive, much effort has gone into developing more efficient algorithms.

An improvement involving restarts was suggested by Beale [67]. It uses equation (3.26) for the first n_w iterations and for restart steps, but for subsequent non-restart steps the search direction includes an extra term to provide conjugacy to the previous restart direction. Additional restart criteria have been suggested by *Powell* [68] to improve the speed, that is restart when

$$\left| g^{(k+1)^T} g^{(k)} \right| \geq 0.2 \left\| g^{(k+1)} \right\|^2 \tag{3.31}$$

or

$$-1.2 \left\| g^{(k+1)} \right\|^2 \leq \underline{d}^{(k+1)} g^{(k+1)} \leq -0.8 \left\| g^{(k+1)} \right\|^2 \tag{3.32}$$

Equation (3.31) ensures orthogonality between consecutive gradient vectors and equation (3.32) ensures that the new search direction is sufficiently downhill. The use of both restart criteria showed promising results in training of multi-layer perceptrons on function estimation problems [59].

Since most of the computational burden in algorithms involves the line search, it would be an advantage to avoid line searches by calculating the step size analytically. Moller [69] has introduced an algorithm that does this, making use of gradient difference information. He reported that the algorithm is about twice as fast as the traditional CG method on a parity problem, partly due to a reduction in the number of iterations. Despite of this computational savings, this algorithm has a drawback of incorporating arbitrary tuning parameters whose values are critical to performance.

While higher-order approaches attempt to find a minimum in E^N efficiently, notice that the computational cost at each iteration may be significantly increased. Thus, while a (slightly) better correction at each step may be achieved, the overall computational cost may actually be higher, when compared with simply using first-order techniques. This partially explains why second-order techniques have not found widespread adoption. Nevertheless, in this study, the *Fletcher-Reeves* CG (CGFR) method, the variant of CG method given by equation (3.29), and

Polak-Ribière CG (CGPR) method (equation (3.30)) will be examined and assessed when applied to matrix computation problems.

3.5.5 Relation between CG Method and Momentum Formulations

The conjugate gradient-based strategy of Equation (3.26) shows a strong similarity to the momentum formulation of Equation (3.11). The first-order *gradient descent-with-momentum* approach produces weight correction of the form

$$\Delta^N w_{sj}^{o(k+1)} = -\eta \left(\frac{\partial E^N}{\partial w_{sj}^o} \right)^{(k+1)} + \alpha \, \Delta^N w_{sj}^{o(k)} \tag{3.33}$$

which is equivalent to the correction of Equation (3.26) if the learning rate η and momentum coefficient α were chosen to make the weight search directions conjugate. In other words, when comparing the traditional CG to the standard BP with momentum, the difference is that in the latter both step size and momentum are fixed, while in the former the step size is computed by a line search and $\beta^{(k+1)}$ is computed analytically.

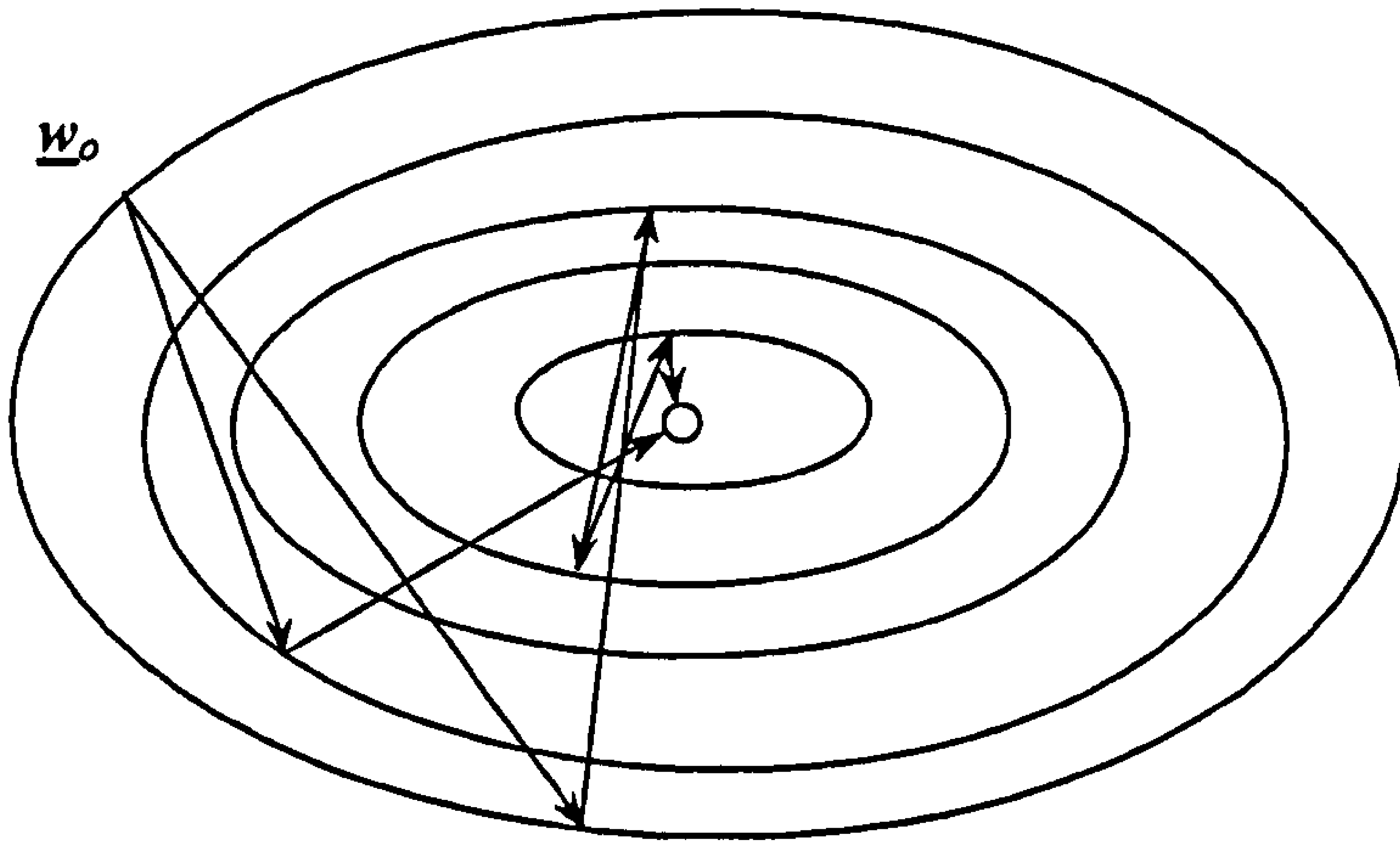
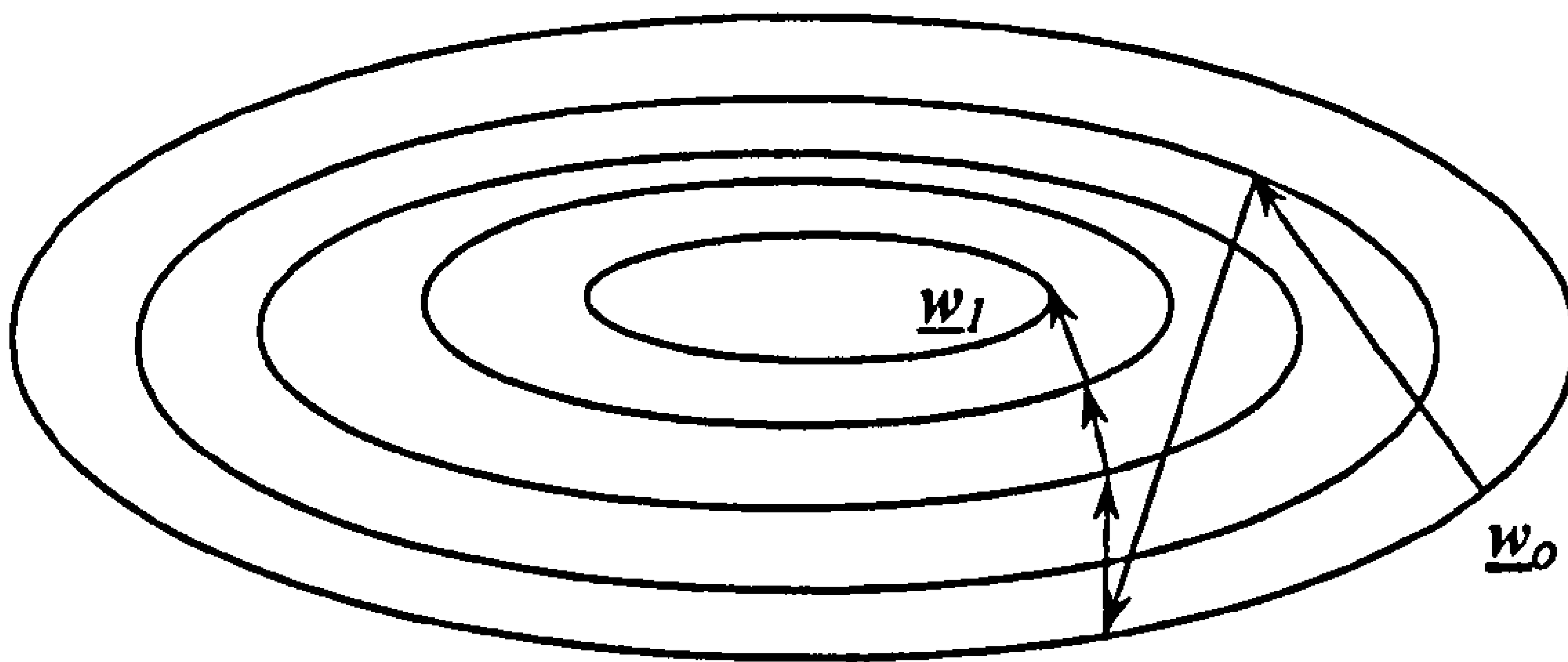
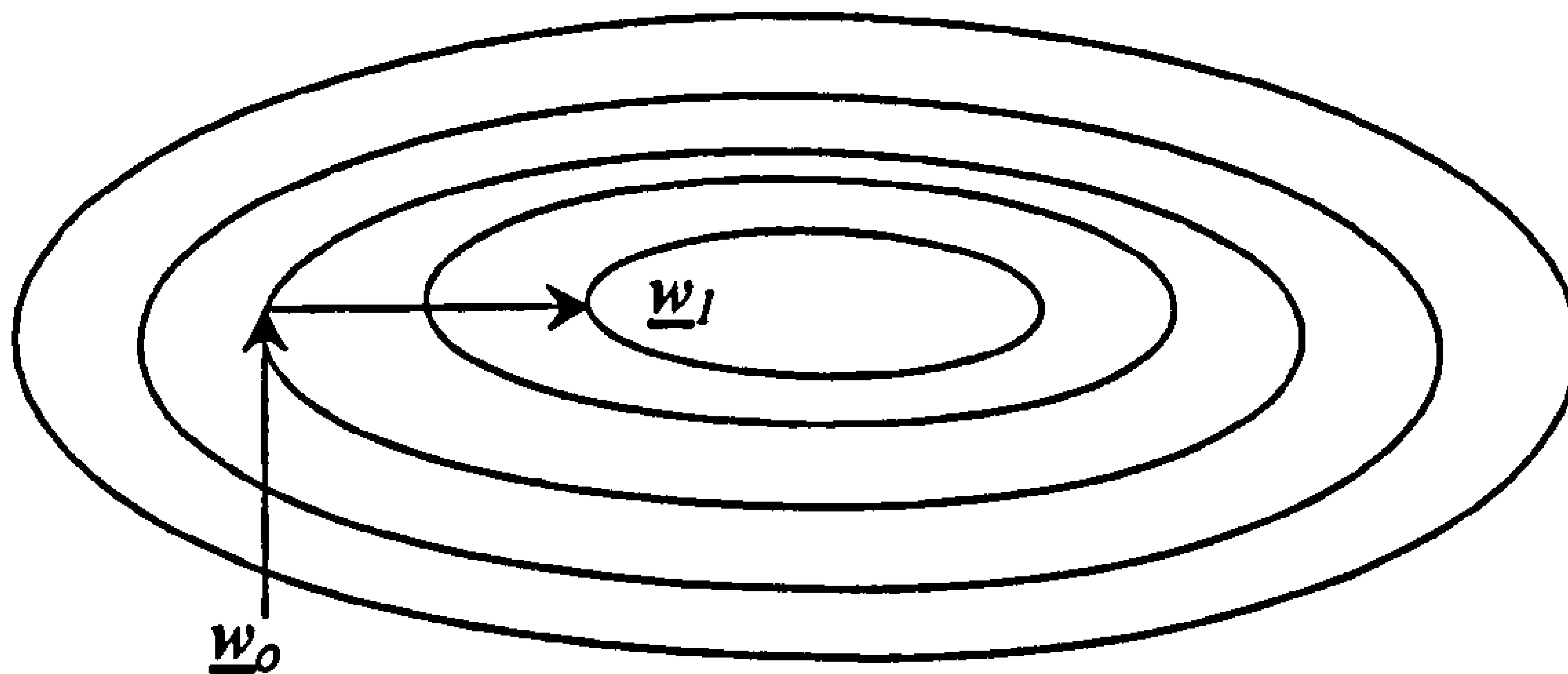


Figure 3.1 Two very different trajectories for weight correction



(a)



(b)

Figure 3.2 Conjugate versus gradient directions weight correction:
(a) Gradient-guided search with constant step size;
(b) Conjugate directions leading directly to the minimum.

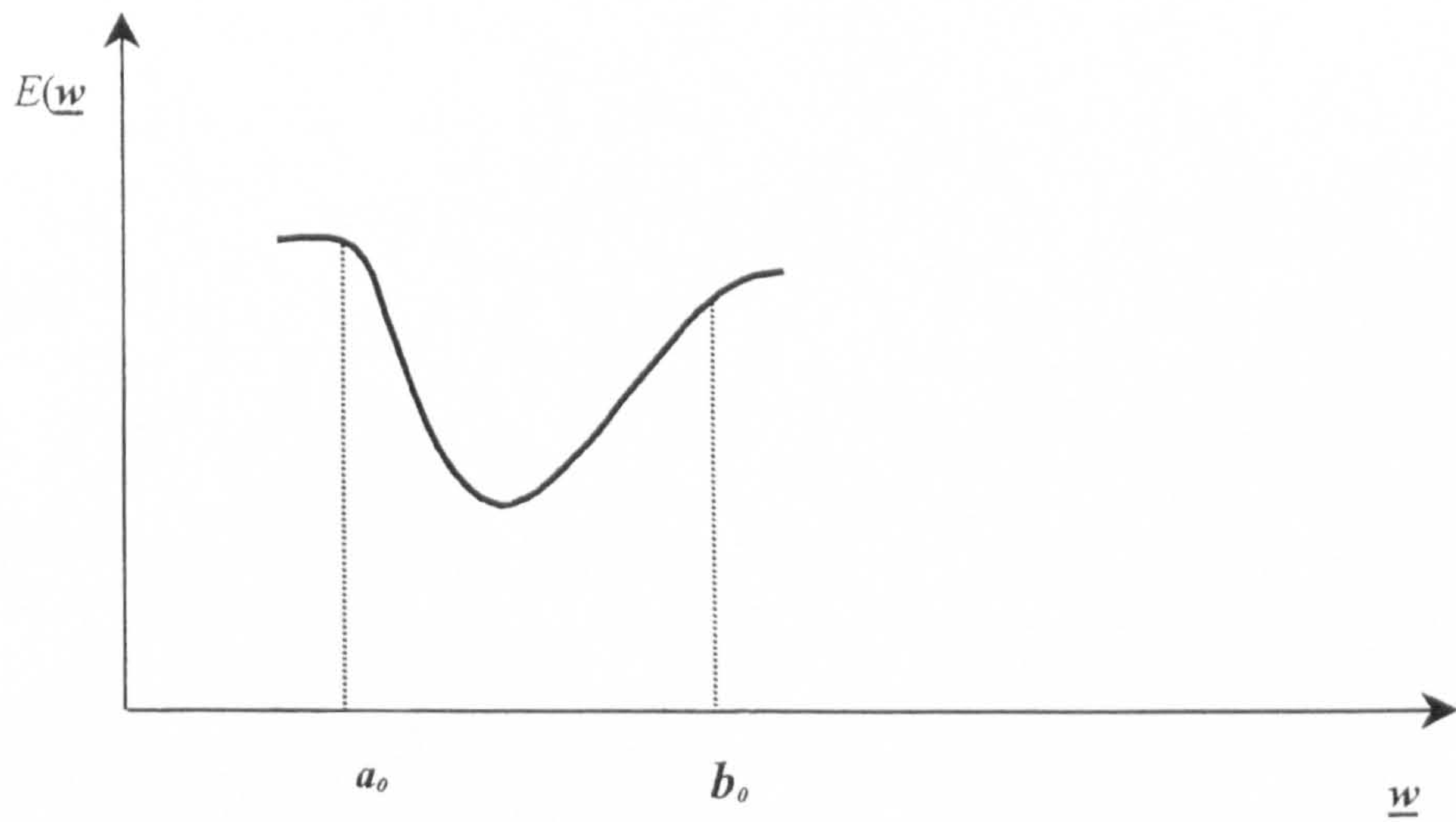


Figure 3.3 A typical unimodal function

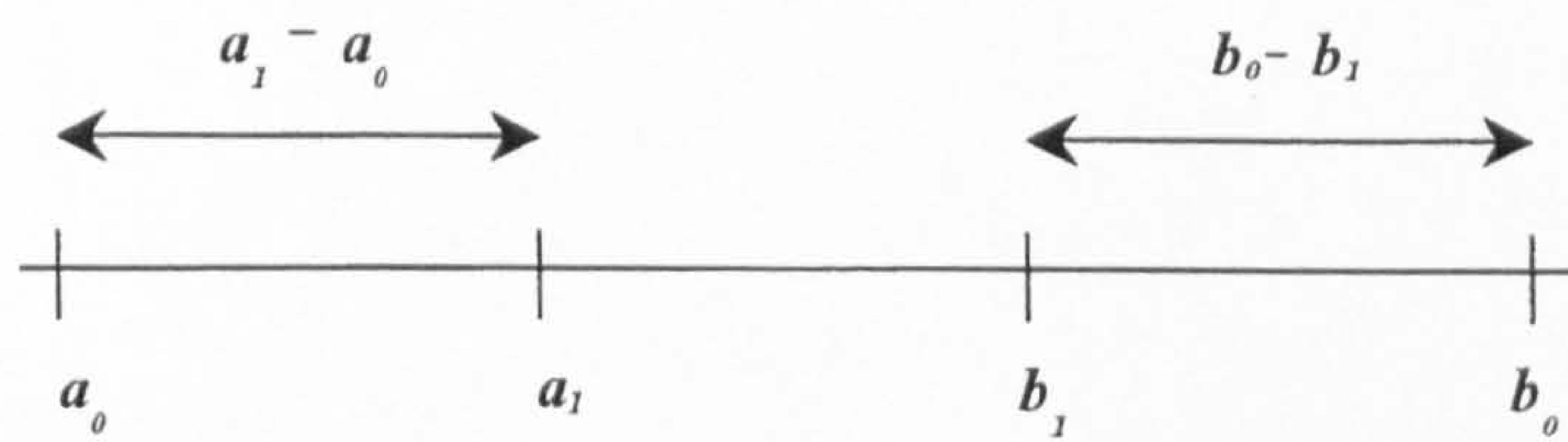


Figure 3.4 Evaluating the objective function at two intermediate points.

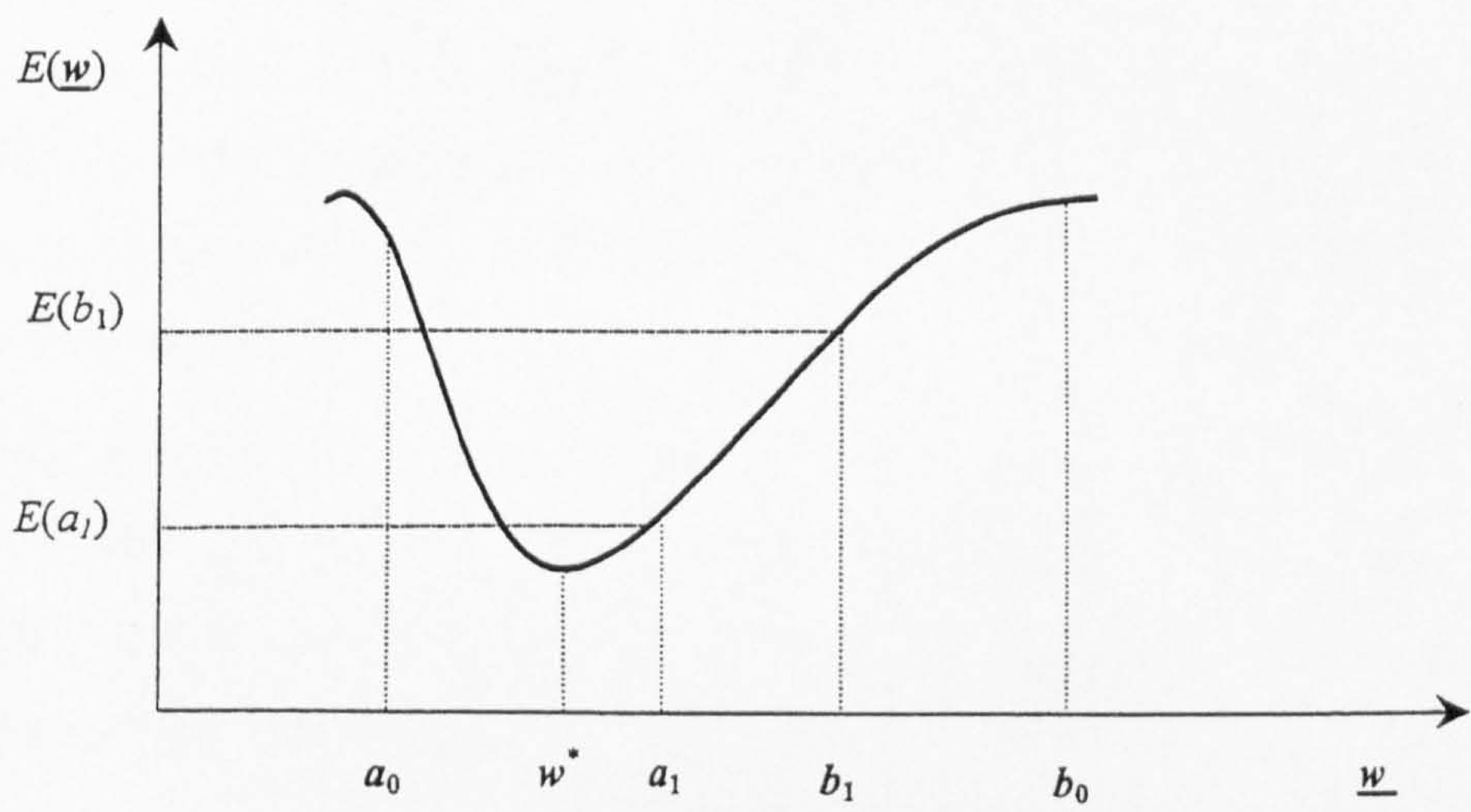


Figure 3.5. The case when $E(a_1) < E(b_1)$; The minimizer $\underline{w}^* \in [a_0, b_1]$

CHAPTER 4

FNNs for Large-Scale Matrix Algebra Problems

To analyze the performance of the minimization methods discussed in chapter 3 when applied to matrix computations, only two matrix algebra problems are considered in the present study. These problems are matrix LU-decomposition and matrix inversion. It should be emphasized here that the performance of these minimization methods would be assessed on large-scale problems as compared to the small-scale matrix problems considered previously by Wang and Mendel [21-23].

Included in this chapter are presentations of the feedforward neural networks used for the above matrix computations. The training procedures for these FNNs under the minimization methods discussed in chapter 3 are also included. The potential for parallelisation of these procedures is also discussed in this chapter.

4.1 FNN for LU-Decomposition Problem

The first test problem considered in this study is the matrix LU-decomposition. Following the work of Wang and Mendel [21-23], this problem can be formulated as follows:

Given a matrix $A \in R^{N \times N}$, the task is to obtain a lower-triangular matrix (with unity diagonal elements) $L \in R^{N \times N}$ and an upper-triangular matrix $U \in R^{N \times N}$ such that $LU = A$. Consider the two-layer linear FNN of Figure 4.1. We constrain the connections between the inputs and the lower-layer neurons so that the output of the lower-layer, z, equals Ux, where x is a given input vector and U is an upper-triangular matrix. Similarly, we constrain the connections between the lower-layer and the upper-layer neurons so that the network output, y, will be LUx, where L is a lower-triangular matrix with all diagonal elements equal to unity. Since the desired overall transformation $LU = A$ is known, the desired output Ax is also known when network input x is specified. Hence, (x, Ax) constitute the training patterns.

There are two basic issues to implementing this structured network:

1. How to choose the input patterns (the x's) so that if their associated patterns are matched, the weight matrices give the solution to the problem; and
2. How to train this linear FNN.

The following Theorem that is given by Wang and Mendel [23] is used to tackle the first issue:

For a given matrix $A \in R^{N \times N}$ *and the network of Figure* 4.1, *we choose N desired input patterns* $x^{(1)}, x^{(2)}, \ldots\ldots, x^{(N)}$ *which are any linearly independent* $N \times 1$ *vectors, and choose the corresponding desired output patterns to be* $Ax^{(1)}, Ax^{(2)}, \ldots\ldots, Ax^{(N)}$. *If the network matches all the n input-output pattern pairs* $(x^{(i)}, Ax^{(i)})$ $(i = 1, 2, \ldots\ldots, N)$, *then the final weights of the lower-layer neurons give the desired* **U** *matrix, and the final weights of the upper-layer neurons give the desired* **L** *matrix, i.e.,* $LU = A$.

On the basis of the above Theorem the input and associated patterns are chosen for the *LU*-decomposition linear FNN to be (e_i, Ae_i) $(i = 1, 2, \ldots, N)$, where e_i is the ith unit vector, i.e., all elements of e_i equal zero except the ith element which equals unity. The purpose is to train the network to match these patterns.

The second issue is very important in the network implementation and needs special attention since the ***LU*** solution is a byproduct of the training. As a matter of fact, the performance of the training algorithm highly affects the success of the *LU*-decomposition. As stated earlier, the BP algorithm described in section 2.4 suffers a lot of problems as a result of the improper choice of the scaling parameter or the learning rate, η, e.g., slow convergence and the possibility of converging at a local minimum. These problems could be overcome either by using the Steepest Descent with Line Search (SDLS) method or by using the Conjugate Gradient (CG) methods. The training algorithms under these methods are developed below.

4.1.1 Training Formulation for *LU*-Decomposition FNN under SDLS Method

In case of adopting the steepest descent method as described in section 2.3.1, the update equations (2.20) and (2.22) need to be modified by replacing w_{sj}^{o} with l_{sj} *and* w_{ji}^{h} with u_{ji}. Specifically, to update the upper-layer neuron weights l_{sj}, the following equation is used:

$$l_{sj}^{(k+1)} = l_{sj}^{(k)} + \Delta l_{sj}^{(k)}, \tag{4.1}$$

The gradient of the mapping error function with respect to the weights l_{sj} is given by

$$\left(\nabla_{l_s}\left(E^{p(k)}\right)\right)_{sj} = -\delta_s^{(k)} z_j^{(k)}, \tag{4.2}$$

The weight correction $\Delta l_{sj}^{(k)}$ in equation (4.1) is set proportional to the negative of gradient as follows:

$$\Delta l_{sj}^{(k)} = \eta \delta_s^{(k)} z_j^{(k)}, \tag{4.3}$$

where

$$z_j^{(k)} = \sum_{i=1}^{n} u_{ji}^{(k)} x_{di}, \tag{4.4}$$

$$\delta_s^{(k)} = (y_{ds} - y_s^{(k)}), \tag{4.5}$$

$$y_s^{(k)} = \sum_{j=1}^{l} l_{sj}^{(k)} z_j^{(k)}. \tag{4.6}$$

In the above equations, y_s is the actual output of the *s*th neuron of the upper layer, y_{ds} is the corresponding desired output (which equals the *i*th component of Ax), z_j is the output of the *j*th neuron of the lower layer, and k denotes the steps of updatings (k = 0, 1, 2,..., k_{max}). To update the lower-layer neuron weights u_{ji}, the following equation is used

$$u_{ji}^{(k+1)} = u_{ji}^{(k)} + \Delta u_{ji}^{(k)}, \tag{4.7}$$

The gradient of the mapping error function with respect to the weights u_{ji} is given by

$$\left(\nabla_{u_i}\left(E^{p(k)}\right)\right)_{ji} = -\left[\sum_{s=1}^{m} \delta_s^{(k)} l_{sj}^{(k)}\right] x_i, \tag{4.8}$$

Hence, the weight correction $\Delta u_{ji}^{(k)}$ in equation (4.7) is set proportional to the negative of gradient as follows:

$$\Delta u_{ji}^{(k)} = \eta \left[\sum_{s=1}^{m} \delta_s^{(k)} l_{sj}^{(k)} \right] x_i \quad (4.9)$$

Since the connections are constrained, some weights are not updated and are kept constants. These weights are

$$l_{ss} = 1,$$

$l_{sj} = 0$ for $s = 1, 2, ..., m,$ $j = s+1, s+2, ..., l;$

$u_{ji} = 0$ for $j = 2, 3, ..., n,$ $i = 1, 2, ..., j\text{-}1.$

Notice that in the present case, the index m is equal to n and l, and they are all equal to the order of the matrix (N). The optimum value of learning rate, η, in equations (4.3) and (4.9) is obtained at each updating step via the hybrid line search method described in chapter 3.

4.1.2 Training Procedure for *LU*-Decomposition FNN under SDLS Method

The training procedure comprises one forward computation of $z = Ux$ and $y = Lz$; and one computation of the equations (4.1) and (4.7), i.e., one backward training iteration. Accordingly, the SDLS training procedure using epoch (batch) training strategy is summarized in the following steps and its flow chart is shown in figure 4.2:

1. Start with $k = 0$ and set the initial $L^{(0)}$ and $U^{(0)}$ to zero matrices.
2. Present an input-desired output pair P, and do steps (2.a)-(2.c) for all $P \in [1, N]$, to compute the total (epoch) sum of squared error, $E^{N(k)}$, and the total gradient, $g^{(k)}$, using equations (2.29) and (2.30), respectively, where in this case, $(y_{d,1}^p, ..., y_{d,m}^p)^T = Ae_p$ is the Pth desired output vector, and $(y_1^p, ..., y_m^p)^T$ is the actual output vector for input e_p. Notice that the patterns are presented to the network in the following order: $(e_1, Ae_1), (e_2, Ae_2), ..., (e_N, Ae_N)$. Where, N is equal to the matrix order (n).
   a. Compute z and y.
   b. Calculate the sum of squared error, $E^{p(k)}$, given by equation (2.14).
   c. Calculate the gradient $\nabla E^{p(k)}$ for this pattern from equations (4.2) and (4.8).
3. Check the epoch based-error, $E^{N(k)} < \varepsilon_1$, or $\| g^{(k)} \| < \varepsilon_2$. If one of these conditions is met, go to step 8.

4. Otherwise, set the search direction $\underline{d}^{(k)} = - g^{(k)}$.
5. Perform line search along $\underline{d}^{(k)}$ using the hybrid method described in chapter 3 to determine the optimal value $\eta^{(k)}$ that minimizes $E^N(\eta^{(k)}, L^{(k)}, U^{(k)})$.
6. Update $L^{(k+1)}$ and $U^{(k+1)}$ using equations (4.1) and (4.7), respectively.
7. Set $k = k+1$. If $k > k_{max}$, go to step 8, otherwise, repeat from step 2.
8. Stop.

4.1.3 Training Formulation for *LU*-Decomposition FNN under CG Methods

In case of adopting the conjugate gradient method as described in section 3.3, the update equations (2.20) and (2.22) need to be modified by replacing w^o_{sj} with l_{sj} *and* w^h_{ji} with u_{ji}. Specifically, to update the upper-layer neuron weights l_{sj}, the following equation is used:

$$l_{sj}^{(k+1)} = l_{sj}^{(k)} + \Delta l_{sj}^{(k)} \qquad (4.10)$$

The weight correction $\Delta l_{sj}^{(k)}$ in equation (4.10) is set proportional to the search direction as follows:

$$\Delta l_{sj}^{(k)} = \eta \underline{d}^{(k)} \qquad (4.11)$$

The search direction $\underline{d}^{(k)}$ in this equation is determined using one of the following equations:

$$\underline{d}^{(k)} = -\underline{g}^{(k)} \qquad k = 0 \qquad (4.12)$$

$$\underline{d}^{(k+1)} = -\underline{g}^{(k+1)} + \beta^{(k+1)} \underline{d}^{(k)} \qquad k > 0 \qquad (4.13)$$

The gradient of the mapping error function with respect to the weights l_{sj}, $g^{(k)}$, is given by equation (4.2). The value of $\beta^{(k+1)}$ in equation (4.13) is obtained using one of the following equations:

$$\beta^{(k+1)} = \frac{\left(\underline{g}^{(k+1)}\right)^T \left(\underline{g}^{(k+1)}\right)}{\left(\underline{g}^{(k)}\right)^T \left(\underline{g}^{(k)}\right)} \qquad (4.14)$$

Fletcher-Reeves Equation [57]

$$\beta^{(k+1)} = \frac{\left(\underline{g}^{(k+1)}\right)^T \left(\underline{g}^{(k+1)} - \underline{g}^{(k)}\right)}{\left(\underline{g}^{(k)}\right)^T \left(\underline{g}^{(k)}\right)}$$

Polak - Ribiere Equation [68]

(4.15)

Similarly, the lower-layer neuron weights, u_{ji}, are updated using the following equation:

$$u_{ji}^{(k+1)} = u_{ji}^{(k)} + \Delta u_{ji}^{(k)} \qquad (4.16)$$

The weight correction $\Delta u_{ji}^{(k)}$ in equation (4.16) is set proportional to the search direction as follows:

$$\Delta u_{ji}^{(k)} = \eta \underline{d}^{(k)} \qquad (4.17)$$

The search direction $d^{(k)}$ in this equation is determined using either equation (4.12) or equation (4.13). The gradient of the mapping error function with respect to the weights u_{ji}, $g^{(k)}$, in these equations is given by equation (4.8).

The optimum value of learning rate, η, in equations (4.11) and (4.17) is obtained at each updating step via the hybrid line search method described in chapter 3.

4.1.4 Training Procedure for *LU*-Decomposition FNN under CG Methods

The training procedure comprises one forward computation of $z = Ux$ and $y = Lz$; and one computation of the equations (4.10) and (4.16), i.e., one backward training iteration. Accordingly, the CG training procedure using epoch (batch) training strategy is summarized in the following steps and its flow chart is shown in figure 4.3:

1. Start with $k = 0$ and set the initial $L^{(0)}$ and $U^{(0)}$ to zero matrices.
2. Present an input-desired output pair P, and do steps (2.a)-(2.c) for all $P \in [1, N]$, in order to compute the total (epoch) sum of squared error, $E^{N(k)}$, and the total gradient, $g^{(k)}$, using equations (2.29) and (2.30), respectively, where in this case, $(y_{d,1}^{p}, \ldots\ldots, y_{d,m}^{p})^T = Ae_p$ is the pth desired output vector, and $(y_1^p, \ldots\ldots, y_m^p)^T$ is the actual output vector for input e_p. Notice that the patterns are presented to the network in the following order: $(e_1, Ae_1), (e_2, Ae_2), \ldots, (e_N, Ae_N)$. Where, N is equal to the matrix order (n).
   a. Compute z and y.
   b. Calculate the sum of squared error, $E^{p(k)}$, given by equation (2.14).
   c. Calculate the gradient $\nabla E^{p(k)}$ for this pattern from equations (4.2) and (4.8).

3. Check the epoch based-error, $E^{N(k)} < \varepsilon_1$, or $\| g^{(k)} \| < \varepsilon_2$. If one of these conditions is met, go to step 13.
4. Otherwise, set the search direction $\underline{d}^{(k)} = - g^{(k)}$.
5. Perform line search along $\underline{d}^{(k)}$ using the hybrid method described in chapter 3 to determine the optimal value $\eta^{(k)}$ that minimizes $E^N(\eta^{(k)}, L^{(k)}, U^{(k)})$.
6. Update $L^{(k+1)}$ and $U^{(k+1)}$ using equations (4.10) and (4.16), respectively.
7. Set $k = k+1$. If $k > k_{max}$, go to step 13.
8. Otherwise, present an input-desired output pair P, and perform steps (8.a)-(8.c) for all $P \in [1, N]$, to compute the total (epoch) sum of squared error, $E^{N(k)}$, and the total gradient, $g^{(k)}$, using equations (2.29) and (2.30), respectively
   a. Compute z and y.
   b. Calculate the sum of squared error, $E^{p(k)}$, given by equation (2.14).
   c. Calculate the gradient $\nabla E^{p(k)}$ for this pattern from equations (4.2) and (4.8).
9. Check the epoch based-error, $E^{N(k)} < \varepsilon_1$, or $\|g^{(k)}\| < \varepsilon_2$. If one of these conditions is met, go to step 13.
10. Otherwise, check *Powell*'s restart criteria given by equations (3.31) and (3.32). If one of these conditions is met, set $\underline{d}^{(k)} = -g^{(k)}$, go to step12.
11. Otherwise, set the search direction $\underline{d}^{(k)} = - g^{(k)} + \beta^{(k)} \underline{d}^{(k-1)}$, where $\beta^{(k)}$ is given by equation (4.14) in case of *Fletcher-Reeves* (CGFR) variant of the method and is given by equation (4.15) in case of *Polak-Ribiere* (CGPR) variant of the method.
12. Repeat from step 5.
13. Stop.

4.2 FNN for the Inversion of Large Scale Matrix

The second matrix computation problem considered in this study is the Matrix Inversion. Following the work of Wang and Mendel [21-23], this problem can be formulated as follows:

Given an invertible matrix $A \in R^{N \times N}$, the task is to determine $B \in R^{N \times N}$ such that $BA = I$; i.e., $(B = A^{-1})$. Consider the two-layer structured network of figure 4.4, where the lower layer performs the transformation A and the upper layer performs the transformation B. According

to $BA = I$, the desired overall transformation is $I_{N \times N}$. Since A is given, the lower layer weights are fixed at A and do not change in the training procedure.

The following Theorem given by Wang and Mendel [23] is used to choose the desired patterns:

For invertible $A \in R^{N \times N}$ and the network of Figure 4.4, we choose the following n pairs of patterns: $(x^{(i)}, x^{(i)})$) where $i = 1, 2, \ldots, N$ and $x^{(i)}$'s are linearly independent $N \times 1$ vectors. If the structured network matches all these n pattern pairs, the upper-layer weights B will satisfy $BA = I_{N \times N}$.

Therefore, on the basis of the above Theorem the desired patterns are chosen for the matrix inversion linear FNN as (e_i, e_i) $(i = 1, 2, \ldots, N)$, where e_i is the ith unit vector, i.e., all elements of e_i equal zero except the ith element which equals unity. Once again, the purpose is to train the network to match these patterns.

4.2.1 Training Formulation for Matrix Inversion FNN under SDLS Method

In case of adopting the steepest descent algorithm as described in section 2.3.1, the update equation (2.20) needs to be modified by replacing w_{sj}° with b_{sj}. Specifically, to update the upper-layer neuron weights b_{sj}, the following equation is used:

$$b_{sj}^{(k+1)} = b_{sj}^{(k)} + \Delta b_{sj}^{(k)} \tag{4.18}$$

The gradient of the mapping error function with respect to the weights b_{sj} is given by

$$\left(\nabla_{b_s}\left(E^{p(k)}\right)\right)_{sj} = -\delta_s^{(k)} z_j^{(k)}, \tag{4.19}$$

Hence, the weight correction $\Delta b_{sj}^{(k)}$ in equation (4.18) is set proportional to the negative of gradient as follows:

$$\Delta b_{sj}^{(k)} = \eta \delta_s^{(k)} z_j^{(k)}, \tag{4.20}$$

where

$$z_j^{(k)} = \sum_{i=1}^{n} a_{ji}^{(k)} x_i, \tag{4.21}$$

$$\delta_s^{(k)} = (y_{ds} - y_s^{(k)}), \tag{4.22}$$

$$y_s^{(k)} = \sum_{s=1}^{l} b_{js}^{(k)} z_s^{(k)}. \tag{4.23}$$

In the above equations, y_s is the actual output of the sth neuron of the upper layer, y_{ds} is the corresponding desired output which equals the sth component of the input x, z_j is the output of the jth neuron of the lower layer, and k denotes the steps of updating ($k = 0, 1, 2, ..,. k_{max}$). In equations (4.21) through (4.23), l and n are equal to the matrix order N.

The lower-layer weights in this case are set equal to A initially and are not updated. Similar to the LU-decomposition problem, the optimum value of η in equation (4.20) is obtained at each updating step via the hybrid line search method described in the chapter 3.

4.2.2 Training Procedure for Matrix Inversion FNN under SDLS Method

The training procedure comprises one forward computation of $z = Ax$ and $y = Bz$; and one computation of the equation (4.18), i.e., one backward training iteration. Accordingly, the SDLS training procedure using epoch (batch) training strategy is summarized in the following steps and its flow chart is shown in figure 4.5:

1. Start with $k = 0$ and set the initial $B^{(0)}$ to zero matrices.
2. Present an input-desired output pair P, and do steps (2.a)-(2.c) for all $P \in [1, N]$, in order to compute the total (epoch) sum of squared error, $E^{N(k)}$, and the total gradient, $g^{(k)}$, using equations (2.29) and (2.30), respectively, where in this case, $(y_{d,1}^p,........., y_{d,m}^p)^T = e_p$ is the pth desired output vector, and $(y_1^p,........., y_m^p)^T$ is the actual output vector for input e_p. Notice that the patterns are presented to the network in the following order: $(e_1, e_1), (e_2, e_2),...,(e_N, e_N)$. Where, N is equal to the matrix order n.
   a. Compute z and y.
   b. Calculate the sum of squared error, $E^{p(k)}$, given by equation (2.14).
   c. Calculate the gradient $\nabla E^{p(k)}$ for this pattern from equation (4.19).
3. Check the epoch based-error, $E^{N(k)} < \varepsilon_1$, or $\|g^{(k)}\| < \varepsilon_2$. If one of these conditions is met, go to step 8.

4. Otherwise, set the search direction $\underline{d}^{(k)} = - g^{(k)}$.
5. Perform line search along $\underline{d}^{(k)}$ using the hybrid methods mentioned in chapter 3 to determine the optimal value $\eta^{(k)}$ that minimizes $E^N(\eta^{(k)}, B^{(k)})$.
6. Update $B^{(k+1)}$ using equation (4.18), respectively.
7. Set $k = k+1$. If $k > k_{max}$, go to step 8, otherwise, repeat from step 2.
8. Stop.

4.2.3 Training Formulation for Matrix Inversion FNN under CG Methods

In case of adopting the conjugate gradient algorithm as described in section 3.3, the update equation (2.20) needs to be modified by replacing w^o_{sj} with b_{sj}. Specifically, to update the upper-layer neuron weights b_{sj}, the following equation is used:

$$b_{sj}^{(k+1)} = b_{sj}^{(k)} + \Delta b_{sj}^{(k)} \tag{4.24}$$

The weight correction $\Delta b_{sj}^{(k)}$ in equation (4.24) is set proportional to the search direction as follows:

$$\Delta b_{sj}^{(k)} = \eta \underline{d}^{(k)} \tag{4.25}$$

The search direction $\underline{d}^{(k)}$ in this equation is determined using either equation (4.12) or equation (4.13). The gradient of the mapping error function with respect to the weights b_{sj}, $g^{(k)}$, in equations (4.12) and (4.13) is given by equation (4.19). Similar to the case of the *LU*-decomposition problem, the value of $\beta^{(k+1)}$ in equation (4.13) is obtained using either equation (4.14) for *Fletcher-Reeves* variant of the algorithm or equation (4.15) for the *Polak-Ribiere* variant of the algorithm. Also, the optimum value of η in equation (4.25) is obtained at each updating step via the hybrid line search method described in the chapter 3.

It should be noted here that the lower-layer weights in this case are set equal to A initially and are not updated.

4.2.4 Training Procedure for Matrix Inversion FNN under CG Methods

The training procedure comprises one forward computation of $z = Ax$ and $y = Lz$; and one computation of the equation (4.18), i.e., one backward training iteration. Accordingly, the CG training procedure using epoch (batch) training strategy is summarized in the following steps and its flow chart is shown in figure 4.6:

1) Start with $k = 0$ and set the initial $B^{(0)}$ to zero matrices.
2) Present an input-desired output pair P, and do steps (2.a)-(2.c) for all $P \in [1, N]$, in order to compute the total (epoch) sum of squared error, $E^{N(k)}$, and the total gradient, $g^{(k)}$, using equations (2.29) and (2.30), respectively, where in this case, $(y^{p}_{d,1}, \dots\dots\dots, y^{p}_{d,m})^{T} = e_p$ is the pth desired output vector, and $(y^{p}_{1}, \dots\dots\dots, y^{p}_{m})^{T}$ is the actual output vector for input e_p. Notice that the patterns are presented to the network in the following order: (e_1, e_1), $(e_2, e_2), \dots\dots\dots, (e_N, e_N)$. Where N is equal to the matrix order n.
   a) Compute z and y.
   b) Calculate the sum of squared error, $E^{p(k)}$, given by equation (2.14).
   c) Calculate the gradient $\nabla E^{p(k)}$ for this pattern from equation (4.19).
3) Check the epoch based-error, $E^{N(k)} < \varepsilon_1$, or $\| g^{(k)} \| < \varepsilon_2$. If one of these conditions is met, go to step 13.
4) Otherwise, set the search direction $\underline{d}^{(k)} = - g^{(k)}$.
5) Perform line search along $\underline{d}^{(k)}$ using the hybrid method described in chapter 3 to determine the optimal value $\eta^{(k)}$ that minimizes $E^{N}(\eta^{(k)}, B^{(k)})$.
6) Update $B^{(k+1)}$ using equation (4.24), respectively.
7) Set $k = k+1$. If $k > k_{max}$, go to step 13.
8) Otherwise, present an input-desired output pair P, and do steps (8.a)-(8.c) for all $P \in [1, N]$, in order to compute the total (epoch) sum of squared error, $E^{N(k)}$, and the total gradient, $g^{(k)}$, using equations (2.29) and (2.30), respectively
   a) Compute z and y.
   b) Calculate the sum of squared error, $E^{p(k)}$, given by equation (2.14).
   c) Calculate the gradient $\nabla E^{p(k)}$ for this pattern from equation (4.19).

9) Check the epoch based-error, $E^{N(k)} < \varepsilon_1$, or $||g^{(k)}|| < \varepsilon_2$. If one of these conditions is met, go to step 13.
10) Otherwise, check *Powell*'s restart criteria given by equations (3.31) and (3.32). If one of these conditions is met, set $\underline{d}^{(k)} = -g^{(k)}$, go to step12.
11) Otherwise, set the search direction $d^{(k)} = -g^{(k)} + \beta^{(k)} d^{(k-1)}$, where $\beta^{(k)}$ is given by equation (4.14) in case of *Fletcher-Reeves* (CGFR) variant of the method and is given by equation (4.15) in case of *Polak-Ribiere* (CGPR) variant of the method.
12) Repeat from step 5.
13) Stop.

4.3 Potential for Parallelisation

An analysis of SDLS and CG training reveals that the batch (epoch) gradient calculation and the step-size calculation (line search) are by far the most computationally intensive components, amounting to 98 to 99% of the computation in the SDLS and CG algorithms. Thus, if these training procedure components are parallelised, a considerable time saving can be achieved.

4.3.1 Batch (Epoch) Gradient Calculation

This can be parallelised in terms of either the weights (n_w) or the training set (N). The former would involve implementing the FNN in parallel on an array of processors. This would give very little advantage, if any, because there is a large amount of communication between neurons in the forward and back propagation stages resulting in an excessive communication load.

The alternative therefore is to partition the algorithms in terms of the training set vectors (N), that is, divide up the calculation of the batch gradient over a number of processors, each process calculating partial batch gradients, which are combined to produce the overall batch gradient. This approach has a number of advantages:

a. The parallel algorithms are relatively easy to create, since the subsets of the partitioned training data can be considered as new training sets. Consequently,

existing sequential routines for the FNN and back propagation routines can be employed directly without change.

b. Each parallel process works with only a portion of the training set; hence, much larger problems can be accommodated than if the complete training set is required on each processor.

4.3.2 Step-Size Calculation

Here there are three choices for parallelisation, on the weights, on the training set or on the function evaluations. The comments made above on the first two possibilities, in relation to the batch gradient, also apply here.

Parallelisation in terms of the function evaluations cannot be done directly as the bracketing with golden section (hybrid) line-search method is essentially sequential. However, if a bisection approach is used, where a number of points are evaluated uniformly over a given interval and the minimum chosen, then each evaluation can be assigned to a parallel processor. To get the desired accuracy, the bisection can be done twice, the second time using the interval around the lowest points obtained in the first interval. Thus if T seconds is the duration of one evaluation over the training set then the bisection method when parallelised takes about $2T$ seconds. Increasing the number of parallel processes simply increases the accuracy with which the step-size is calculated. For 6 processors a total of 12 evaluations would be carried out which would take approximately $12T$ seconds if implemented sequentially. This method has a number of disadvantages:

a. It requires the complete training set to be available on each processor.

b. Increasing the number of processors does not improve speed-up.

c. If there are a limited number of processors then the accuracy of step-size determination will be restricted.

Consequently parallelisation of the training set is favored here also. In this instance the evaluation of the sum squared-error over all the training set is divided up over a number of parallel processors so that each one evaluates a sum squared-error for the portion of the training set assigned to it.

Adopting this approach the sequential hybrid line-search technique can be parallelised, without affecting the structure of the method. For NP processors the speed-up will be of the

order of n_W x T / NP while the sequential implementation takes approximately n_W x T seconds. For a hybrid line-search with NP =8 (a value which would give an accuracy greater than the 12 point bisection method) the speed-up will be greater than that achievable with the bisection technique whenever $NP > 4$. The advantages of partitioning the step-size calculation in this manner are:

a. It is compatible with the batch gradient calculation in that only a portion of the training set is stored on each processor, hence memory saving.
b. Speed up improves with the number of processors and accuracy is the same irrespective of the number of processors.
c. The algorithm is numerically equivalent to the sequential version.

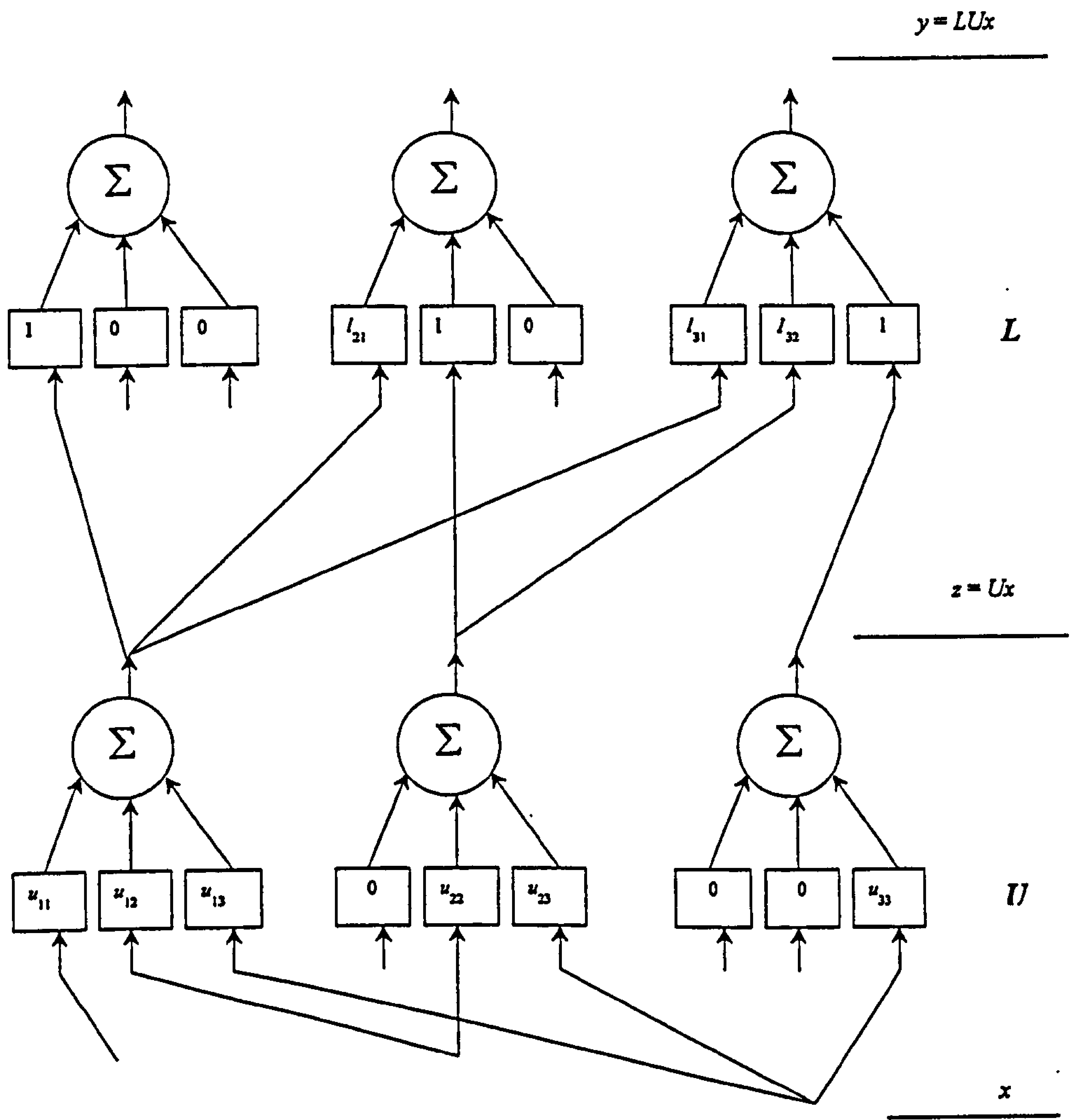


Figure 4.1 Structured network for *LU*- decomposition.

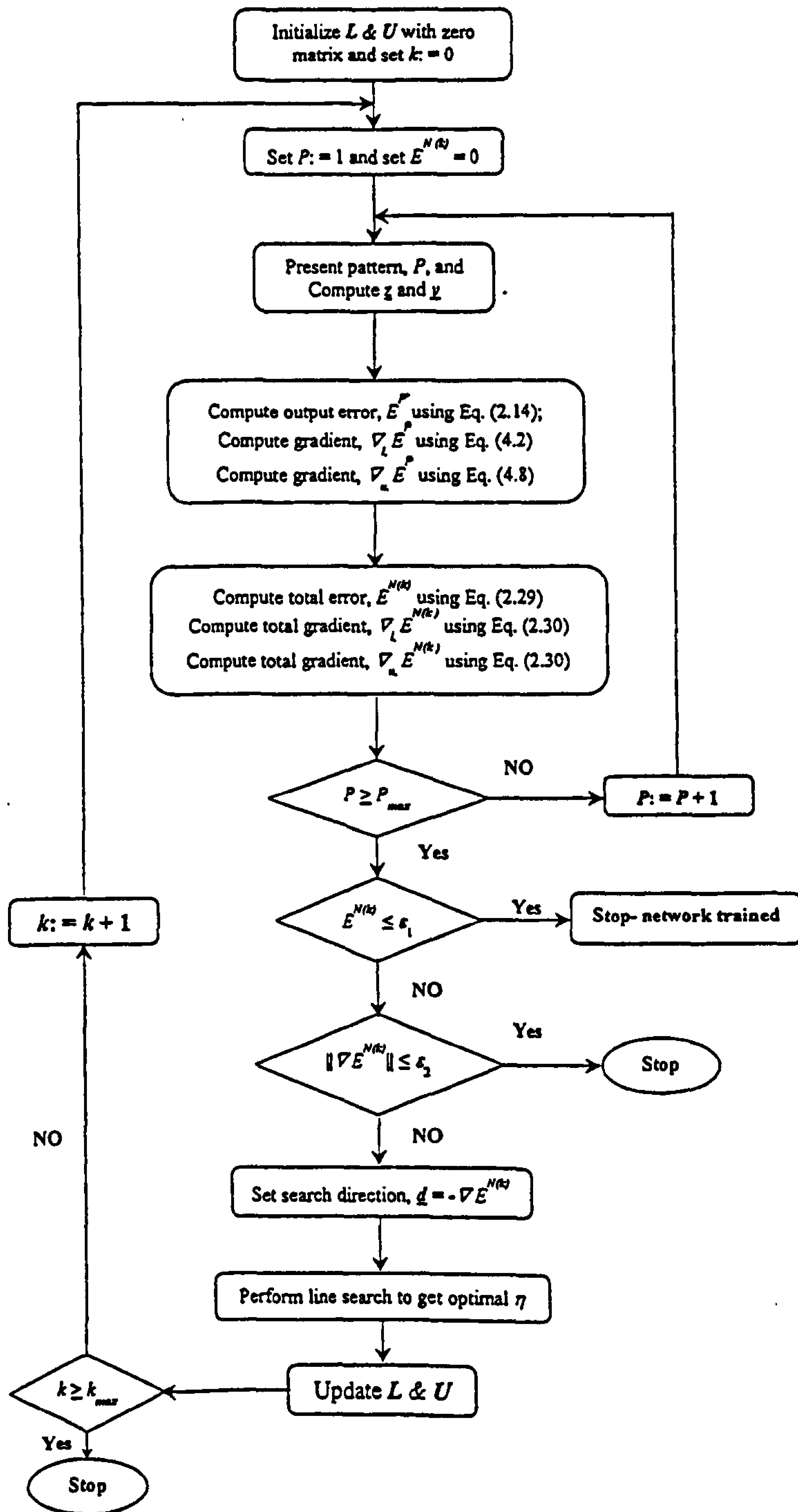


Figure 4.2 Flow chart of the SDLS method in case of *LU*-decomposition

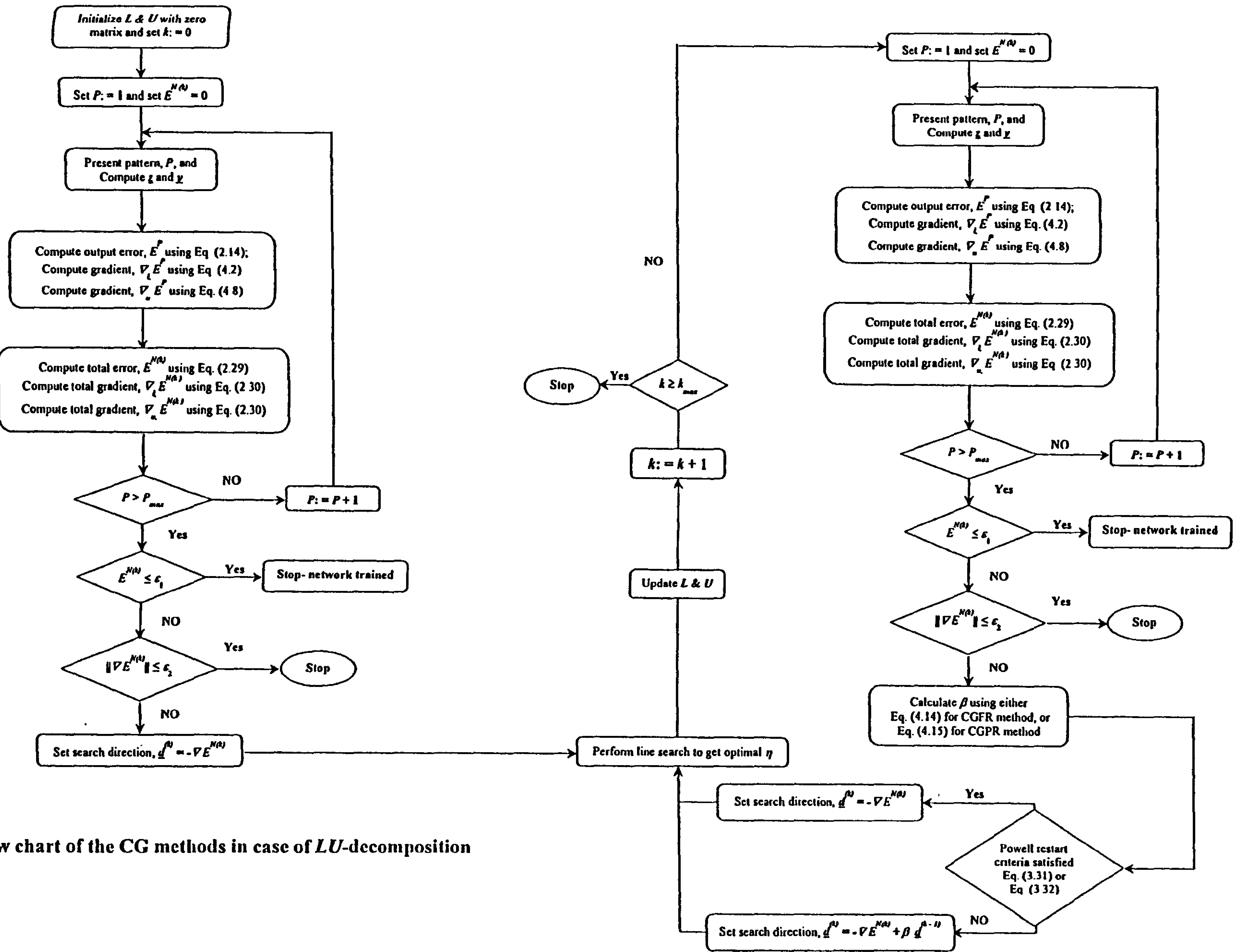


Figure 4.3 Flow chart of the CG methods in case of *LU*-decomposition

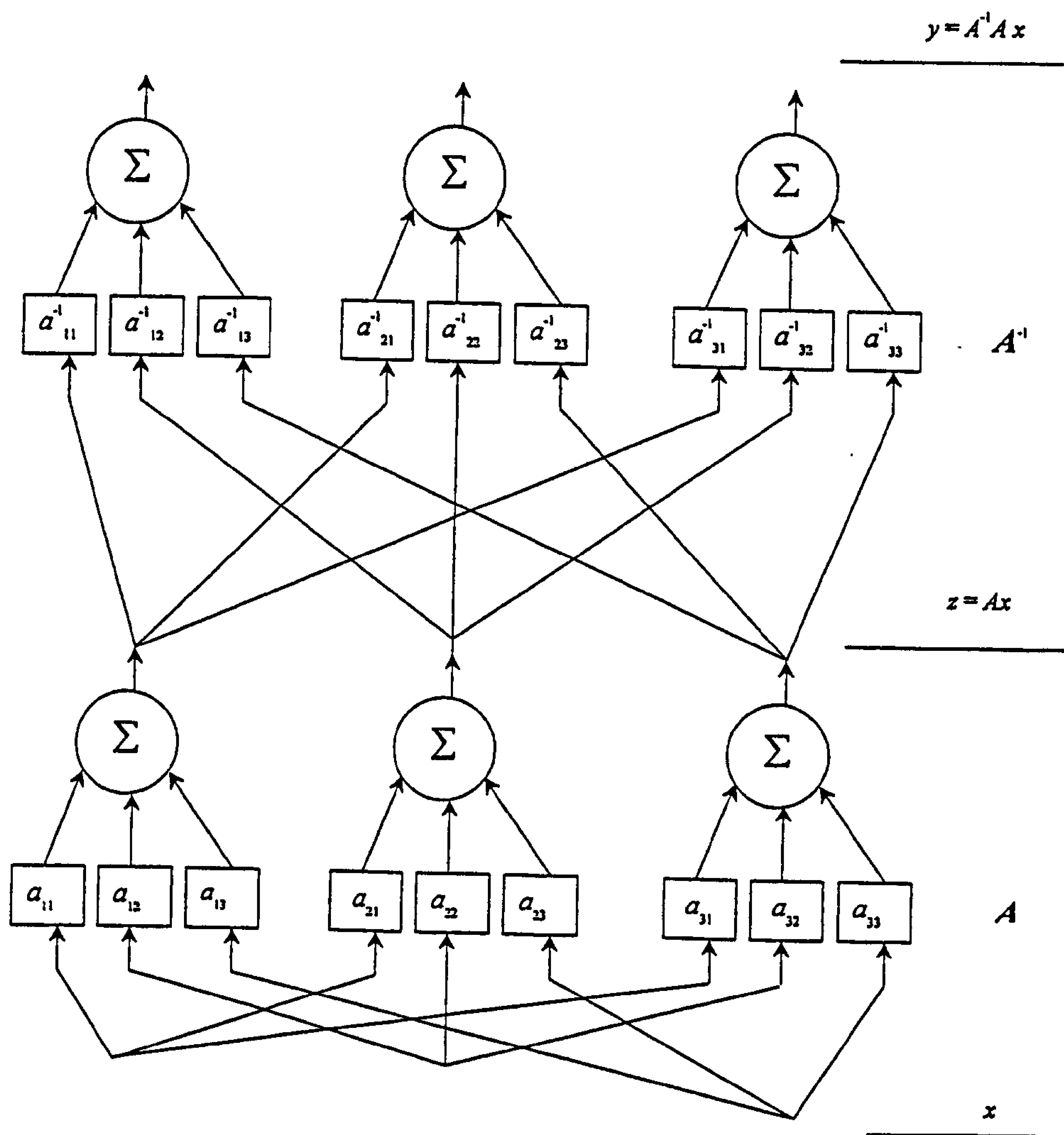


Figure 4.4 Structured network for Matrix Inversion.

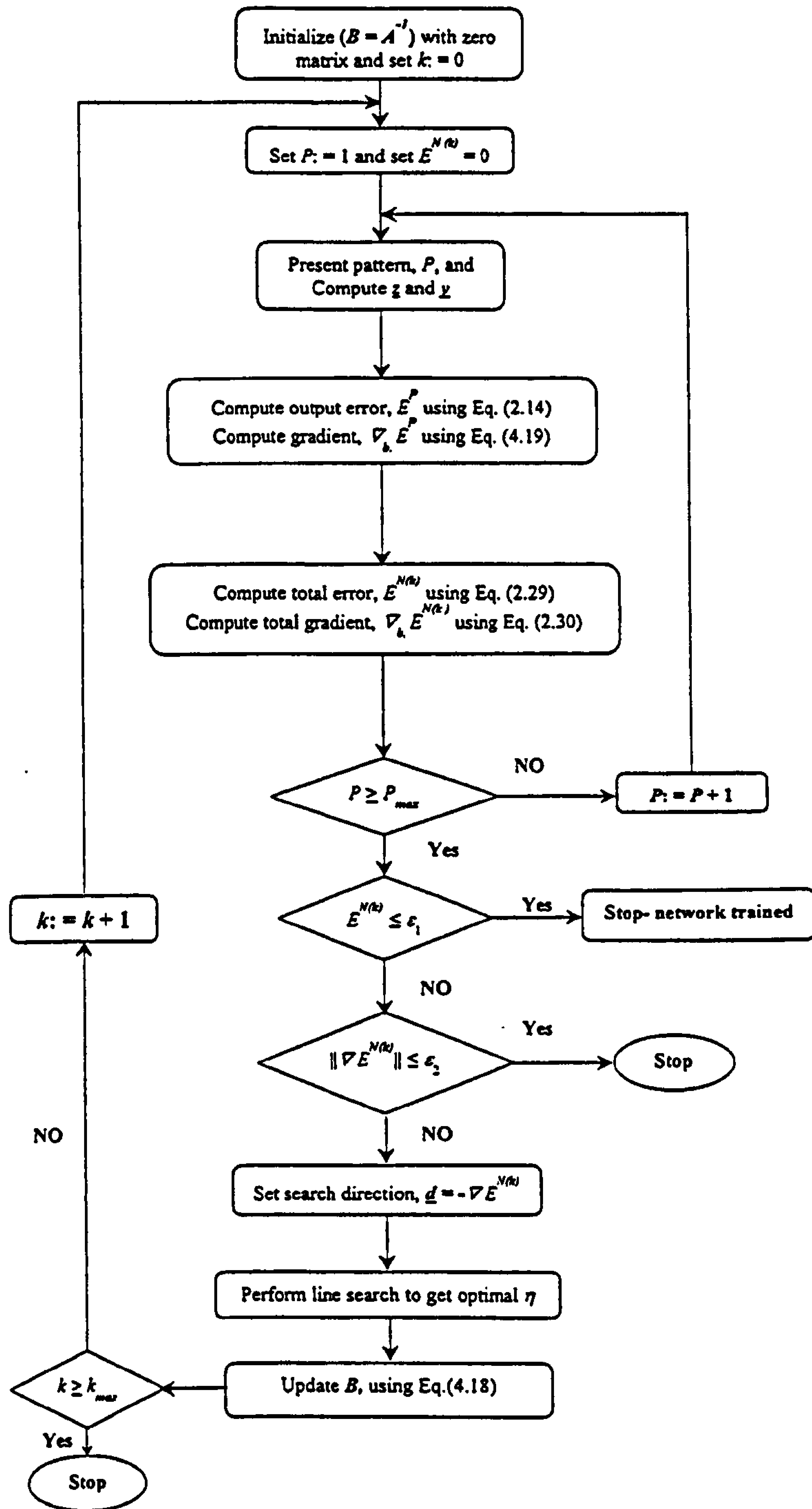


Figure 4.5 Flow chart of the SDLS method in case of matrix inversion

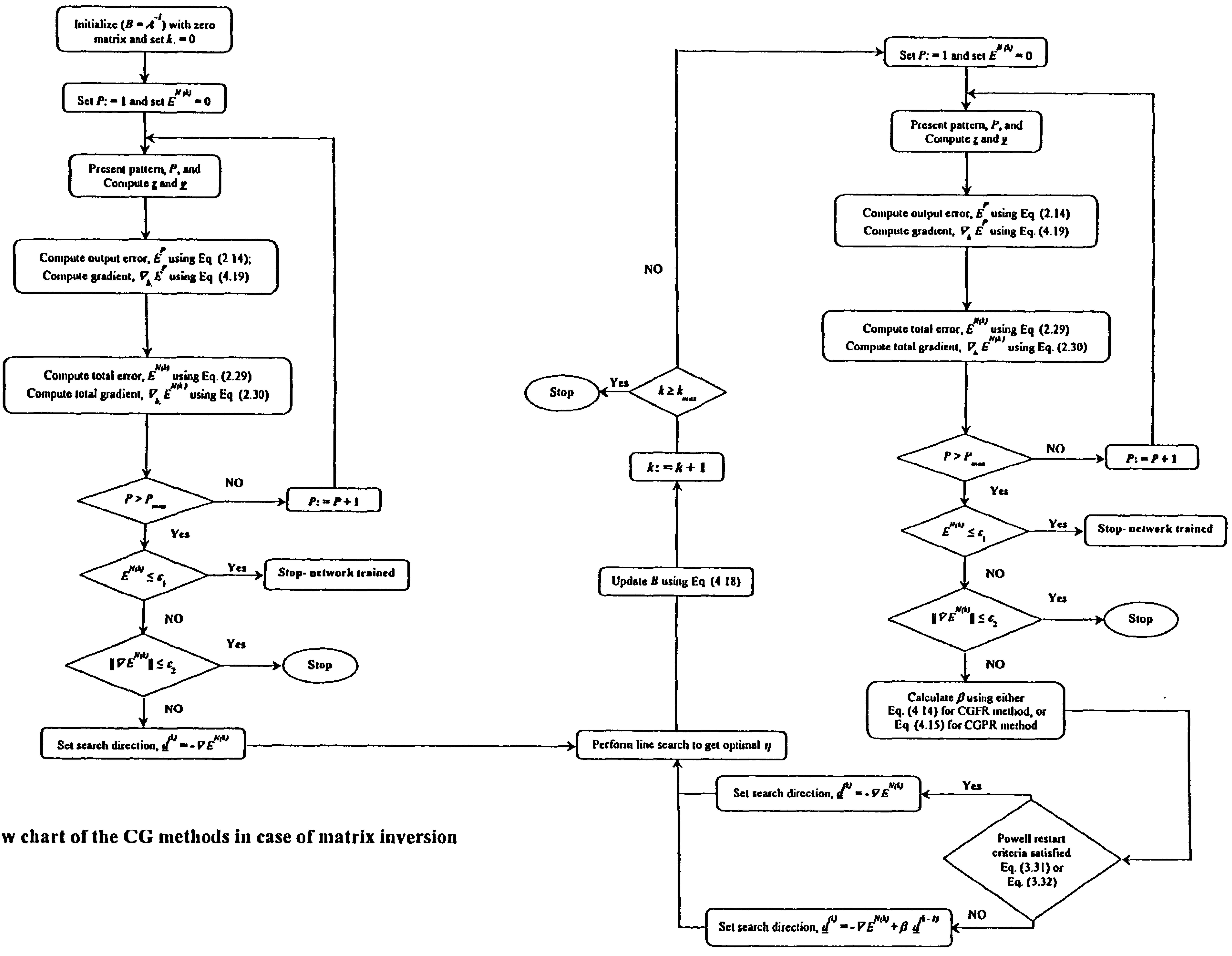


Figure 4.6 Flow chart of the CG methods in case of matrix inversion

CHAPTER 5

Simulation Results and Discussion

As mentioned in chapter 1, the development of a new approach for matrix computations depends highly on the training algorithms of FNN. Since the minimization method represents one of the key elements of the training algorithm, Section 5.1 is devoted to the assessment of those minimization methods described in chapter 3, especially the conjugate gradient methods. Some demonstrations showing the capability of the new approach as applied to various matrix computations are given in Sections 5.2 and 5.3. Specifically, Section 5.2 covers a series of computer simulations to assess the accuracy and feasibility of the FNN used for the *LU*-decomposition of square and band large unsymmetric matrices of various dimensions. Whereas, Section 5.3 presents an assessment of the FNN approach performance as applied to the inversion of large square unsymmetric matrices of different dimensions.

The condition numbers have been calculated for all tested matrices and it is found that their values are relatively large and vary between 8000 and 10000. This means that all tested matrices are not only large and unsymmetric but also ill-conditioned.

In all test cases of the above matrix computation problems, the step sizes of the conventional BP method are calculated using the following variant of the equation introduced by Wang and Mendel [23],

$$\eta = 0.00005 \times 0.9^{x_m} \qquad ; x_m = 0, 1, 2, 3 \ldots\ldots \qquad (5.1)$$

In this equation, the value of the coefficient is different from that recommended by Wang and Mendel. In fact, the value of 0.00005 is carefully chosen to achieve the fastest convergence. The value of x_m in the above equation varies according to the following rule: If the value of the total squared error $E^N(\underline{w})$ keeps decreasing as training proceeds, the step size is kept constant. Whereas, if the total squared error $E^N(\underline{w})$ does not reach its threshold value and, at the same time, starts to increase, then x_m takes a larger value and the step size is updated using this equation.

The learning processes terminate when one of the following conditions is met: First, when the total number of iterations exceeded a specified iteration limit (k_{max}); Second, when the total squared error $E^N(\underline{w})$ is less than a small threshold value (ε_1); Third, when the norm of the batch gradient error $\|g\|$ is less than a small threshold value (ε_2). To make sense, the convergence performance versus time is compared. All programs for the simulations were written in standard Fortran language and performed on an IBM PC/Pentium 400 MHz microcomputer.

5.1 An Assessment of Conjugate Gradient Minimization Methods

Figure 5.1 shows the learning curves obtained when using the conjugate gradient (CG) methods with various reset values (r) as applied to one of the test problems considered in the present study. This problem is the *LU*-decomposition of a band matrix having a dimension of 19 x 195; i.e., where the upper- and lower-band width is equal to 9. Conjugate gradient with r = 1 in this figure is simply SDLS (steepest descent with line search) while conjugate gradient with r = ∞ corresponds to pure conjugate gradient (i.e., no reset). As can be seen, conjugate gradient with reset performs in general much better than pure conjugate gradient and conjugate gradient with some reset values is significantly better than SDLS. For all reset values greater than 3, the learning curves of this problem reach plateau which sometimes cannot be lowered below 10^{-2} (r = 4 and r = 7). From this figure, it is also clear that the performance varies considerably depending on the choice of reset (r). For instance, the performance improves as r increases from 2 to 3 and it deteriorates when r takes a value of 4. Then, the reset values (r) of 5 and 6 tend to give better results than that of r = 4. Finally, the performance deteriorates when r takes a value of 7.

Accordingly, there is no clear choice for the optimum reset value. However, for the problems under consideration, this value is evidently 3; but it cannot be generalized for other problems since it could vary widely from one problem to another. Thus, rather than keeping the reset value fixed throughout training, it could be allowed to vary as training proceeds. This can be done manually; that is, starting with a large value and reduce it as training proceeds or visa versa, or it can be done automatically based on some rules. Possible rules for determining (r) could be a function of some or all of the following: the total squared error, the number of iterations, the number of weights, and the magnitude of the gradient. Some

variable resetting criteria, based on the last of these, have been developed by *Powell* [68] and are given in chapter 3 by equations (3.31) and (3.32).

After applying the *Powell*'s restarting criteria to the above problem, it is found that the learning curve, which is denoted by (r = computed) in figure (5.1), is much better than those obtainable using a fixed reset parameter (r = 4, 5, 6). Moreover, this approach of resetting gives much greater consistency in performance as compared to the use of fixed reset parameters. Surprisingly, when checking the values of resetting for this case as training proceeds, it is found that they vary between 2 and 3. Actually, this is consistent with the above obtained optimum reset value. In other words, *Powell*'s restarting criteria given by equations (3.31) and (3.32) result in reset values very close to the optimal value. This is because the effect of resetting can be thought of as building up a quadratic model of the objective function over a number of iterations. In the earlier stages of training the algorithm may be operating in a region of the error space that is highly non-quadratic and, consequently, the validity of the model being built is very localized. Frequent resetting under these conditions is advantageous because it discards information that is only locally applicable, information that would distort the model in the later stages of training. As training moves the weights nearer to a minimum the quadratic model will become more widely valid and, consequently, resetting should be employed less frequently or not at all. A second interpretation of the effect of resetting is that it limits the accuracy of the quadratic model and, hence, it limits the search directions produced by the algorithm. Consequently, the algorithm is less likely to be attracted towards a shallow minimum, which is of course advantageous. However, once in the basin of attraction of a minimum, resetting has a detrimental effect and should be phased out.

Despite of these advantages of automatic resetting, it is clear from figure (5.1) that the number of iterations required to reduce the total error to a value of 10^{-6} is twice as much as those required for the cases of fixed reset parameter (r = 2) and (r = 3). This indicates that the computational cost required in case of using *Powell*'s resetting criteria is much higher than those for the cases of r = 2 or 3. Consequently, in the present study, in order to achieve the fastest convergence when using conjugate gradient methods, a fixed reset value of 3 is taken for all similar problems considered herein since this value produces the best results.

In figure 5.2, a set of the optimal step sizes, η *(k)*, gathered from different learning procedures applied to this problem is shown. As can be seen, the optimal step size (LR), η, in case of conjugate gradient methods (Fig. 5.2a and Fig.5.2b) varies almost randomly from iteration to iteration with relatively large dynamic range. This implies, in a sense, the difficulty in updating LR based on their previous values and also gives a sound support in the necessity of using dynamic LR. Comparing these optimal step sizes (η) with those obtained when using the SDLS algorithm (Fig. 5.2c), one can find that they cover a much wider range than those of the SDLS algorithm and are on average 10 times larger. This is an indication that conjugate gradient is a much better method, that is, large step values occur because the search directions are much more accurate. The oscillatory pattern of the SDLS step sizes can be interpreted as an indication to the inefficiency of this method because it contains information not utilized by the algorithm. In case of the conjugate gradient methods, it is clear that there is no any particular pattern in the η values, which are almost random.

5.2 The *LU*-Decomposition Problem

The first group of test cases considered in this study is the matrix *LU*-decomposition. In this group, the present approach is examined on two classes of *LU*-decomposition problem. The first class is the *LU*-decomposition of large unsymmetric ill-conditioned square matrices, whereas, the second class is the *LU*-decomposition of large unsymmetric ill-conditioned band matrices. In both classes, the three training methods presented in chapter 4; namely, SDLS, CGFR, and CGPR methods are applied to several test cases to compare their performances as to the successful convergence within an iteration limit or when the termination condition of learning is met. For comparison, the corresponding results for the conventional BP method are also included. In this method, the step size is calculated through equation (5.1) to achieve the fastest convergence.

Since the BP method is sensitive to different initial weights, trials starting from several initial weights, $\underline{w}_o$, were carried out. It is found that only when the initial weights take the values of zeros, the training is successful. Therefore, for comparison purposes, all methods will start from the same initial weight vectors ($\underline{w}_o = 0$) and will receive the same sequence of training patterns.

For all test cases, the threshold value (ε_1) of the total squared error is kept the same at a value of 10^{-9}, whereas, the threshold value (ε_2) of the batch gradient error is kept the same at a value of 10^{-8}. The iteration limit is taken as 10000 iterations for the SDLS and BP methods and as 81 cycles for the conjugate gradient methods.

5.2.1 *LU*-Decomposition of Large Square Matrices

For all cases considered in this class of problem, the network structure is similar to the one given in figure 4.1. In this network, the total number of weights to be updated in the network each training cycle is equal to the matrix dimension *(n* x *n)*. Table 5.1 gives a summary of the test cases considered for this class of problem.

Table 5.1

A Definition of training tests of *LU*-decomposition of square matrices

Name	Matrix dimension	Training set size
Case 1	100 x 100	100 vectors
Case 2	130 x 130	130 vectors
Case 3	200 x 200	200 vectors
Case 4	260 x 260	260 vectors

An inexact line search; namely, bracketing with golden section method, is used with the SDLS method. Various degrees of inexactness were tested and the best learning curves obtained in each case are given in figure 5.3. In this figure, each graph expresses the total squared error, $E^N(\underline{w})$, as a function of the number of gradient evaluations (iterational number) for CGFR, CGPR, and SDLS training methods. Whereas, in case of the BP method, the total squared error is expressed as a function of the number of objective function evaluations (iterational number). Except for BP method, all runs converged successfully to a total squared error value, which is below 10^{-7}. All runs for BP method reach plateaus that vary between 10^{-5} and 10^{-6}. As can be seen, the learning curves for the CGFR method are very similar to those of the CGPR method in convergence speed and accuracy.

The terminated number of iterations, averaged running time, convergence accuracy, number of objective function evaluations, and number of gradient evaluations for all the simulated methods are summarized in Tables (5.2) – (5.5) for Cases (1) – (4), respectively.

Table 5.2
Detailed simulation results for Case 1 under four training methods

Method	CGFR (r = 3)	CGPR (r = 3)	SDLS	Conventional BP
Running time (s)	45.19	57.51	203.93	210.58
Terminated iteration no.	51	66	193	1208
Terminated E	8.4×10^{-9}	1.18×10^{-8}	9.5×10^{-9}	1.38×10^{-6}
Function evaluations	674	880	2255	1208
Gradient evaluations	50	66	193	--

Table 5.3
Detailed simulation results for Case 2 under four training methods

Method	CGFR (r = 3)	CGPR (r = 3)	SDLS	Conventional BP
Running time (s)	137.70	155.4	337.39	481.90
Terminated iteration no.	73	81	154	1233
Terminated E	4.62×10^{-9}	1.79×10^{-8}	5.36×10^{-9}	1.53×10^{-6}
Function evaluations	963	1014	1843	1233
Gradient evaluations	73	81	154	--

Table 5.4
Detailed simulation results for Case 3 under four training methods

Method	CGFR (r = 3)	CGPR (r = 3)	SDLS	Conventional BP
Running time (s)	573.58	557.82	1977.81	1737.48
Terminated iteration no.	81	81	250	1078
Terminated E	5.97×10^{-8}	5.61×10^{-8}	5.93×10^{-8}	1.88×10^{-6}
Function evaluations	1066	1052	2826	1078
Gradient evaluations	81	81	250	--

Table 5.5
Detailed simulation results for Case 4 under four training methods

Method	CGFR (r = 3)	CGPR (r = 3)	SDLS	Conventional BP
Running time (s)	1325.68	1338.25	4798.07	3526.65
Terminated iteration no.	81	81	250	1002
Terminated E	2.82×10^{-8}	2.98×10^{-8}	9.26×10^{-8}	3.1×10^{-6}
Function evaluations	1046	1020	2803	1002
Gradient evaluations	81	81	250	--

As can be observed from the above tables, the CG methods require significantly less function evaluations for small size network despite their dependency on accurate line searches. Also, the iteration times of both CG training methods are comparable and much less than those of other methods, even though the method for determining the search direction is much more computationally intensive. In fact, the CG methods require less iterations because it can provide better estimates of the step size (learning rate). Therefore, it can be concluded that the present CG methods have remarkable advantages in accelerated convergence, accuracy, and time savings.

Thus far the performance of the various training methods considered has been assessed in terms of the rate of convergence to a minimum as a function of the number of iterations. This is useful when determining the power of an algorithm, but it does not take into account the algorithm iteration time and hence is not a true reflection of performance.

When comparing training methods the essential criterion is clearly the error reduction obtainable in a given amount of time. This can easily be assessed by plotting the learning curves as a function of time. Those obtained for the problem of *LU*-decomposition of square matrices are given in figure 5.4 and clearly show that the CGFR method is the best for test Cases 1 and 2, followed by CGPR method. Also, as can be seen, each of these two methods stopped training after only couple of seconds, having reached minima in the error surface. From figures (5.4a) and (5.4b), it is clear that each of these methods ended up in a different local minimum with the CGFR method finding a much deeper minimum than other methods.

Furthermore, for relatively large problems (Cases 3 and 4), figures (5.4c) and (5.4d) indicate that both CG training methods are very much superior to other methods. CGFR and CGPR methods perform reasonably well when compared to the SDLS method. Nevertheless, the CGFR has a better rate of convergence for large problems as compared to the CGPR method. This observation is true in general and appears to be related to the ability of the CGFR method to find its way out of plateau type regions on the error surface. The BP method also gives very good results for this problem though, in general, it tends to be inferior to the CG methods.

The size of the network, the number of training vectors and the degree of accuracy with which the line searches are carried out all influence the iteration time. The effect of these parameters is investigated and the results are plotted in figure 5.5. From this figure, it can be stated that the iteration time for all methods increases as the size of the network (matrix dimension) increases. Also, the rate of convergence decreases for all methods as the size of the problem increases.

The effect of the size of the network and the number of training vectors on the optimal step size is also clear from figures (5.6) through (5.9). For clarity purposes, each curve is plotted in a separate graph. On average the CG methods optimal step sizes are much larger than those of the SDLS method. Furthermore, the size of the network has no effect on the behavior of the optimal step size for the CG methods. This indicates the superiority of these methods over the SDLS method. Moreover, by comparing figures (5.6) – (5.9), it is clear that the SDLS optimal step size behavior deteriorates as the size of the network increases.

5.2.2 *LU*-Decomposition of Large Band Matrices

The efficiency of a given matrix algorithm depends on many things. Most obvious is the amount of arithmetic and storage required by a procedure. As an example of exploitable structure, the property of bandedness is chosen. Band matrices have many zero entries and so it is no surprise that band matrix manipulation allows for many arithmetic and storage shortcuts. Formally, it is said [1] that $A = (a_{ij})$ has *upper bandwidth* (superdiagonals) m_1 if $a_{ij} = 0$ whenever $j > i + m_1$ and lower bandwidth (subdiagonals) m_2 if $a_{ij} = 0$ whenever $i > j + m_2$. Substantial saving can be realized when solving banded linear system of equations since

the triangular factors in *LU*-decomposition are also banded. In practice, a band linear equation solver would be organized around a data structure that takes advantage of the many zeros in A. Therefore, if A has a *lower bandwidth* m_2 and an *upper bandwidth* m_1, it can be represented in a $(m_1 + m_2+1)$-by-n array $A.band$ where band entry a_{ij} is stored in $A.band\ (i - j + m_1 + 1, j)$. In this arrangement, the nonzero portion of A's jth column is housed in the jth column of $A.band$.

In this section, the results of several test cases used to examine the performance of the present approach on the *LU-decomposition* of band matrix are presented. Table 5.6 lists a definition of these test cases as well as both matrix and training set size used. As can be seen from this table, these cases are grouped into two categories. Using the first category, the effect of changing the matrix size *(n)* on the performance of training methods is investigated. Hence, the bandwidth in all cases of this category is kept the same at a value of 19 (Cases 5 - 8). Whereas, the second category is used to investigate how changing both the bandwidth $(m_1+ m_2+1)$ and the dimension *(n)* affect the performance of the training methods (Cases 9-11).

Table 5.6
A Definition of training tests of *LU*-decomposition of band matrices

Name	Matrix dimension	Training set size
	<u>Same bandwidth Cases</u>	
Case 5	19 x 100	100 vectors
Case 6	19 x 130	130 vectors
Case 7	19 x 160	160 vectors
Case 8	19 x 195	195 vectors
	<u>Different bandwidth Cases</u>	
Case 9	104 x 200	200 vectors
Case 10	134 x 260	260 vectors
Case 11	164 x 320	320 vectors

For all of the above cases, the network structure is similar to the one given in figure 4.1. In this network, the total number of weights to be updated in the network each training cycle is equal to the band matrix dimension $(m_1 + m_2 + 1)$-by-n. Accordingly, the number of weights to be updated in the lower layer (U) is equal $(m_1 + 1) \times n$, whereas, the number of weights to be updated in the upper layer (L) is equal $m_2 \times n$.

Similar to the *LU*-decomposition of square matrices, an inexact line search; namely, bracketing with golden section method is used with the SDLS method. Various degrees of inexactness were tested and the best learning curves obtained for the two categories of band matrix are given in figures (5.10) and (5.11). In the following two sections, these results are discussed.

5.2.2.1 *LU*-Decomposition for Band Matrices of Same Bandwidth

In this section, the matrix bandwidth in all test cases is kept constant at 19 and the effect of changing the matrix dimension *(n)* on the performance of the training methods is investigated (Cases 5-8). The results of these cases are plotted in figure (5.10). In this figure, each graph expresses the total squared error, $E^N(\underline{w})$, as a function of running time. From figure (5.10), it is clear that, except for the SDLS method, all methods have almost the same convergence rate for the test Cases (5-8). Performance comparison of all training methods reveals that the CGFR method gives the best convergence speed and accuracy followed by the CGPR and SDLS, respectively. As can also be seen, all runs for these three methods converge successfully to a total squared error value of 10^{-8}. Therefore, it can be stated that the learning curves for the SDLS method are very similar in accuracy to those of the CGFR and CGPR methods, but are different in convergence rates and this difference increases as the size of both matrix and training set increases.

It is also interesting to note from figure (5.10) that all runs for BP method reach, in general, the same plateau value of 10^{-6}, regardless of the size of the matrix or training set used. This may be in consequence of the use of equation (5.1) for the calculation of the step size and, hence; it suggests the needs for a better learning rate adaptation technique such as the line search method used in the SDLS method. Therefore, it can be stated that BP method does not have the capability of reaching global minimum when using fixed learning rates.

Comparison between the learning curves, shown in figure (5.10), for SDLS and BP methods indicates that the BP method has a better convergence rate than the SDLS method, which means that SDLS method takes longer actual computational time. However, the BP method achieves much worse accuracy than SDLS method. The difference between convergence rates of these two methods becomes more pronounced as the size of both matrix and training set increases. On the other hand, comparison between the learning curves, shown in figure (5.10), for the CGFR and CGPR methods reveals that these two methods have the same convergence rate, but they differ in accuracy for small size problems. This difference in accuracy becomes less pronounced as the size of the problem increases. Accordingly, it can be stated that, for same bandwidth, as the size of the problem increases the performances of both CGPR and CGFR methods become identical.

It is interesting to compare the performance of each method separately as the size of the problem increases keeping the bandwidth fixed. This is shown in figure 5.12. In this figure, each graph expresses the total squared error, $E^N(\underline{w})$, as a function of the number of gradient evaluations for CGFR, CGPR, and SDLS training methods. Whereas, in case of the BP method, the total squared error is expressed as a function of the number of objective function evaluations. From these graphs, it can be stated that for all methods, the performance does not change as the size of the problem changes. In other words, in each method, the number of iterations required to reach a specified convergence accuracy is the same regardless of the size of the problem. Of course, the time required for reaching this convergence accuracy increases as the size of the problem increases since the time per training cycle increases as the number of updated weights increases. This can also be seen from Tables (5.7 – 5.10) for Cases (5 – 8), respectively. Listed in these tables are the terminated number of iterations, averaged running time, convergence accuracy, number of objective function evaluations, and number of gradient evaluations for all the simulated methods.

Table 5.7
Detailed simulation results for Case 5 under four training methods

Method	CGFR (r = 3)	CGPR (r = 3)	SDLS	Conventional BP
Running time (s)	5.64	6.37	22.51	15.08
Terminated iteration no	67	74	241	1208
Terminated E	4.26×10^{-9}	1.63×10^{-7}	9.69×10^{-9}	1.38×10^{-6}
Function evaluations	878	951	2789	1208
Gradient evaluations	67	74	241	--

Table 5.8
Detailed simulation results for Case 6 under four training methods

Method	CGFR (r = 3)	CGPR (r = 3)	SDLS	Conventional BP
Running time (s)	9.88	11.42	32.03	26.81
Terminated iteration no	67	81	216	1232
Terminated E	7.77×10^{-9}	1.79×10^{-8}	9.69×10^{-9}	1.53×10^{-6}
Function evaluations	891	1014	2493	1232
Gradient evaluations	67	81	216	--

Table 5.9
Detailed simulation results for Case 7 under four training methods

Method	CGFR (r = 3)	CGPR (r = 3)	SDLS	Conventional BP
Running time (s)	16.58	17.12	51.47	41.70
Terminated iteration no	76	81	222	1252
Terminated E	9.93×10^{-9}	4.13×10^{-8}	9.99×10^{-9}	1.67×10^{-6}
Function evaluations	985	1014	2657	1252
Gradient evaluations	76	81	222	--

Table 5.10
Detailed simulation results for Case 8 under four training methods

Method	CGFR (r = 3)	CGPR (r = 3)	SDLS	Conventional BP
Running time (s)	25.54	27.62	91.06	63.07
Terminated iteration no	81	81	250	1254
Terminated E	5.59×10^{-8}	6.22×10^{-8}	5.84×10^{-8}	1.88×10^{-6}
Function evaluations	1032	1025	2845	1254
Gradient evaluations	81	81	250	--

A consideration that is worth mentioning is the difference between epochs (weight vector updates) and objective function evaluations: for the BP method, the number of function evaluations equals the number of epochs; for the SDLS, CGFR and CGPR methods, there is a number of additional objective function evaluations due to the line search problem. Thus, a comparison in terms of function evaluations is preferable in order to readily obtain the computational efficiency of these three methods. For instance, BP method in Case 5 requires 1208 function evaluations to reach the global minimum, while SDLS, CGFR, and CGPR require 2789, 878, and 951 function evaluations, respectively, under the same conditions. In general, as can be seen from the tables, the number of function evaluations in all cases is almost the same for the CG methods and equals almost one third of that required for the SDLS methods. This explains why the running times for all CG methods are almost the same in all cases and, also, explains why the running times for all CG methods are equal to one third of that required for the SDLS method under the same conditions. Accordingly, it can be stated that the CG methods require significantly less function evaluations for all network sizes despite their dependency on accurate line searches. Also, the iteration times of both CG training methods are comparable and are one third of the SDLS method, even though the method for determining the search direction is much more computationally intensive. In fact, the CG methods require a lesser amount of iterations because they can provide better estimates of the step size (learning rate). Therefore, it can be concluded that

the present CG methods have remarkable advantages in accelerated convergence, accuracy, and time saving when applied to *LU*-decomposition of band matrices.

The effect of the size of the network and the number of training vectors on the optimal step size is clear from figures (5.13) – (5.16). For clarity purposes, each curve is plotted in a separate graph. Similar to the Cases (1-4), the CG methods optimal step sizes on average are much larger than those of the SDLS method. Also, the size of the network has no effect on the behavior of optimal step size for the CG methods. Of course, this indicates once again the superiority of the CG methods over the SDLS method. Unlike the Cases (1-4), when comparing figures (5.13) – (5.16), it is clear that the SDLS optimal step size behavior improves as the size of the network increases.

5.2.2.2 *LU*-Decomposition for Band Matrices of Different Bandwidth

In this section, the effect of changing both the bandwidth (m_1+m_2+1) and the dimension (n) on the performance of the training methods is investigated (Cases 8-11). The results of these cases are plotted in figure (5.11). In this figure, each graph expresses the total squared error, $E^N(\underline{w})$, as a function of running time. It is clear from this figure that, the CGFR method gives the best convergence rate and accuracy followed by the CGPR and SDLS, respectively. As can be seen, all runs for these three methods converge successfully to a total squared error value in the range between 10^{-7} and 10^{-8}. Also, from this figure, it can be stated that the learning curves for the SDLS method are very similar in accuracy to those of the CGFR and CGPR methods, but are different in convergence rates and this difference stays the same as the size of both matrix and training set increases.

Similar to Cases (5-8), all runs for BP method, in general, reach the same plateau value of 10^{-6}, regardless of the size of the matrix or training set used. This implies that BP method does not have the capability of reaching a global minimum for this class of problems. Comparison between the learning curves of both SDLS and BP methods indicates that the BP method has a better convergence rate than the SDLS method, which means that SDLS method takes longer actual computational time. However, the BP method achieves much worse accuracy than SDLS method. It is clear from figure (5.11), that the CGFR and CGPR methods have the same convergence rate and accuracy for all cases. Accordingly, it can be

stated that, for different bandwidths, the performances of both CGPR and CGFR methods are identical regardless of the problem size.

Using another performance index, the effect of changing both bandwidth (m_1+m_2+1) and dimension *(n)* on the performance of each method is investigated. Here, rather than expressing the total squared error, $E(\underline{w})$, as a function of time, it is expressed as a function of the number of gradient evaluations for CGFR, CGPR, and SDLS training methods and as a function of the number of objective function evaluations in case of the BP method. This is shown in figure 5.17. From this figure, it can be stated that for all methods, the performance does not change as the size of the problem changes. In another word, in each method, the number of iteration required to reach a specified convergence accuracy is the same regardless of the size of the problem. Of course, the time required for reaching this convergence accuracy increases as the size of the problem increases since the time per training cycle increases as the number of updated weights increases. This can also be seen from Tables (5.10 – 5.13) for Cases (8 – 11), respectively. Listed in these tables are the terminated number of iterations, averaged running time, convergence accuracy, number of objective function evaluations, and number of gradient evaluations for all the simulated methods.

From tables (5.10 – 5.13), it is clear that the CG methods require significantly less function evaluations for Cases (9-11) despite their dependency on accurate line searches. Same conclusion has been reached earlier for Cases (5-8). It is interesting to note that the iteration times of both CG training methods are comparable and are one third of the SDLS method, even though the method for determining the search direction is much more computationally intensive. As stated earlier, since CG methods can provide better estimates of the step size (learning rate), they require a lesser amount of iterations. Therefore, it can be concluded once again that the present CG methods have remarkable advantages in accelerated convergence, accuracy, and time saving when applied to *LU*-decomposition of band matrices.

Table 5.11
Detailed simulation results for Case 9 under four training methods

Method	CGFR (r = 3)	CGPR (r = 3)	SDLS	Conventional BP
Running time (s)	505.21	503.17	1772.56	1431.51
Terminated iteration no	81	81	250	1251
Terminated E	1.86×10^{-8}	4.13×10^{-7}	5.54×10^{-8}	2.79×10^{-6}
Function evaluations	1054	1057	2830	1251
Gradient evaluations	81	81	250	--

Table 5.12
Detailed simulation results for Case 10 under four training methods

Method	CGFR (r = 3)	CGPR (r = 3)	SDLS	Conventional BP
Running time (s)	1119.32	1086.90	4150.93	3484.70
Terminated iteration no	81	81	250	1286
Terminated E	2.82×10^{-8}	3.27×10^{-8}	4.75×10^{-8}	3.10×10^{-6}
Function evaluations	1054	1081	2816	1286
Gradient evaluations	81	81	250	--

Table 5.13
Detailed simulation results for Case 11 under four training methods

Method	CGFR (r = 3)	CGPR (r = 3)	SDLS	Conventional BP
Running time (s)	2088.60	2007.26	7412.18	6839.71
Terminated iteration no	81	81	250	1309
Terminated E	5.22×10^{-8}	4.80×10^{-8}	5.49×10^{-8}	3.4×10^{-6}
Function evaluations	1054	1030	2811	1309
Gradient evaluations	81	81	250	--

The effect of the size of the network and the number of training vectors on the optimal step size is clear from figures (5.18) – (5.20). Once again, for clarity purposes, each curve is plotted in a separate graph. Similar to Cases (5-8), the CG methods optimal step sizes, on average, are much larger than those of the SDLS method. Furthermore, the size of the network has no effect on the behavior of CG methods optimal step size, which indicates the superiority of the CG methods over the SDLS method. From figures (5.18) – (5.20), it is clear that the SDLS optimal step size behavior improves as the size of the network increases.

5.3 The Matrix Inversion Problem

To further demonstrate the capability of the present approach, it is examined on another matrix computation problem. In this section, the results of this problem; namely, the inversion of six large unsymmetric ill-conditioned square matrices, are presented. Table 5.14 lists a definition of the test cases considered herein as well as both matrix and training set size used.

Table 5.14
A Definition of training tests for the matrix inversion problem

Name	Matrix dimension	Training set size
Case 12	100 x 100	100 vectors
Case 13	130 x 130	130 vectors
Case 14	160 x 160	160 vectors
Case 15	200 x 200	200 vectors
Case 16	260 x 260	260 vectors
Case 17	320 x 320	320 vectors

For all of the above cases, the network structure is the same as the one given in figure 4.4. In this network, only the weights of the output layer are updated each training cycle, whereas, the weights of the hidden layer are kept unchanged. Hence, the number of updated weights in each case is equal to the dimension of the matrix to be inverted. The training procedures presented in chapter 4; namely, SDLS, CGFR, and CGPR methods are applied to all of these test cases to compare their performances as to the successful convergence within an iteration limit or when the termination condition of learning is met. For comparison, the

corresponding results for the conventional BP method are also included where the step size of this method is calculated through equation (5.1). Similar to the *LU*-decomposition problem, it is found that only when the initial weights, $\underline{w}_o$, for the BP method take the values of zero, the training is successful. Therefore, for comparison purposes, all methods start from the same initial weight vectors ($\underline{w}_{init}$ = 0) and receive the same sequence of training patterns. Also, the threshold value (ε_1) of the total squared error is kept the same at a value of 10^{-9}, whereas, the threshold value (ε_2) of the batch gradient error is kept the same at a value of 10^{-8}. The iteration limit is taken as 10000 iterations for the SDLS and BP methods and as 210 cycles for the conjugate gradient methods.

Similar to the *LU*-decomposition problem, an inexact line search; namely, bracketing with golden section method is used with the SDLS method. Various degrees of inexactness were tested and the best learning curves obtained for selected cases are given in figure 5.21. In this figure, each graph expresses the total squared error, $E^N(\underline{w})$, as a function of the learning time for all training methods. From the figure, it is clear that, except for the SDLS method, all methods have almost the same convergence rate in all test cases. Furthermore, the CGFR method gives the best convergence speed followed by the CGPR and BP, respectively. As can be seen, all runs for SDLS and CGPR methods converge successfully to a total squared error value, which is below 10^{-8}. In fact, the learning curves for the SDLS method are very similar in accuracy to those of the CGPR method, but are different in convergence rates and this difference increases as the size of both matrix and training set increases. It is also interesting to note that all runs for BP and CGFR methods reach, in general, plateaus that become closer to one another as the size of both matrix and training set increases. Comparison between the learning curves of both SDLS and BP methods indicates that the BP method has a better convergence rate than the SDLS method, which means that SDLS method takes longer actual computational time. However, the BP method achieves much worse accuracy than SDLS method. The difference between convergence rates of these two methods becomes more pronounced as the size of both matrix and training set increases.

The terminated number of iterations, averaged running time, convergence accuracy, number of objective function evaluations, and number of gradient evaluations for all the simulated methods are summarized in Tables (5.15 – 5.20) for Cases (12 – 17), respectively.

In order to further assess the computational efficiency of the training methods, a comparison in terms of function evaluations is carried out. For instance, SDLS, CGFR, and CGPR methods in Case 12 require 2766, 1855, and 1849 function evaluations, respectively, to reach the global minimum, while the BP method requires 1616 function evaluations to reach a plateau of 10^{-5} under the same conditions. This explains why the running times for CGFR, and CGPR methods are almost in the same of order of magnitude for Case 12. Another interesting comparison is between Cases 16 and 17, where it is clear that the CGPR and SDLS methods reach almost the same convergence accuracy (global minimum); however, the CGPR method is much superior to the SDLS method in terms of convergence speed.

Table 5.15
Detailed simulation results for Case 12 under four training methods

Method	CGFR (r = 3)	CGPR (r = 3)	SDLS	Conventional BP
Running time (s)	136.40	121.32	206.98	172.70
Terminated iteration no	210	154	233	1616
Terminated E	6.17×10^{-8}	9.80×10^{-9}	6.97×10^{-9}	2.70×10^{-5}
Function evaluations	1849	1855	2766	1616
Gradient evaluations	210	154	233	--

Table 5.16
Detailed simulation results for Case 13 under four training methods

Method	CGFR (r = 3)	CGPR (r = 3)	SDLS	Conventional BP
Running time (s)	360.58	274.19	690.74	423.04
Terminated iteration no	210	126	336	1659
Terminated E	4.93×10^{-7}	9.55×10^{-9}	1.32×10^{-9}	3.73×10^{-5}
Function evaluations	2036	1583	3803	1659
Gradient evaluations	210	126	336	--

Table 5.17
Detailed simulation results for Case 14 under four training methods

Method	CGFR (r = 3)	CGPR (r = 3)	SDLS	Conventional BP
Running time (s)	858.37	623.19	2471.88	1130.90
Terminated iteration no	210	137	434	1698
Terminated E	1.67×10^{-6}	7.80×10^{-9}	1.88×10^{-9}	4.88×10^{-5}
Function evaluations	2127	1722	4822	1698
Gradient evaluations	210	137	434	--

Table 5.18
Detailed simulation results for Case 15 under four training methods

Method	CGFR (r = 3)	CGPR (r = 3)	SDLS	Conventional BP
Running time (s)	1383.70	1283.5	2720.35	1735.20
Terminated iteration no	210	165	317	1661
Terminated E	1.47×10^{-5}	9.04×10^{-9}	8.15×10^{-9}	5.40×10^{-5}
Function evaluations	1935	1999	4693	1661
Gradient evaluations	210	165	317	--

Table 5.19
Detailed simulation results for Case 16 under four training methods

Method	CGFR (r = 3)	CGPR (r = 3)	SDLS	Conventional BP
Running time (s)	3560.87	2573.48	8664.06	4228.94
Terminated iteration no	210	130	427	1681
Terminated E	1.42×10^{-6}	8.29×10^{-9}	8.62×10^{-9}	7.45×10^{-5}
Function evaluations	2088	1631	4847	1681
Gradient evaluations	210	130	427	--

Table 5.20
Detailed simulation results for Case 17 under four training methods

Method	CGFR (r = 3)	CGPR (r = 3)	SDLS	Conventional BP
Running time (s)	8088.70	8189.70	29171.55	13100.00
Terminated iteration no	210	175	490	1751
Terminated E	6.98×10^{-6}	1.50×10^{-9}	2.11×10^{-9}	9.7×10^{-5}
Function evaluations	1809	2090	5489	1751
Gradient evaluations	210	175	490	--

The superiority of the CGPR method is also reflected in the optimal step sizes, η, obtained when using it. These are compared with the optimal step sizes obtained using the CGFR and SDLS methods in figures (5.22) – (5.27). On average the CGPR optimal step sizes are much larger than those of the CGFR and SDLS methods. In addition, by the inspection of the optimal step size values of the CGFR method, it is found that the method gives zero optimal step sizes at final training stages. This explains why the CGFR learning curve reaches a plateau. By comparing figures (5.22) – (5.27), it can be stated that SDLS method provides optimal step sizes that are one magnitude smaller than the optimal step size of other two methods.

To study the effect of matrix size on the behavior of all methods, the values of the total squared error versus the number of gradient updates are plotted in figure (5.28). Since the BP method does not use gradient updates, its learning curve is plotted against the number of objective function evaluations. Obviously, the CG methods have much faster rate of convergence than SDLS and BP methods while the CGPR method has a superior performance to the CGFR method. These results are consistent with those reported in literature on general optimization problems. As shown in figure (5.28a), BP method does not succeed to converge to the global minimum rather it reaches a plateau. Similarly, from figure (5.28c), it can be seen that the CGFR method also fails to reach the global minimum for the Cases (14-17). However, for the Cases (12-13), CGFR method has succeeded to reach the plateau of the global minimum. This implies that CGFR method performs better for relatively small size matrices. Comparing the SDLS and CGPR methods, [figures (5.28b) and (5.28d)],

it can be stated that the SDLS and CGPR methods generally reach the global minimum in a similar fashion although their optimal step sizes behave differently. This can be seen from figures (5.22) – (5.27), which show that the optimal step size of the SDLS is kept small in early stages of training to prevent oscillations and becomes similar to that of CGPR at the final stages of training, i.e., when reaching the neighborhood of the global minimum.

It is interesting to note in figure (5.29) how the variation of matrix size affects the performance of the methods. For all methods, comparing the learning curves at various values of matrix size, it can be stated that the rate of convergence slightly decreases as the matrix size increases till it becomes (200 x 200); afterwards, the decrease in convergence rate becomes more significant.

Despite the fact that the SDLS and CGPR methods have similar learning curves for this problem, their performances differ much as far as the rate of convergence is concerned (see figure 5.29). For all cases, the time required to reach the global minimum in case of CGPR generally is one third of that required for the SDLS method. Also, as can be seen from CGPR learning curves (figure 5.29d), there are a lot of plateau type regions on the error surface (flat regions) which increase significantly as the size of the inverted matrix increases. Thus, it can be stated that CGPR has the ability to find its way out of plateau type regions on the error surface and gives a good performance.

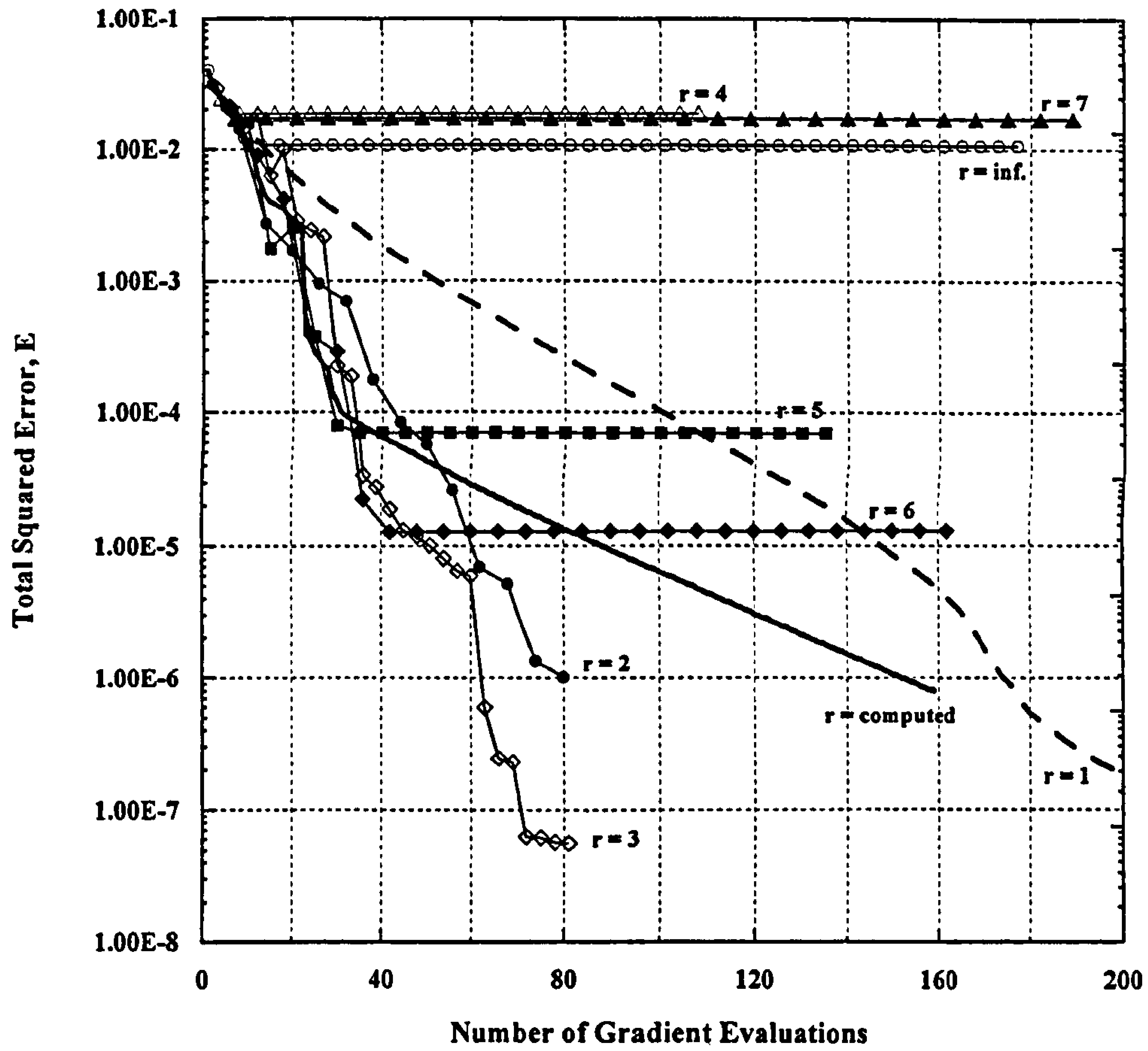


Figure 5.1 Performance of the conjugate gradient training for various reset values.

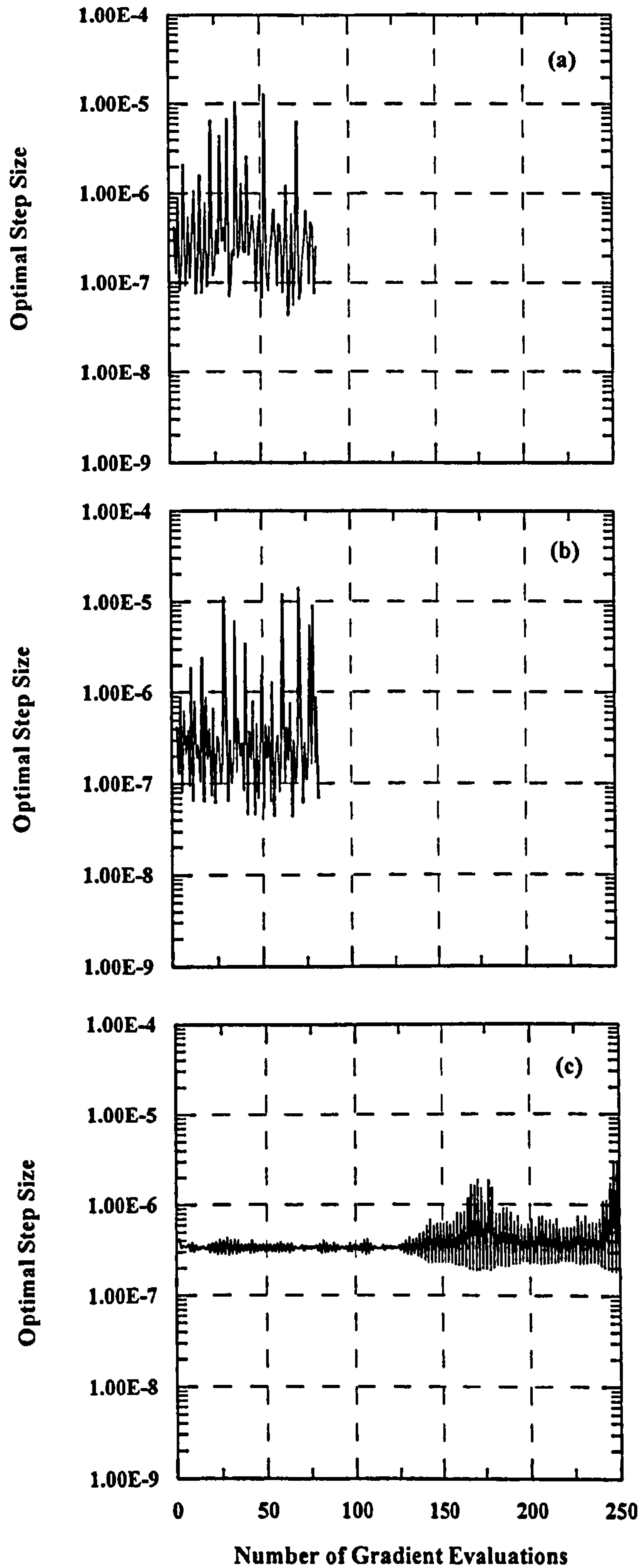
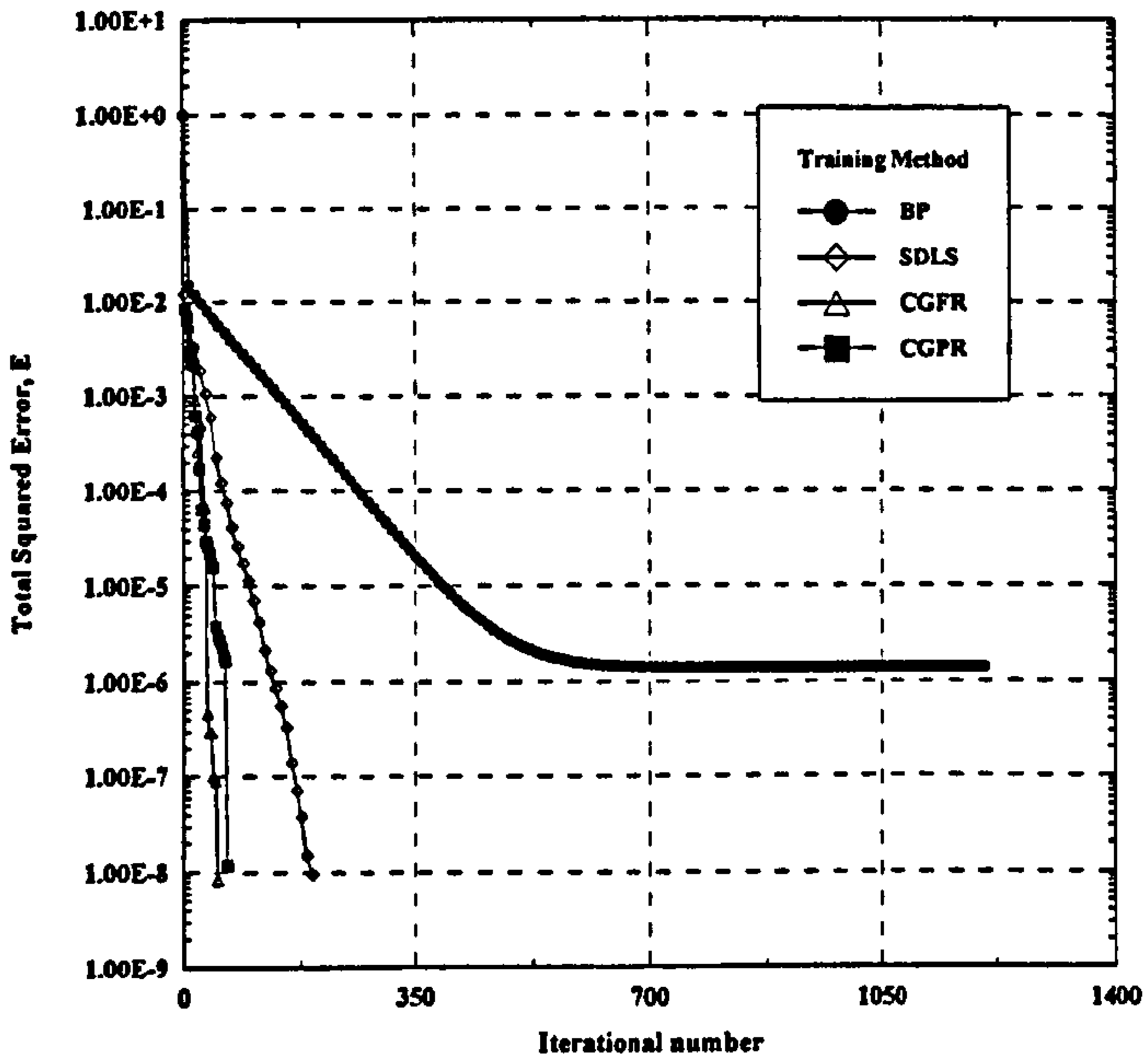
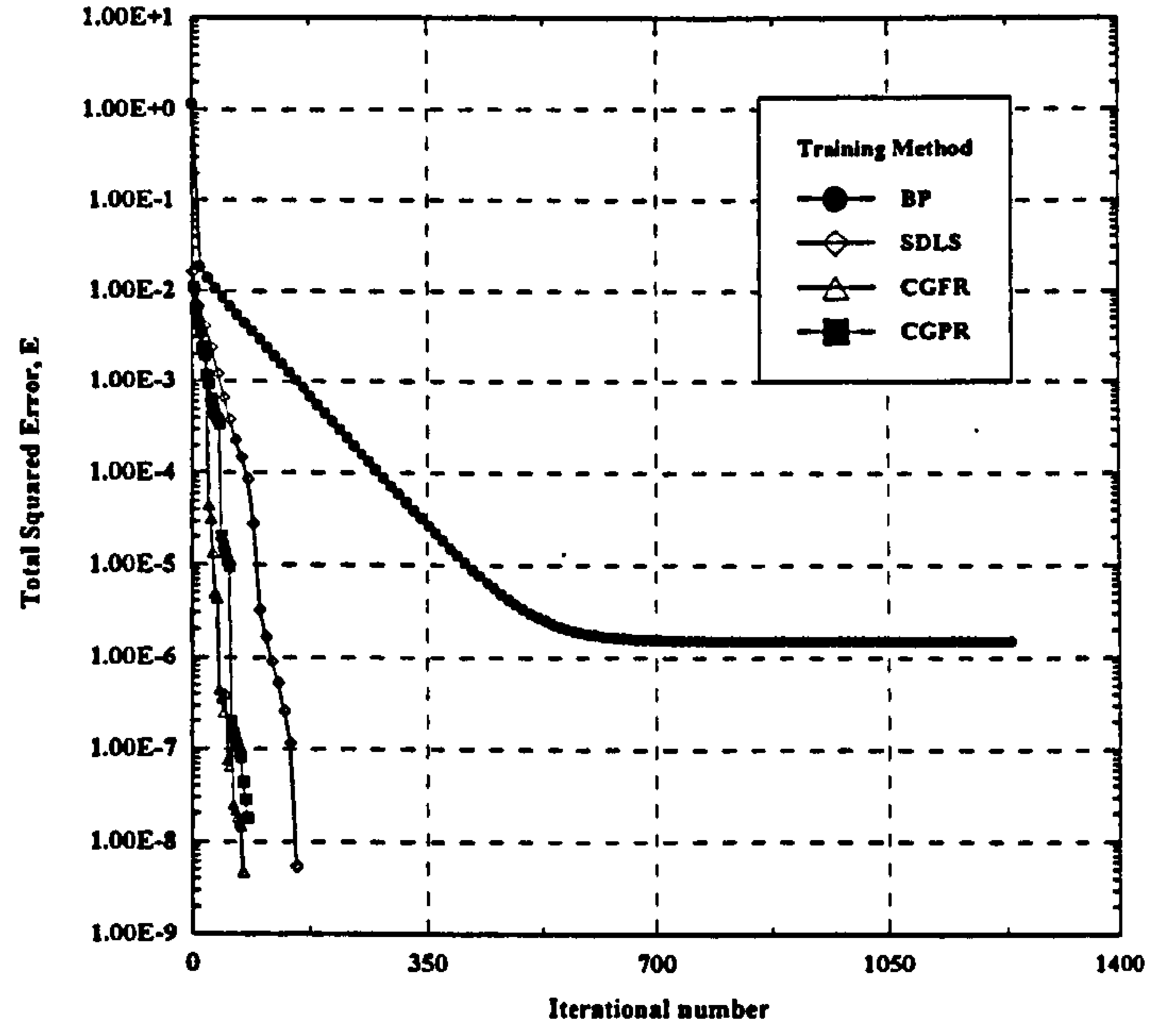


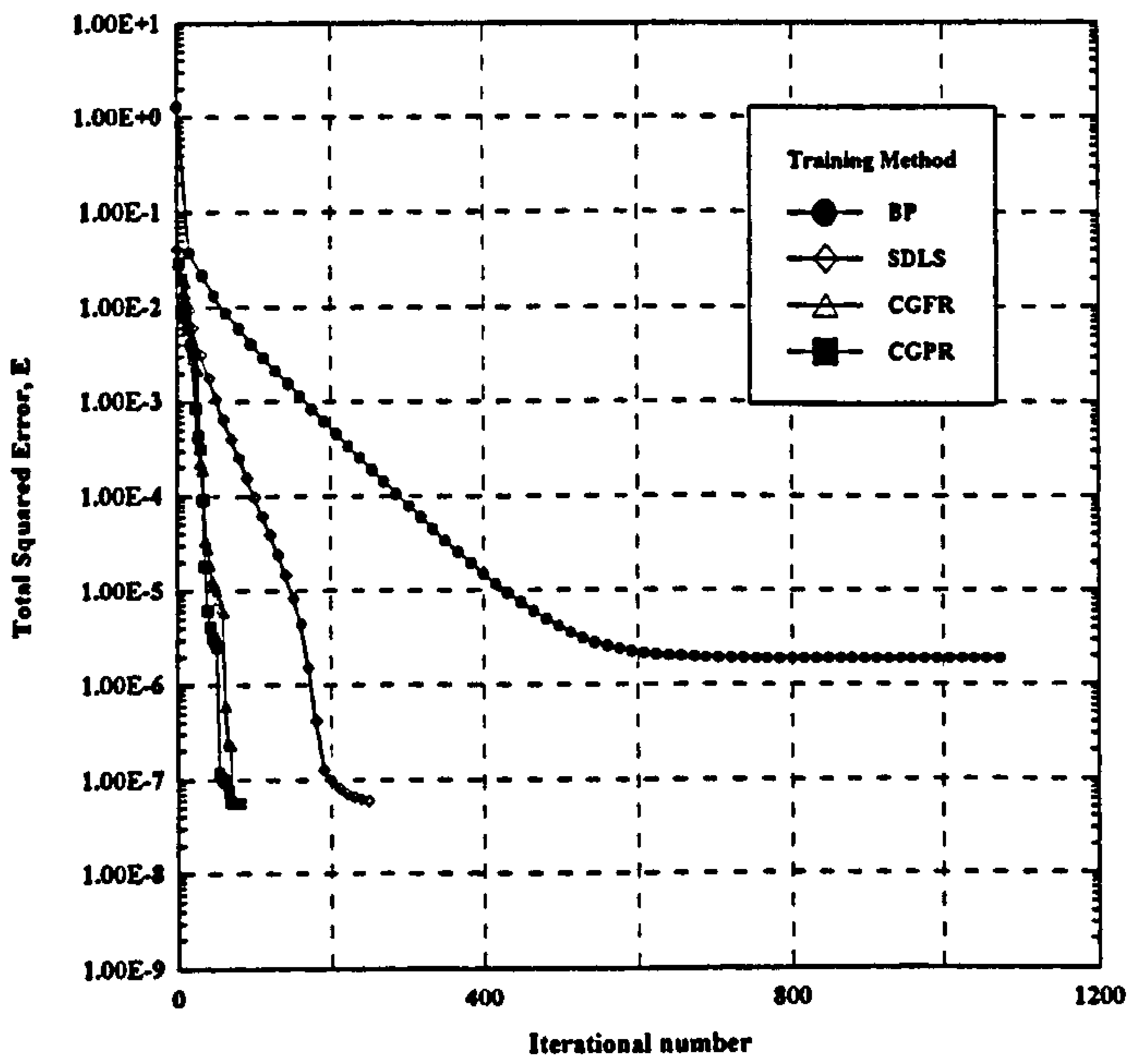
Figure 5.2 Optimal step size behavior for the LU-decomposition of a band matrix (19 x 195) under different training methods: (a) CGPR (r =3); (b) CGFR (r = 3); and (c) SDLS.



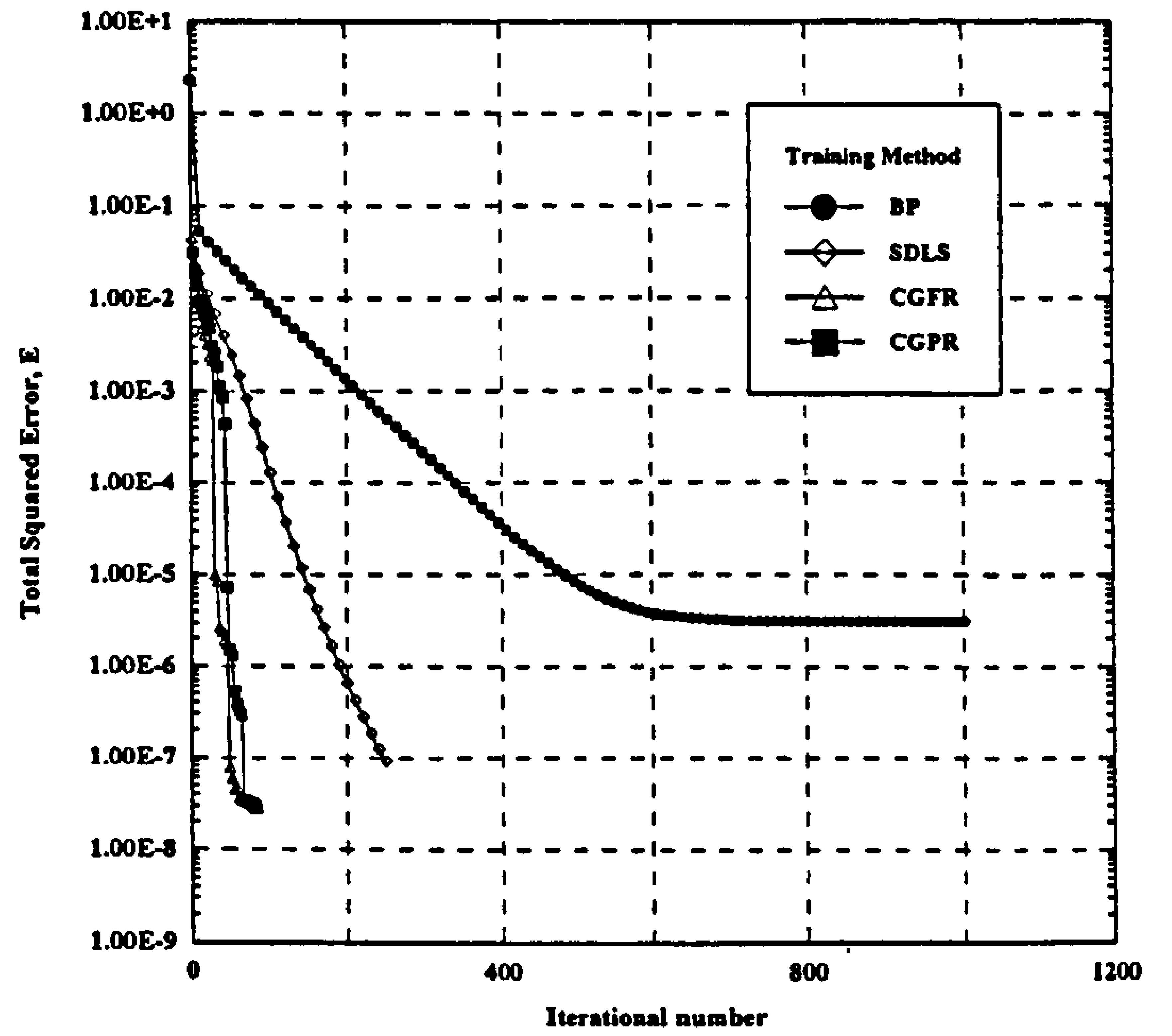
(a) Learning curves for Case 1.



(b) Learning curves for Case 2.

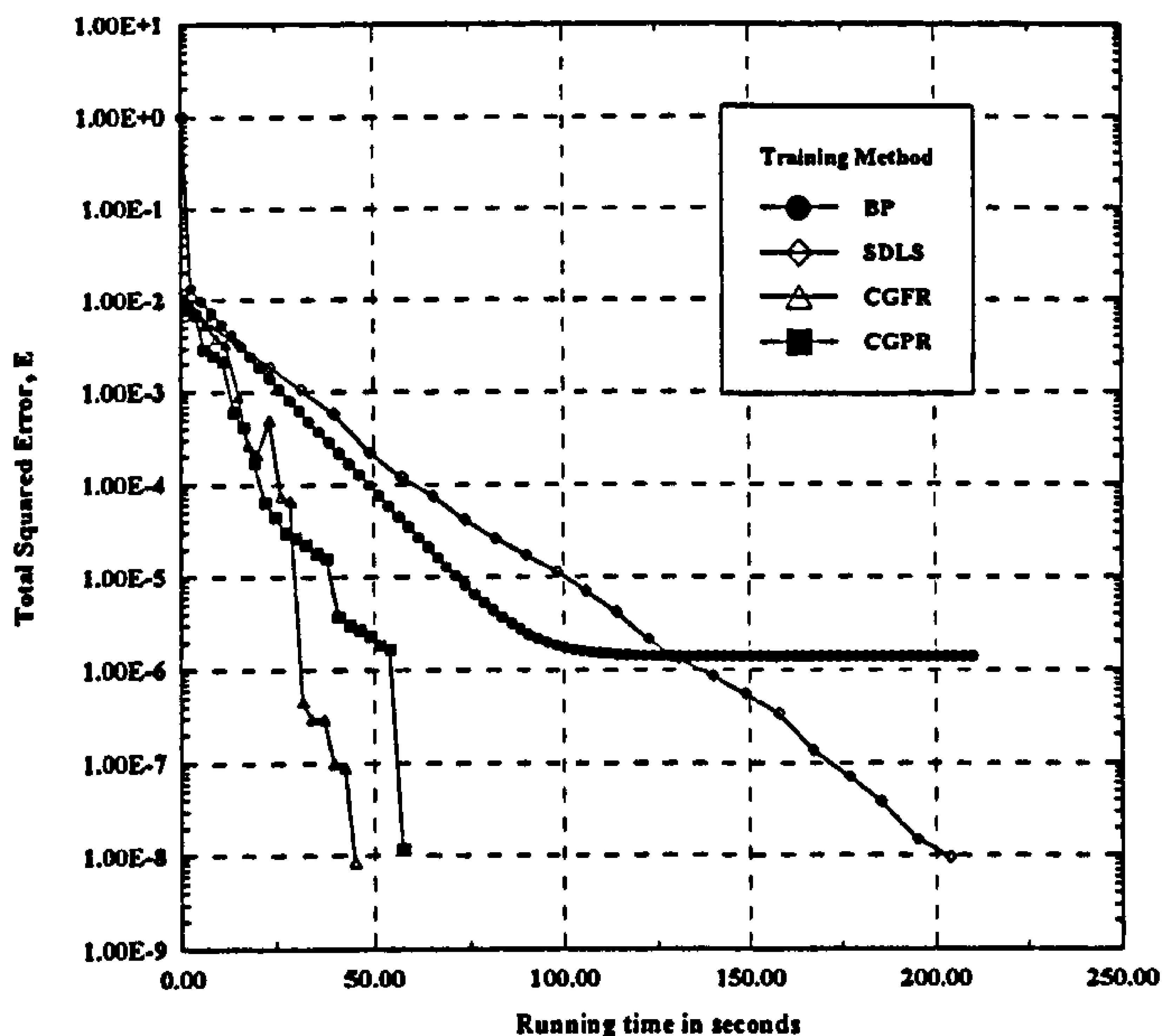


(c) Learning curves for Case 3.

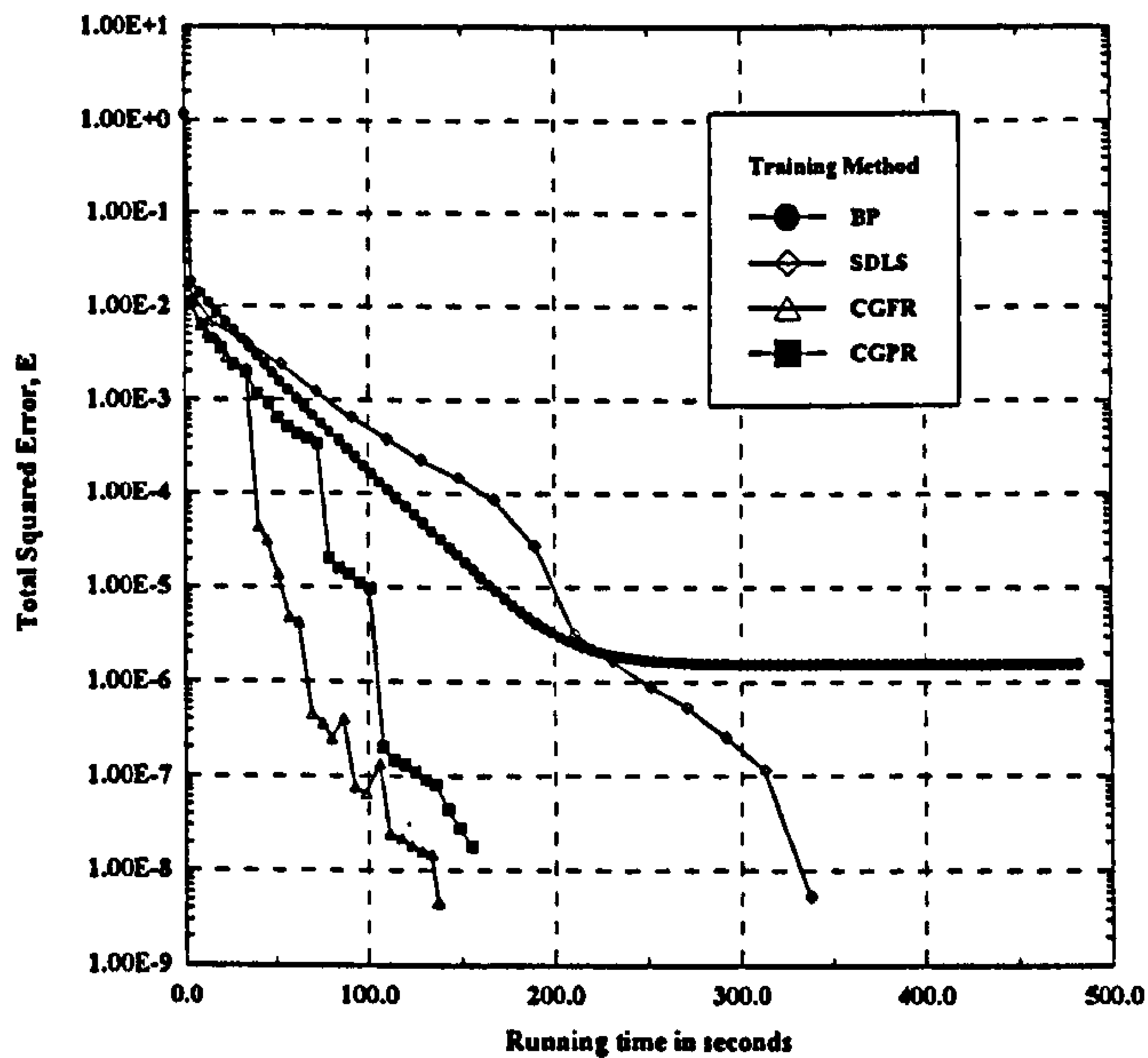


(d) Learning curves for Case 4.

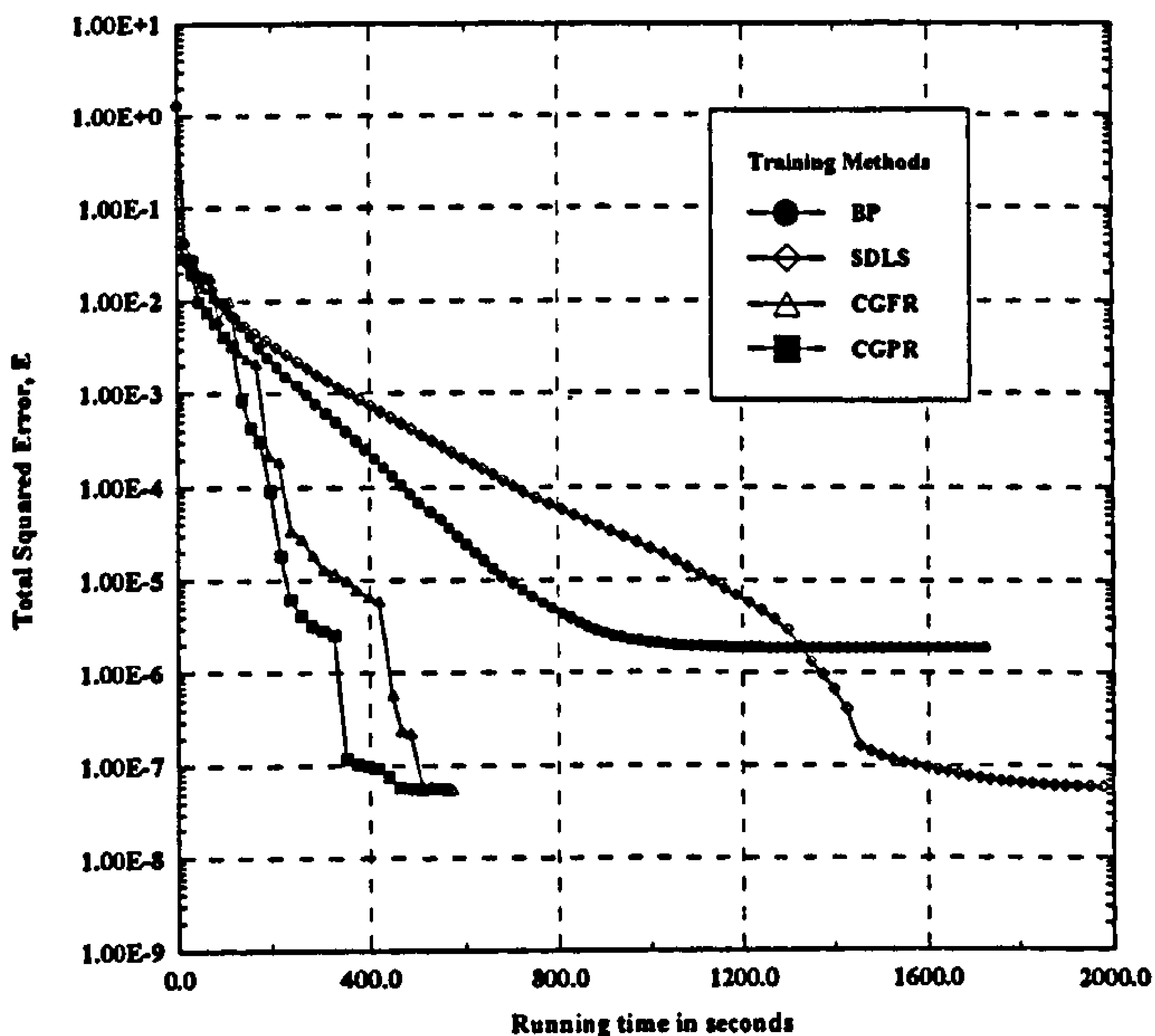
Figure 5.3 Performance comparison of training methods for the LU-decomposition of square matrices having different dimensions.



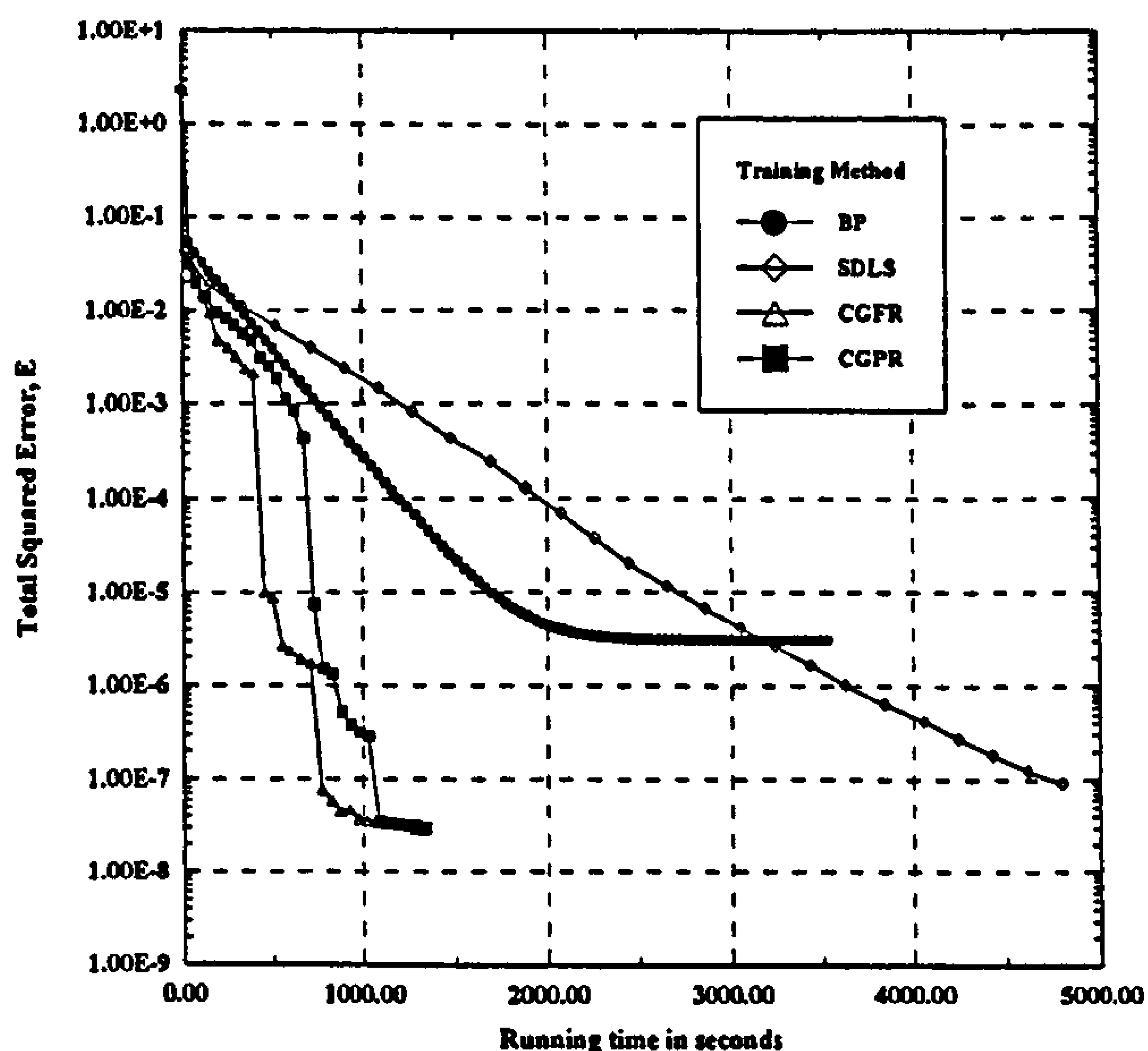
(a) Convergence history for Case 1.



(b) Convergence history for Case 2.

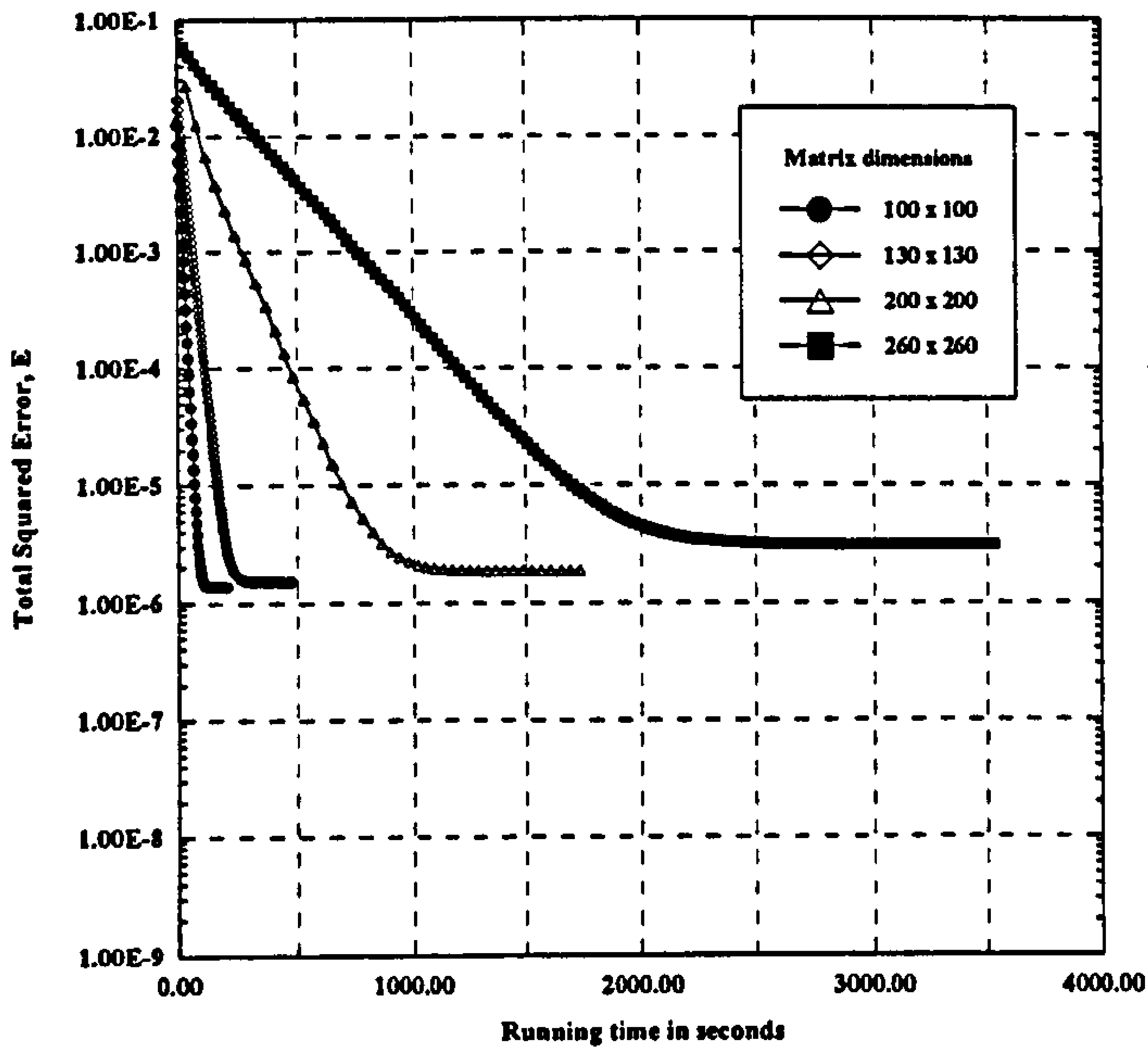


(c) Convergence history for Case 3.

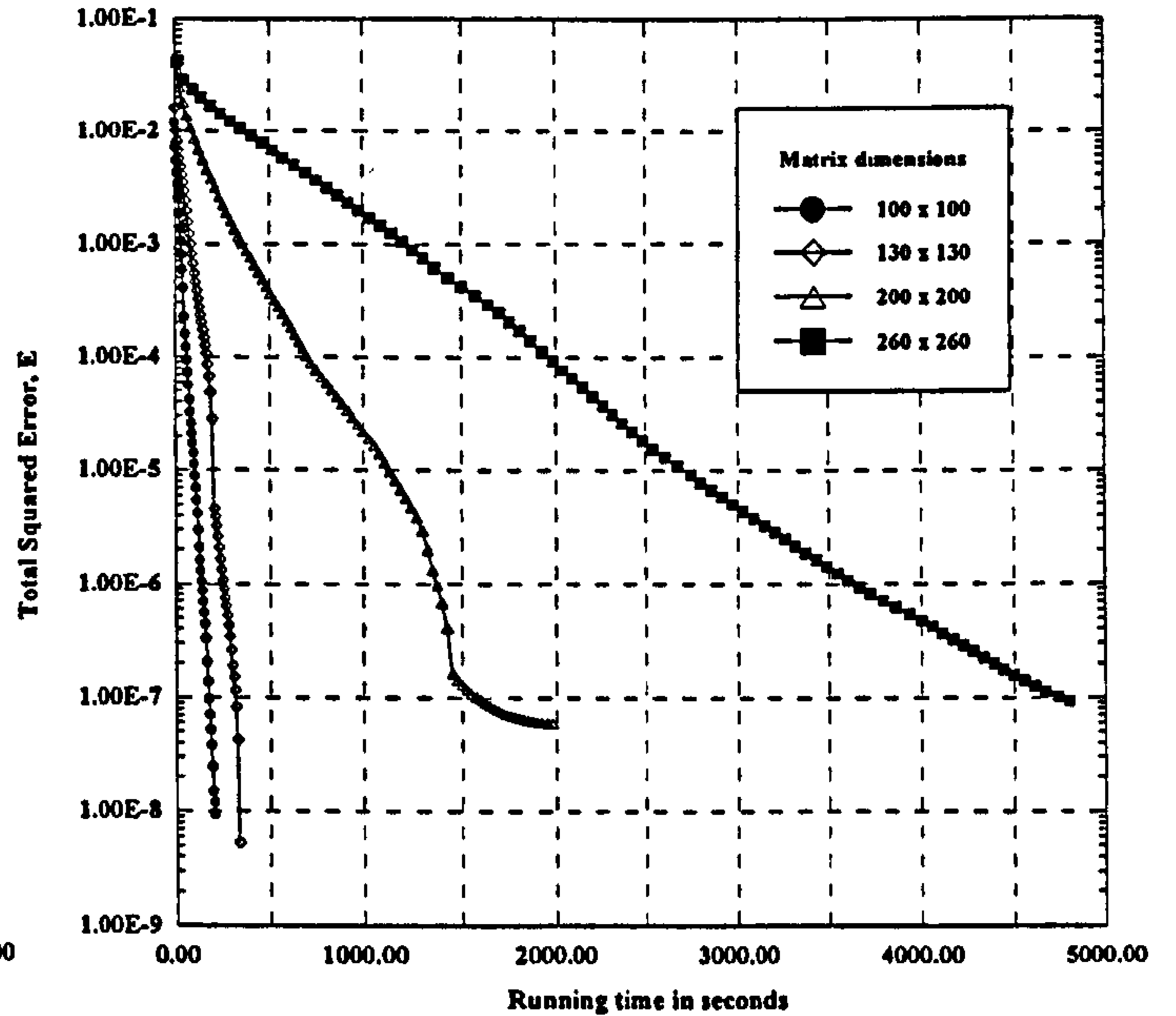


(d) Convergence history for Case 4.

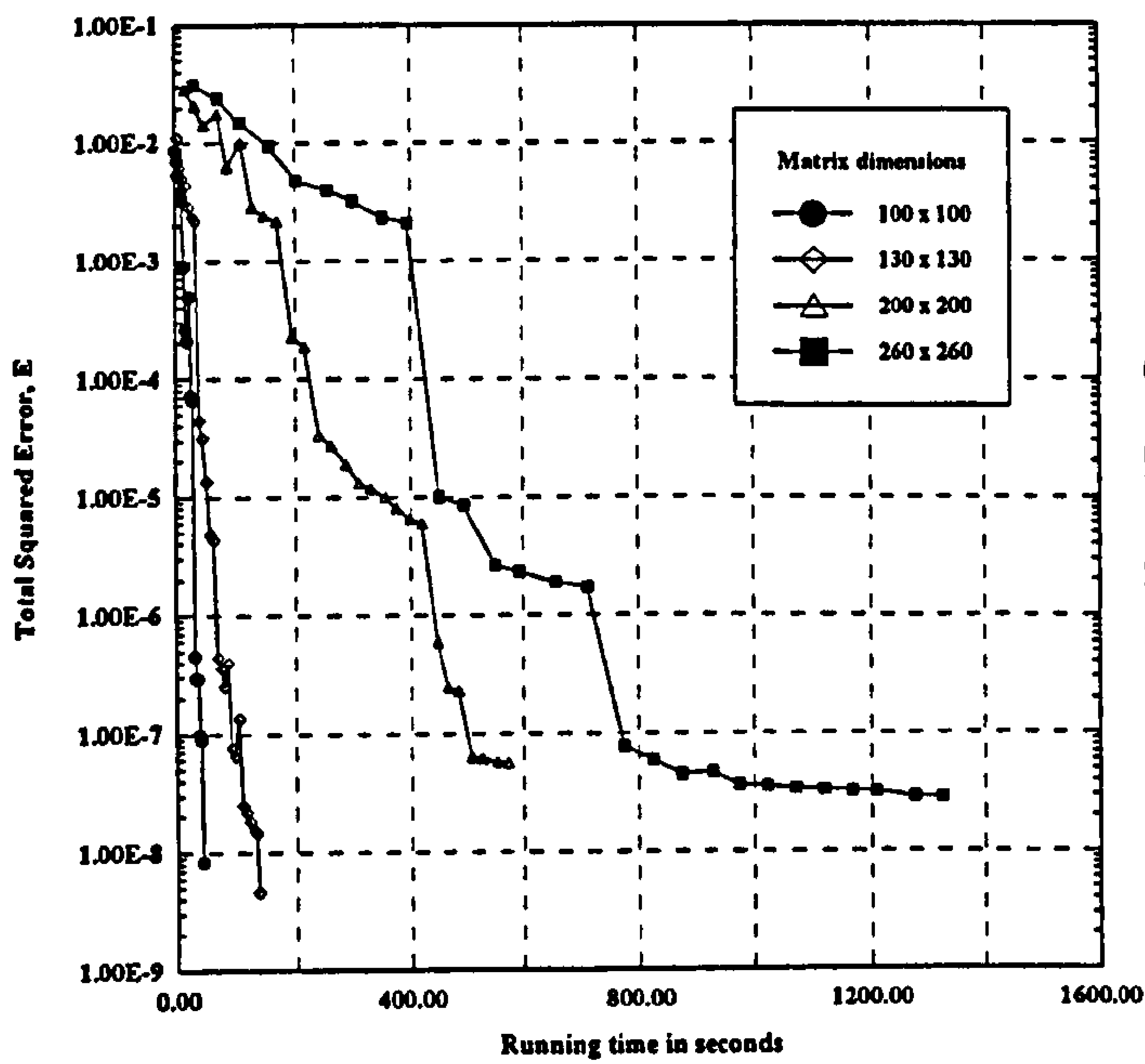
Figure 5.4 Performance of training methods as a function of time for the LU-decomposition of square matrices having different dimensions.



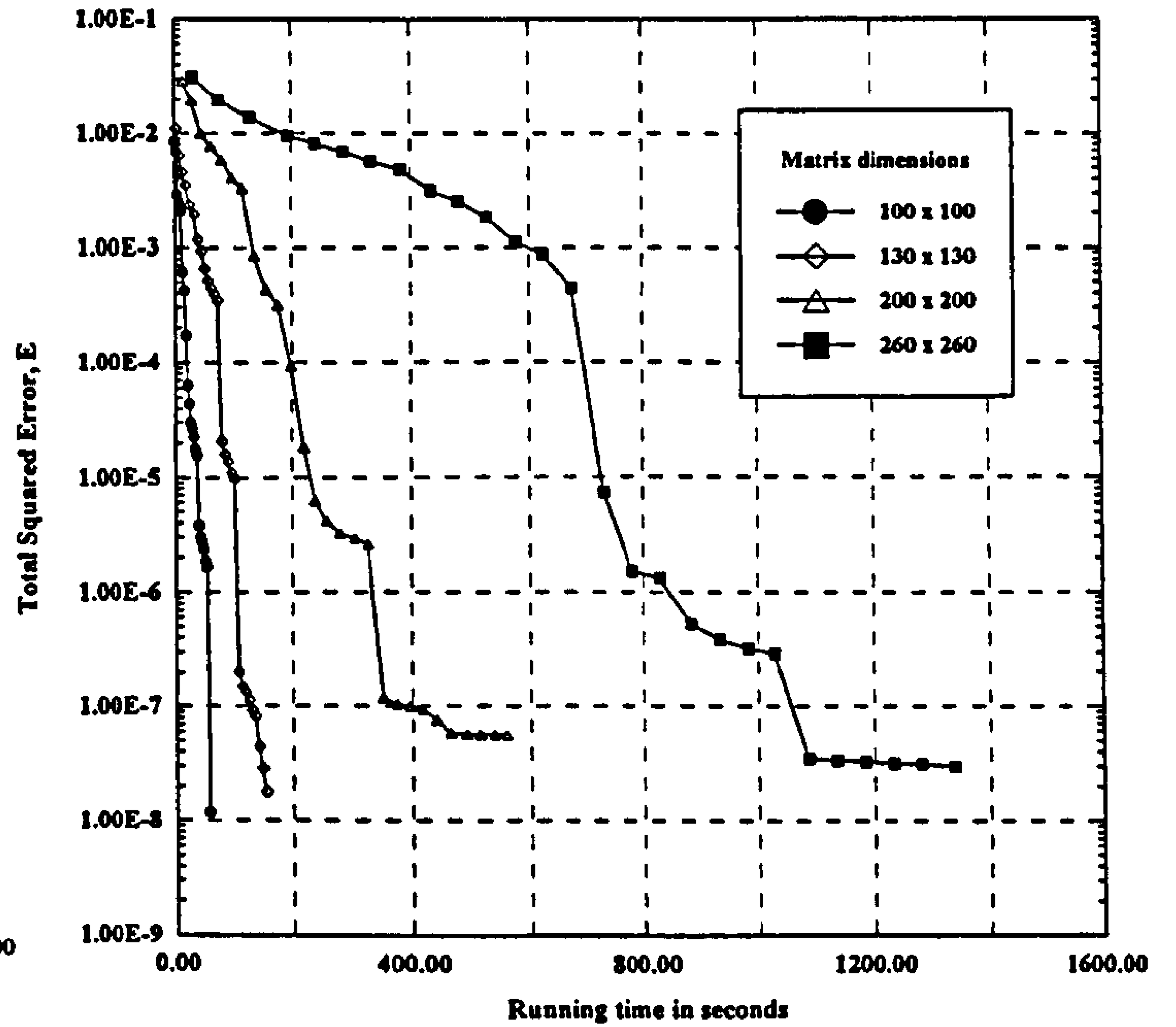
(a) Learning curves for BP method



(b) Learning curves for SDLS method



(c) Learning curves for CGFR method



(d) Learning curves for CGPR method

Figure 5.5 Effect of matrix size on the performance of training methods used for LU-decomposition of sqaure matrices.

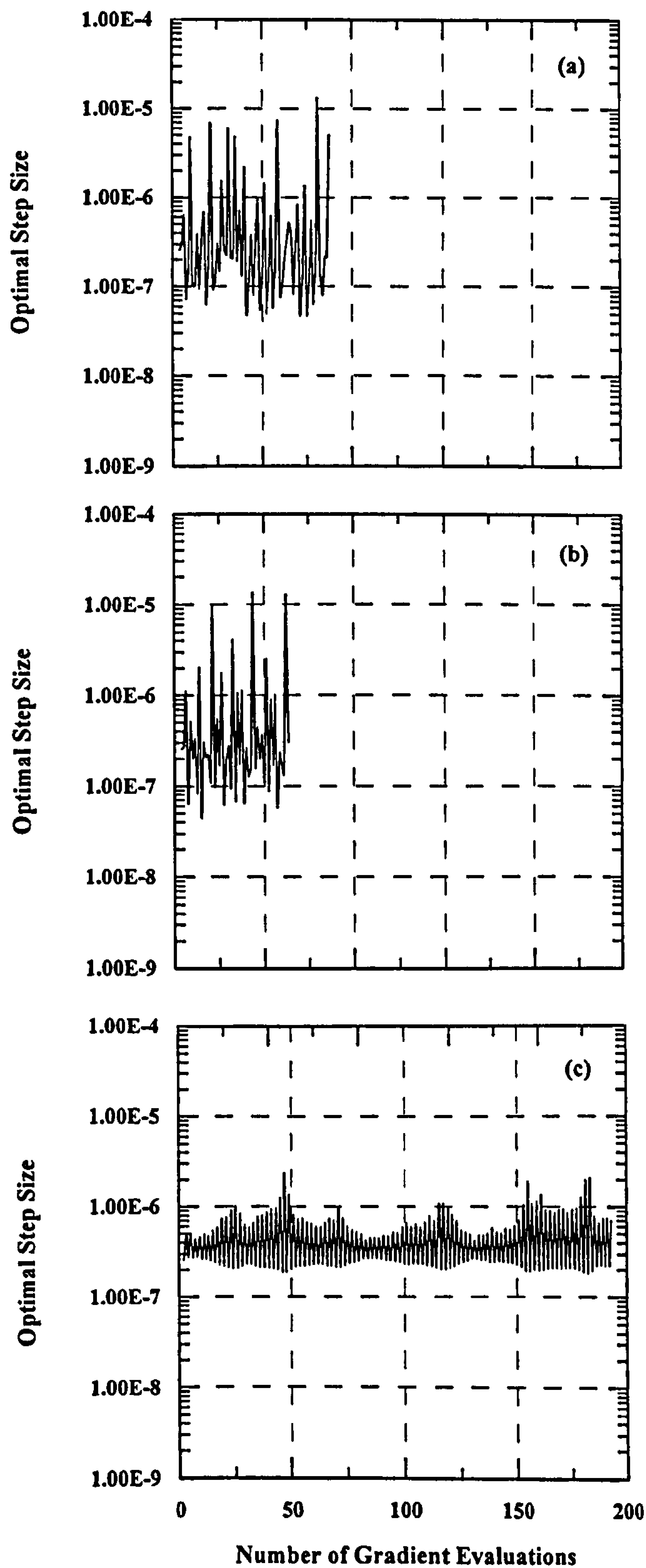


Figure 5.6 Illustration of optimal step size behavior for Case 1 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.

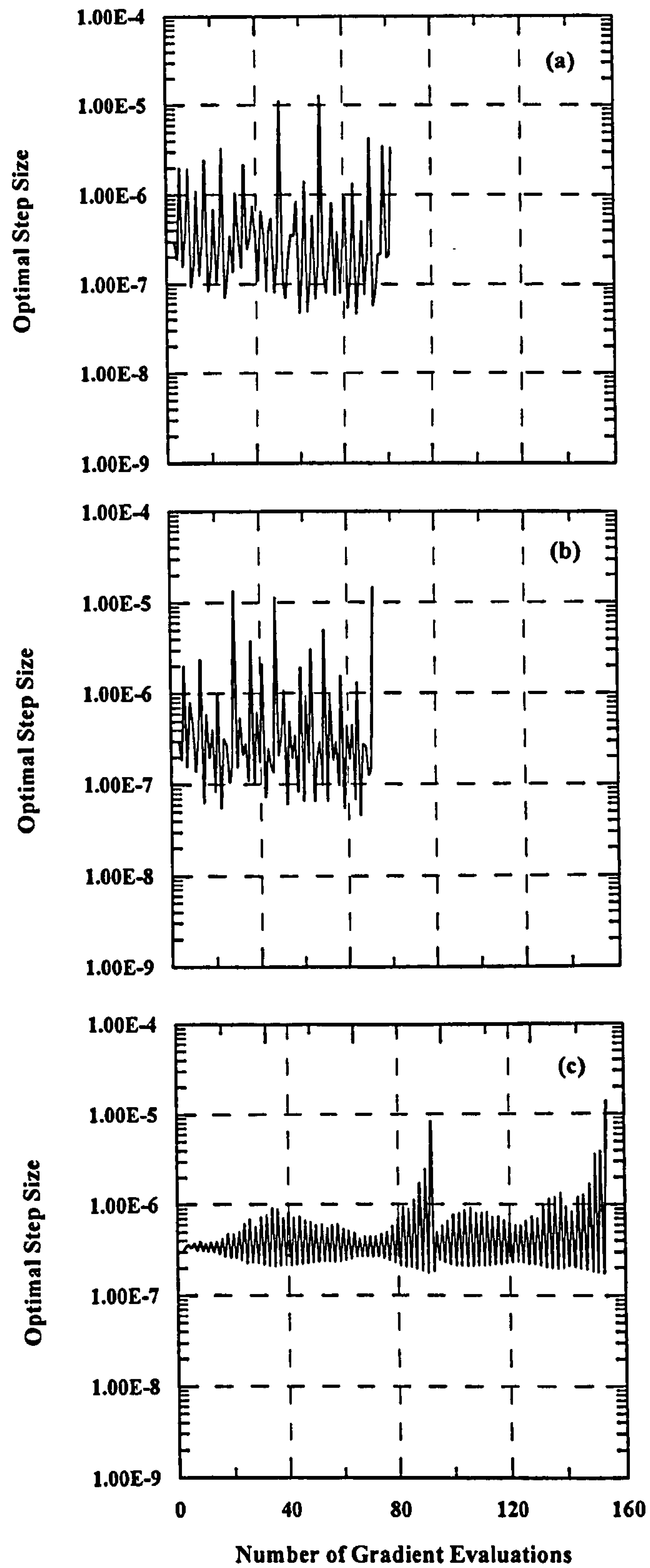


Figure 5.7 Illustration of optimal step size behavior for Case 2 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.

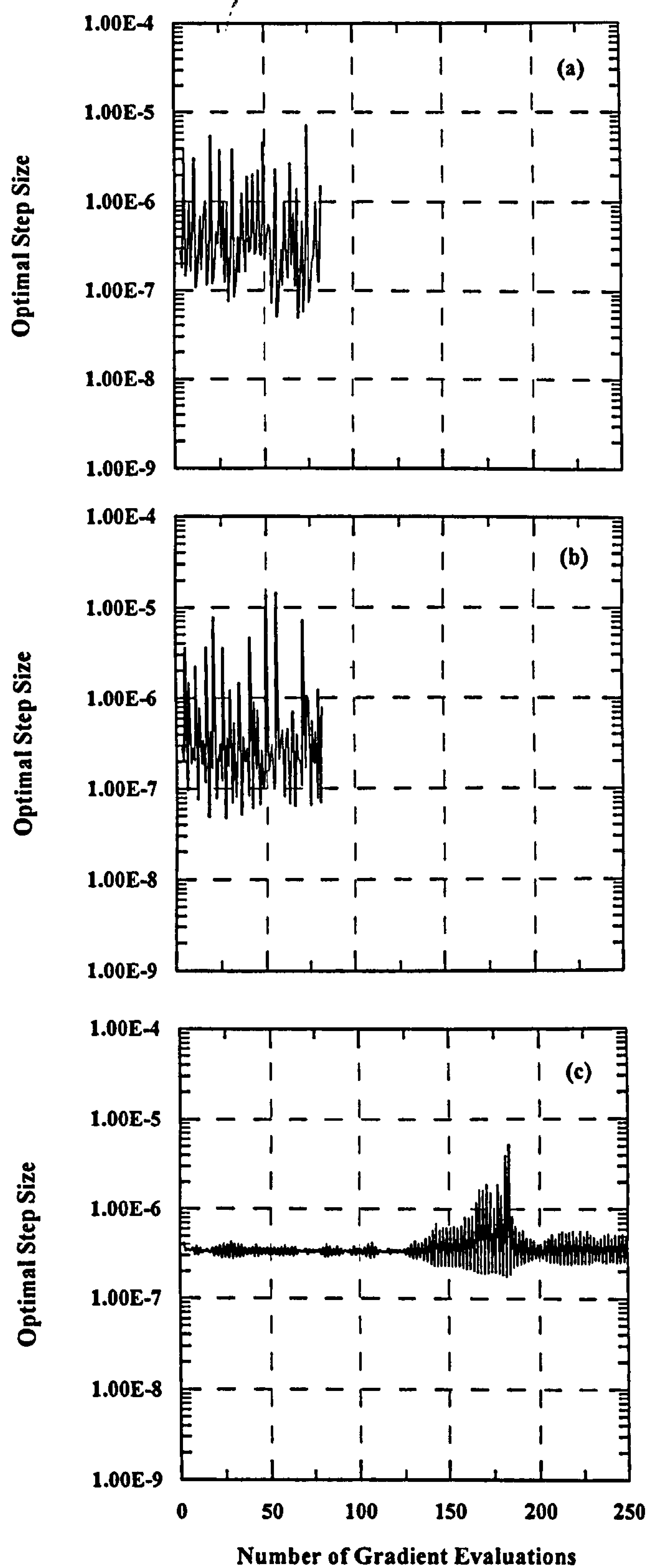


Figure 5.8 Illustration of optimal step size behavior for Case 3 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.

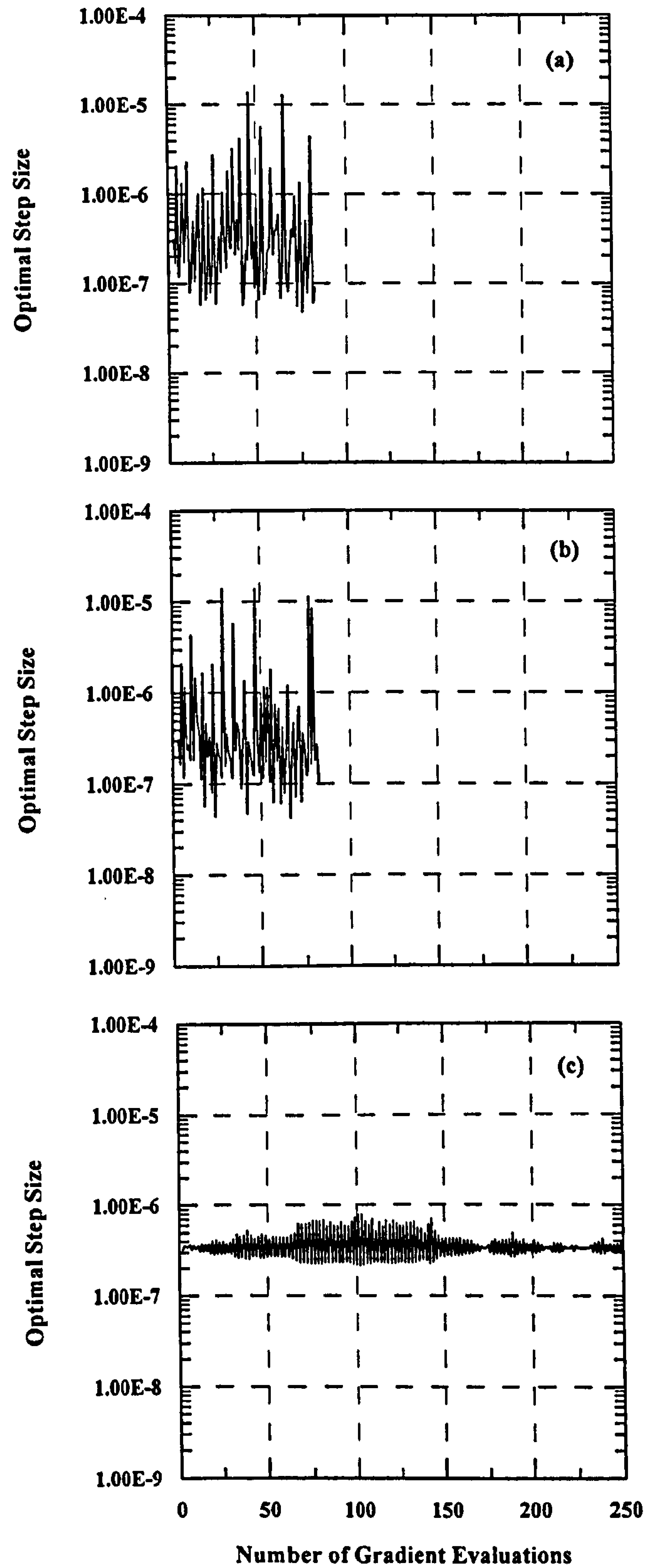
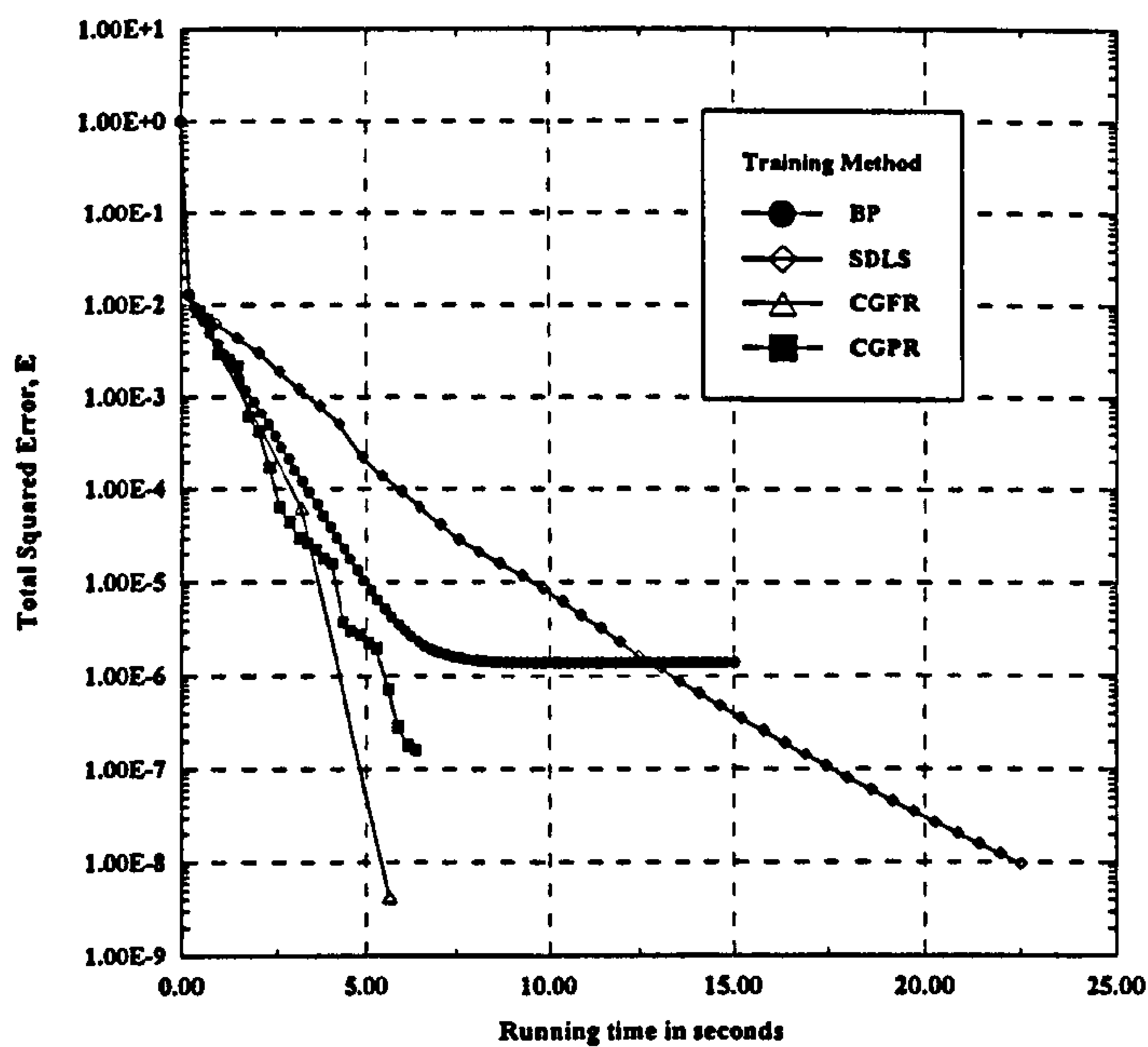
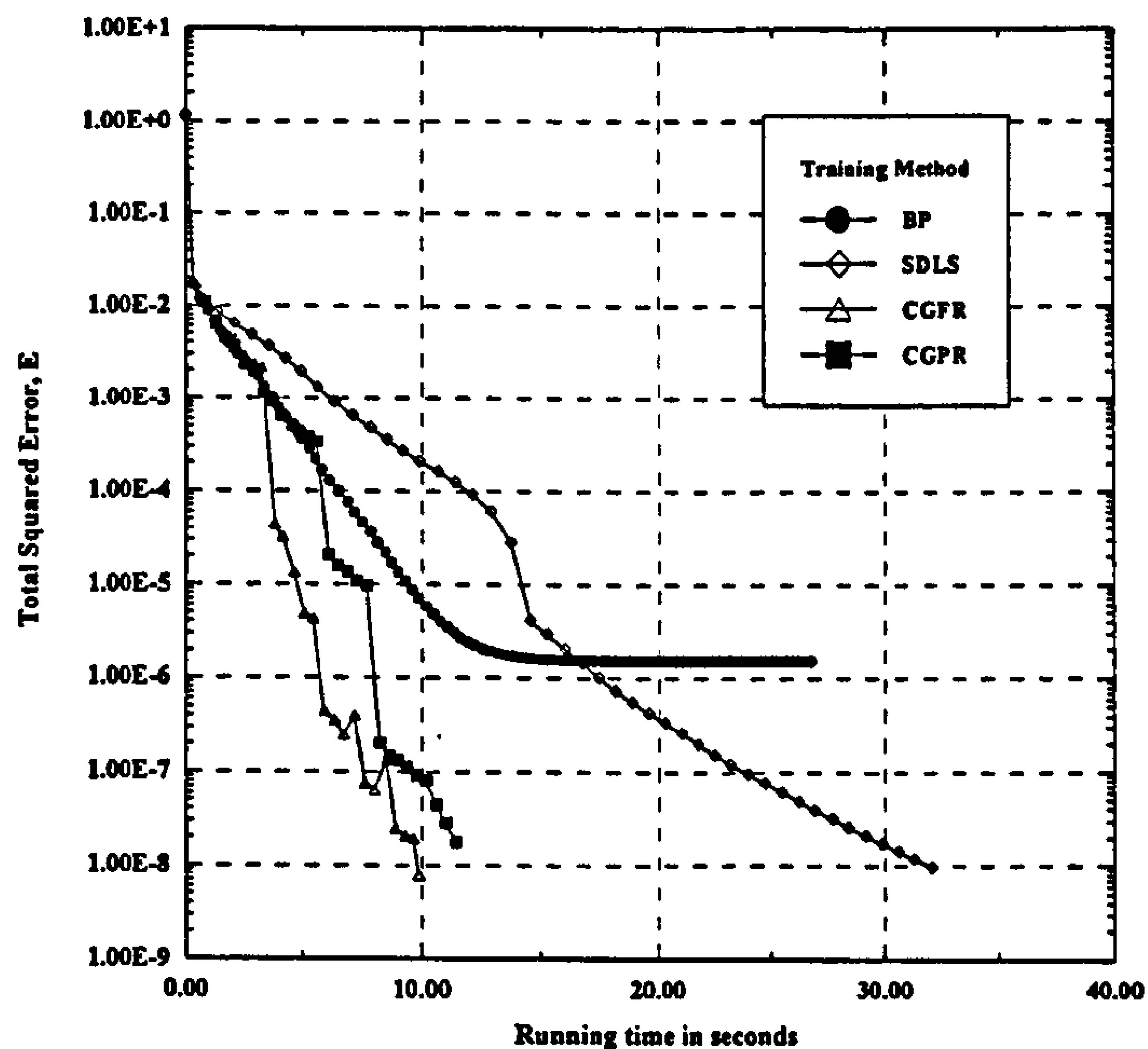


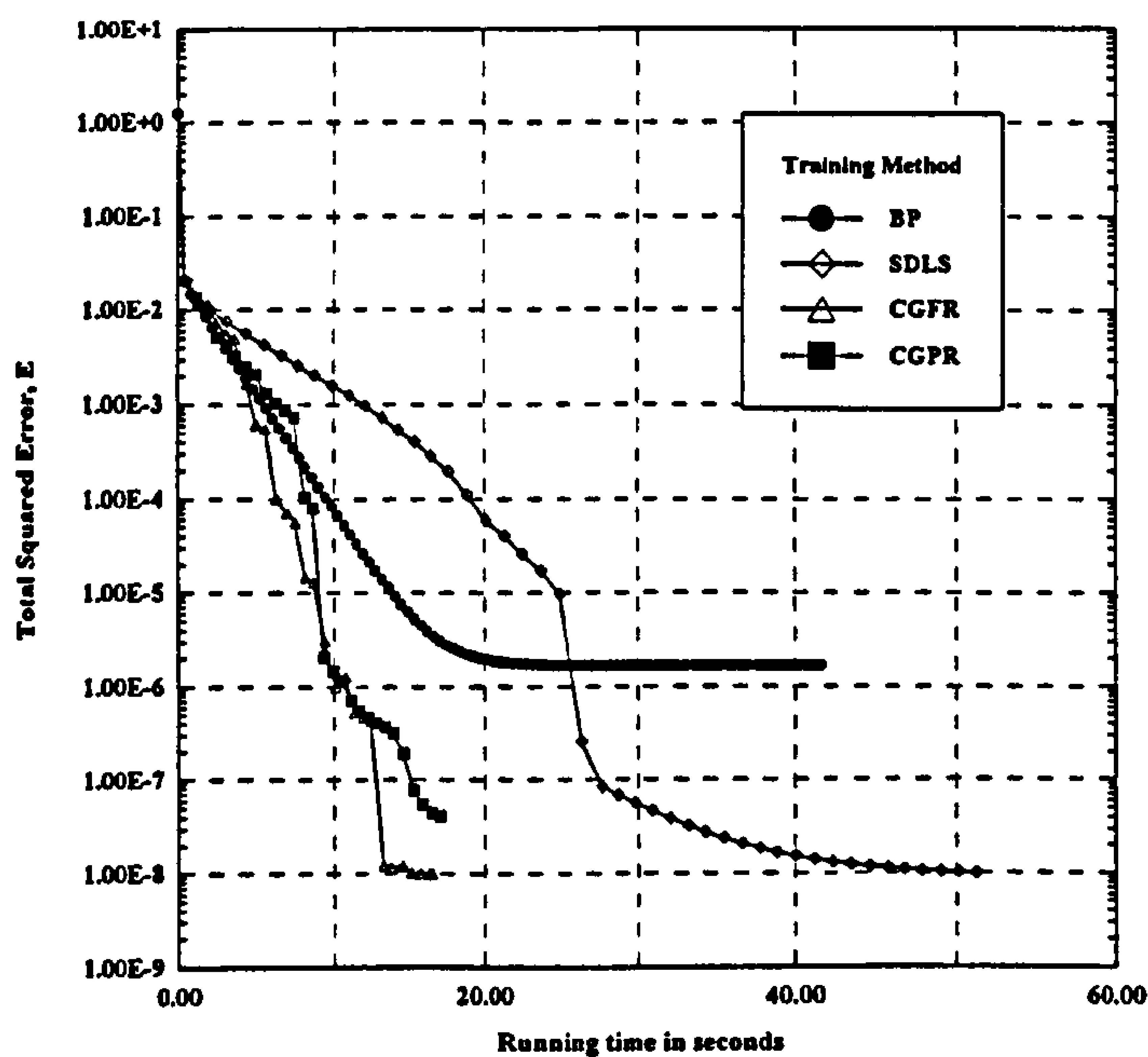
Figure 5.9 Illustration of optimal step size behavior for Case 4 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.



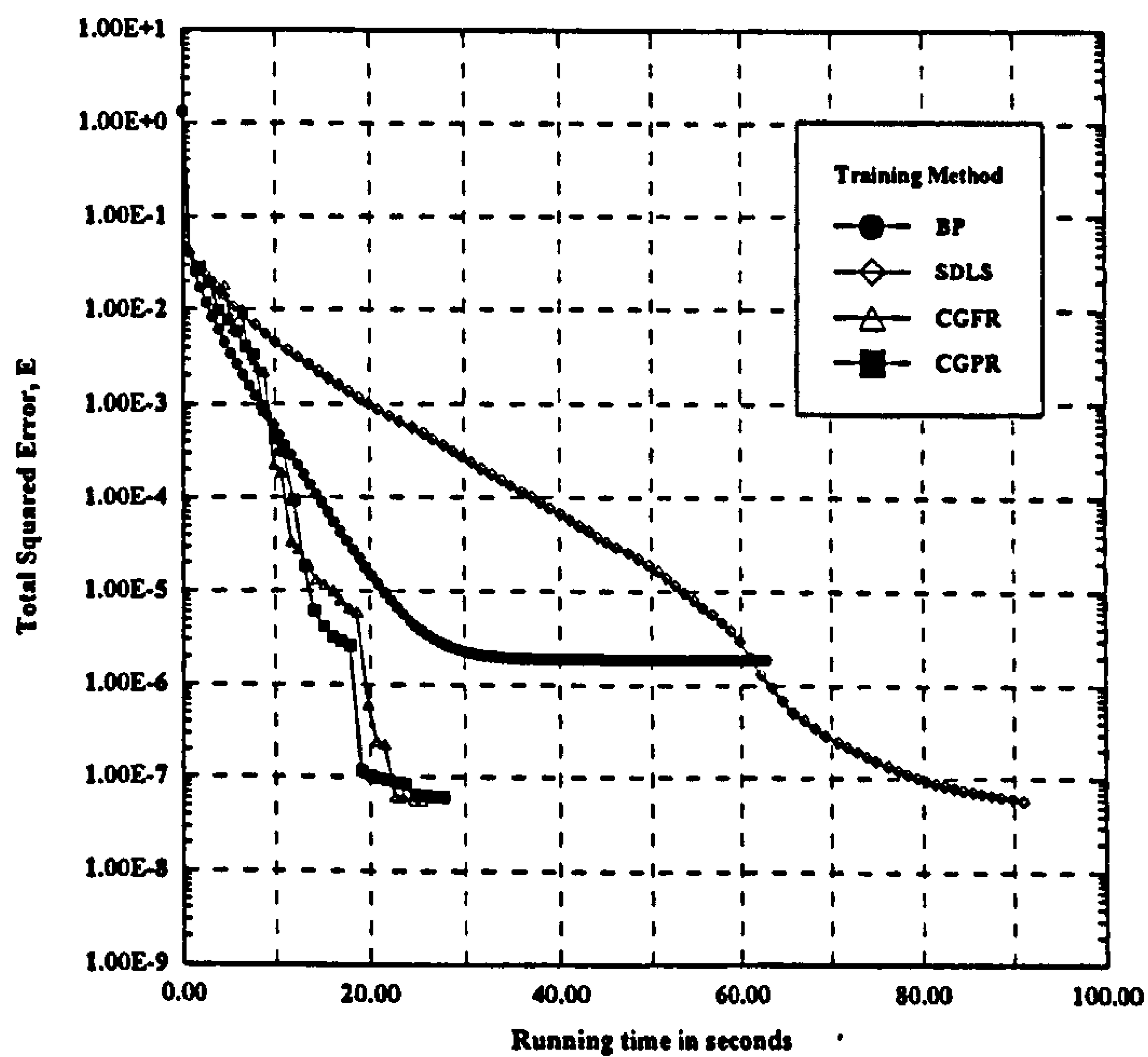
(a) Convergence history for Case 5.



(b) Convergence history for Case 6.

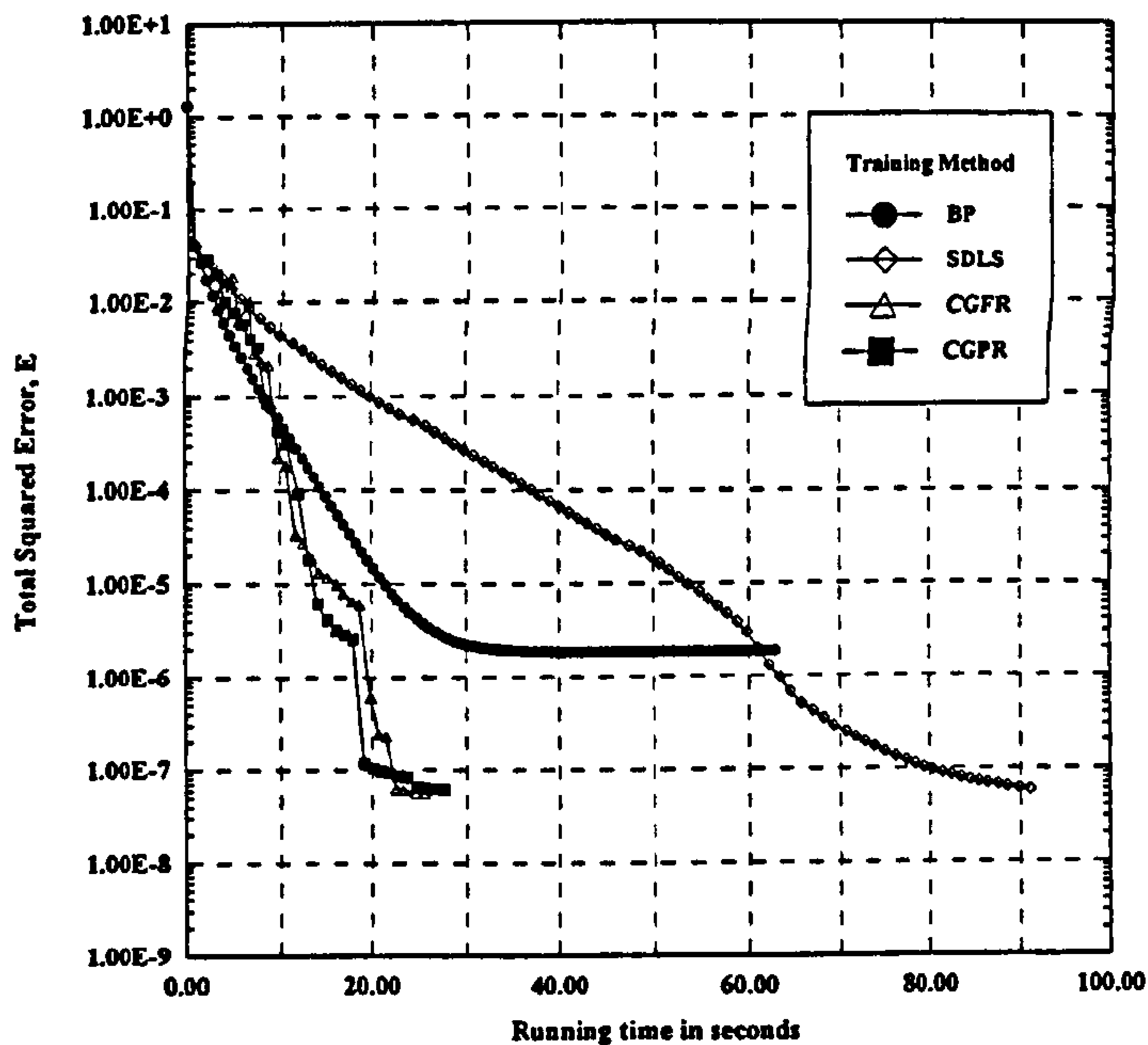


(c) Convergence history for Case 7.

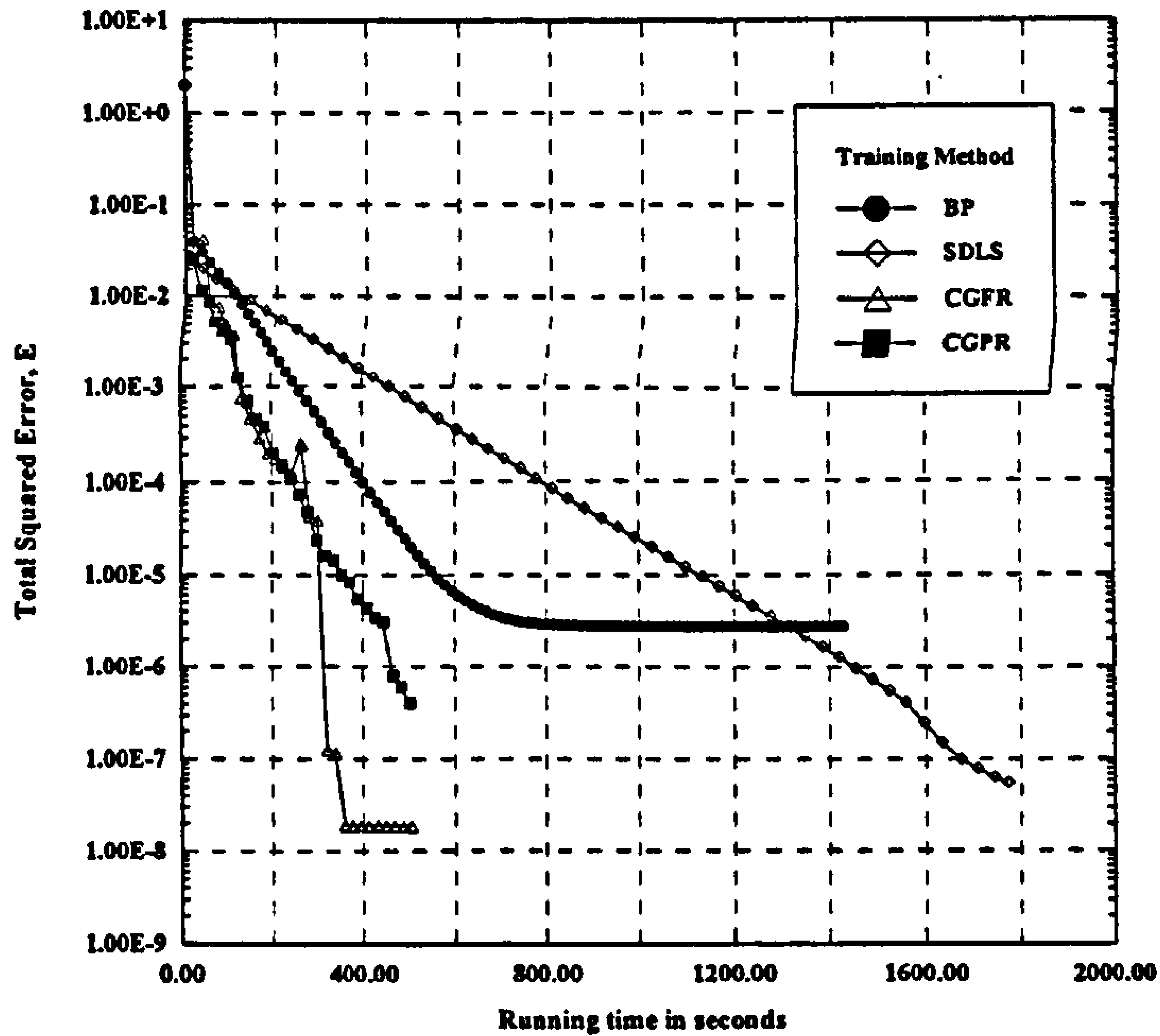


(d) Convergence history for Case 8.

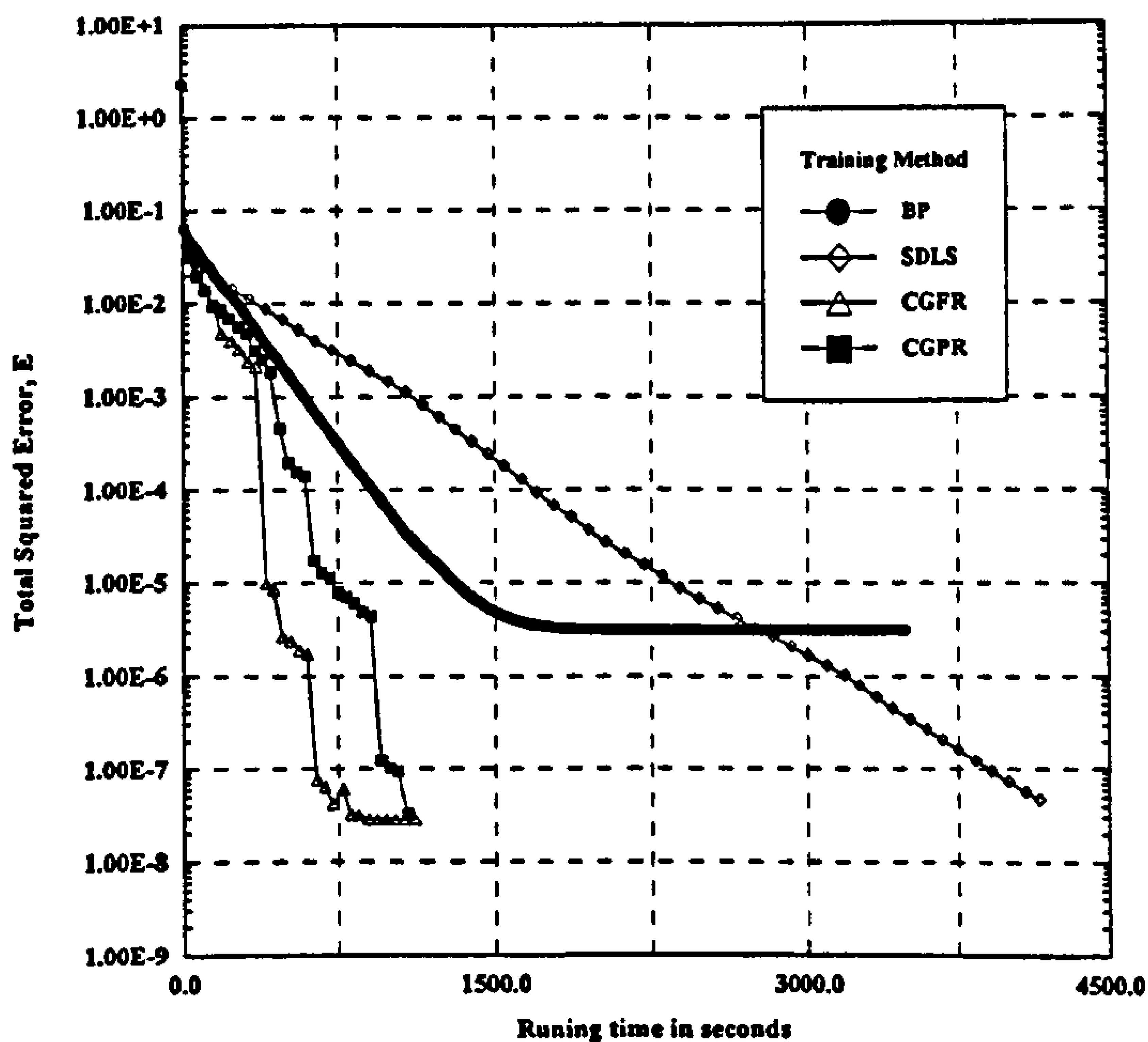
Figure 5.10 Performance of training methods as a function of time for the LU-decomposition of band matrices having the same band-width.



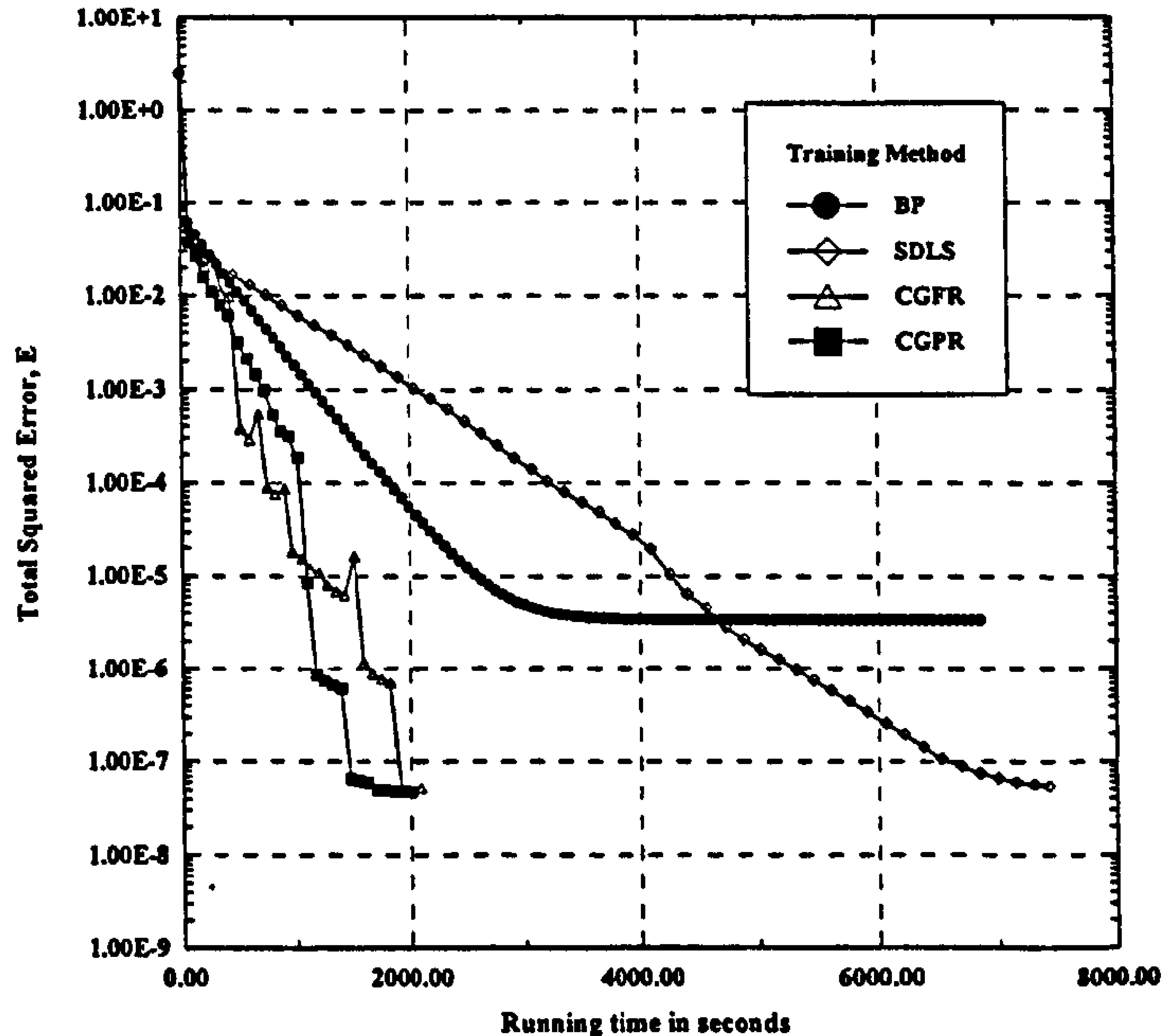
(a) Convergence history for Case 8.



(b) Convergence history for Case 9.

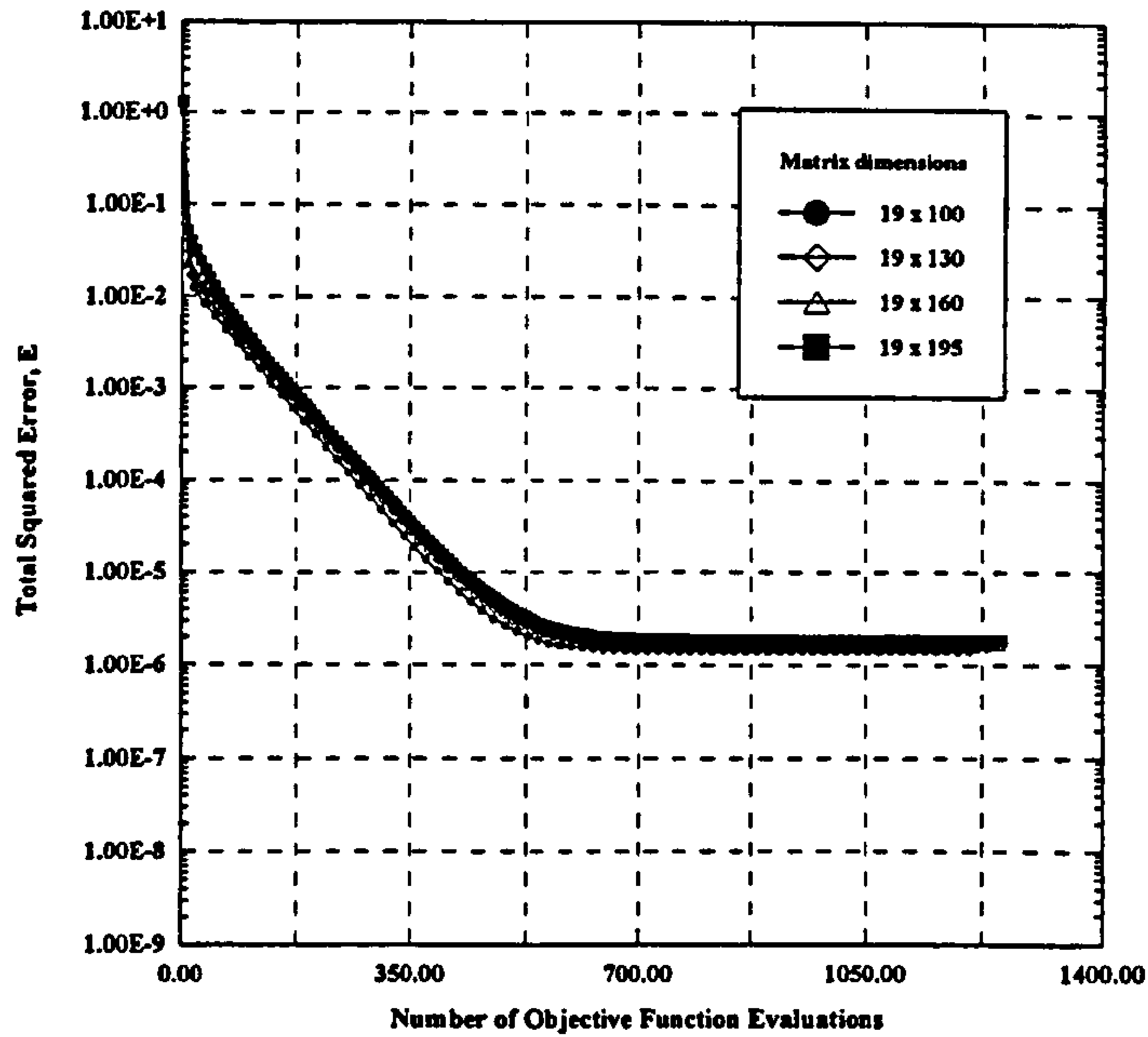


(c) Convergence history for Case 10.

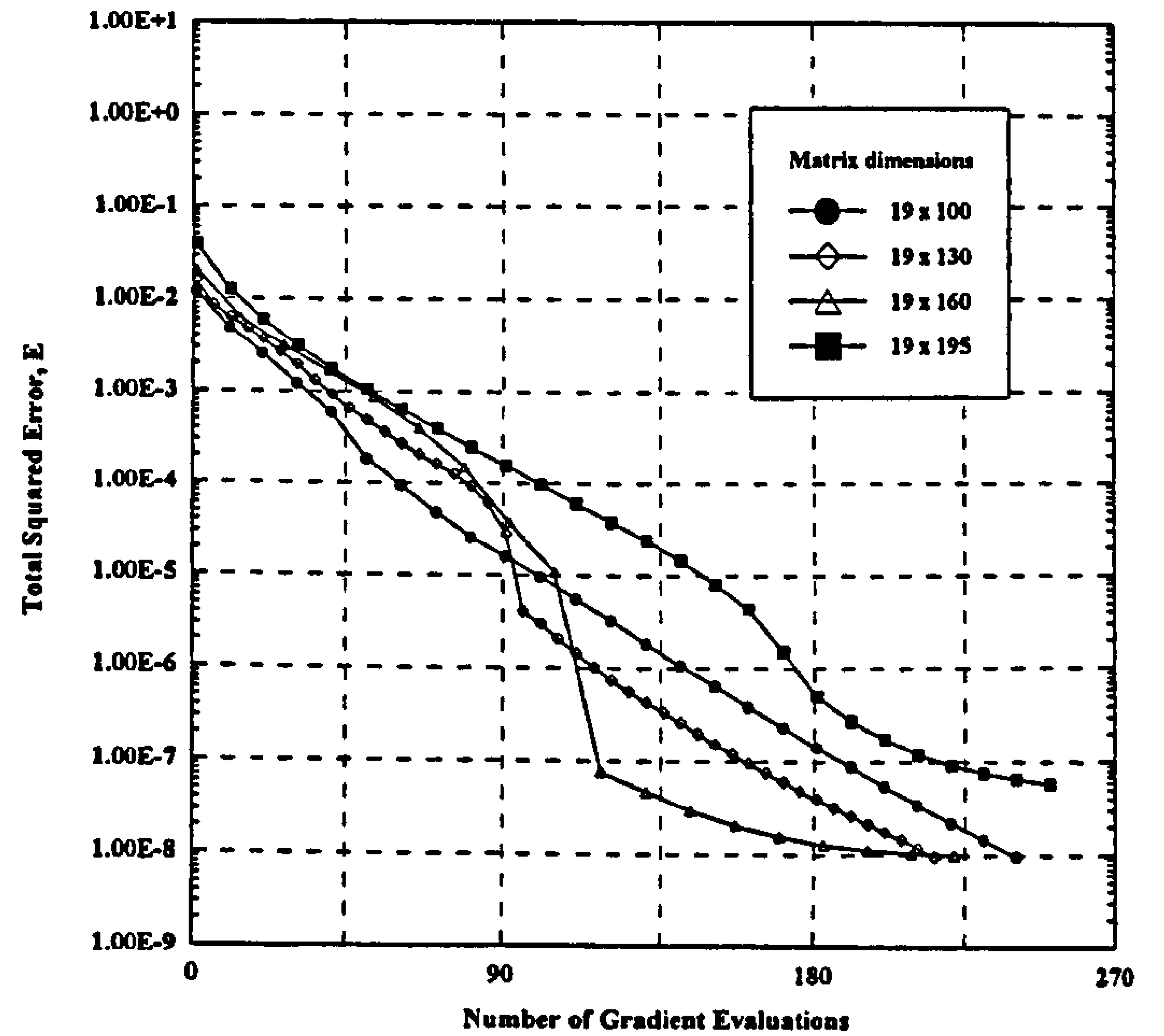


(d) Convergence history for Case 11.

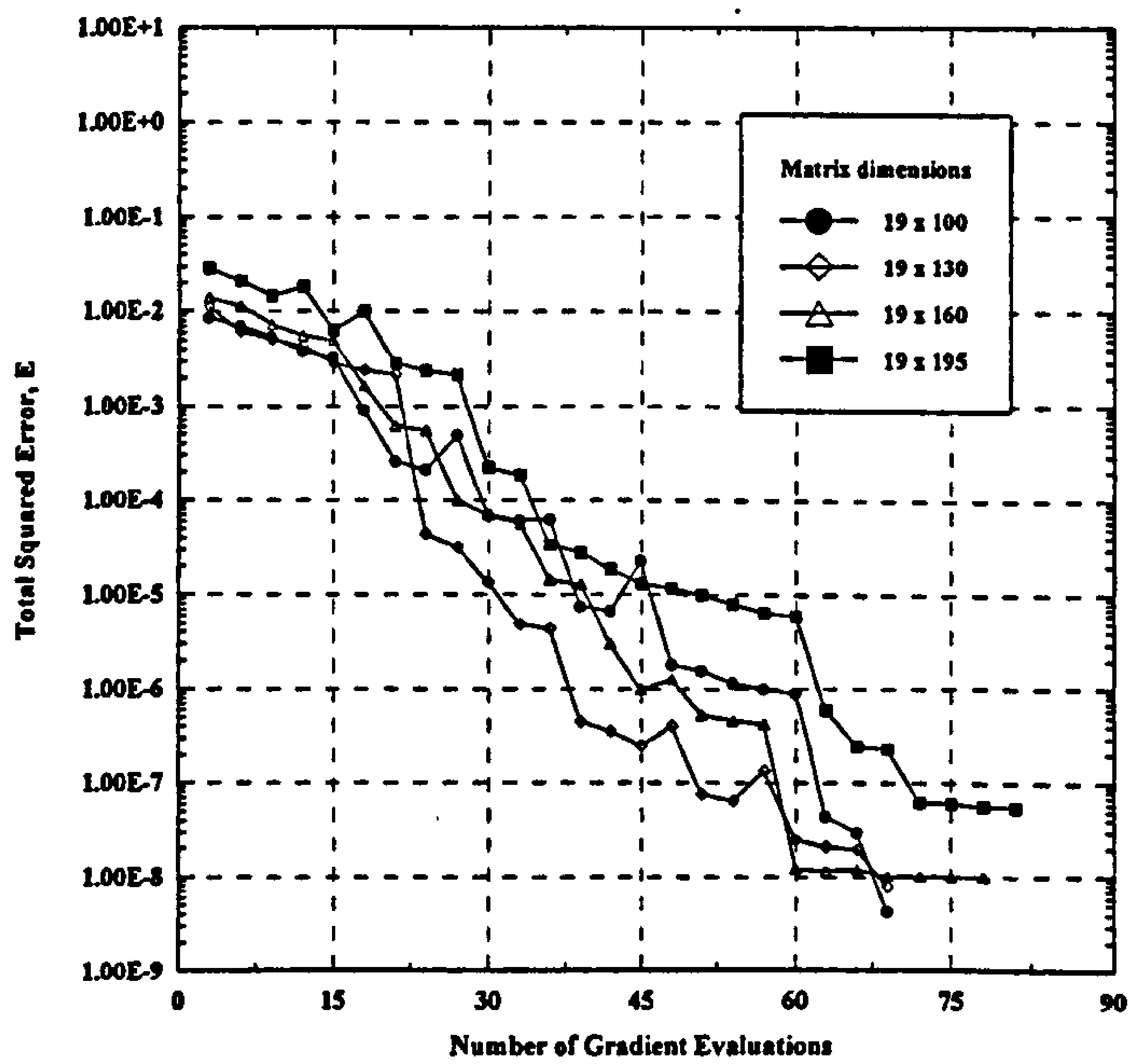
Figure 5.11 Performance of training methods as a function of time for the LU-decomposition of band matrices having different band-width.



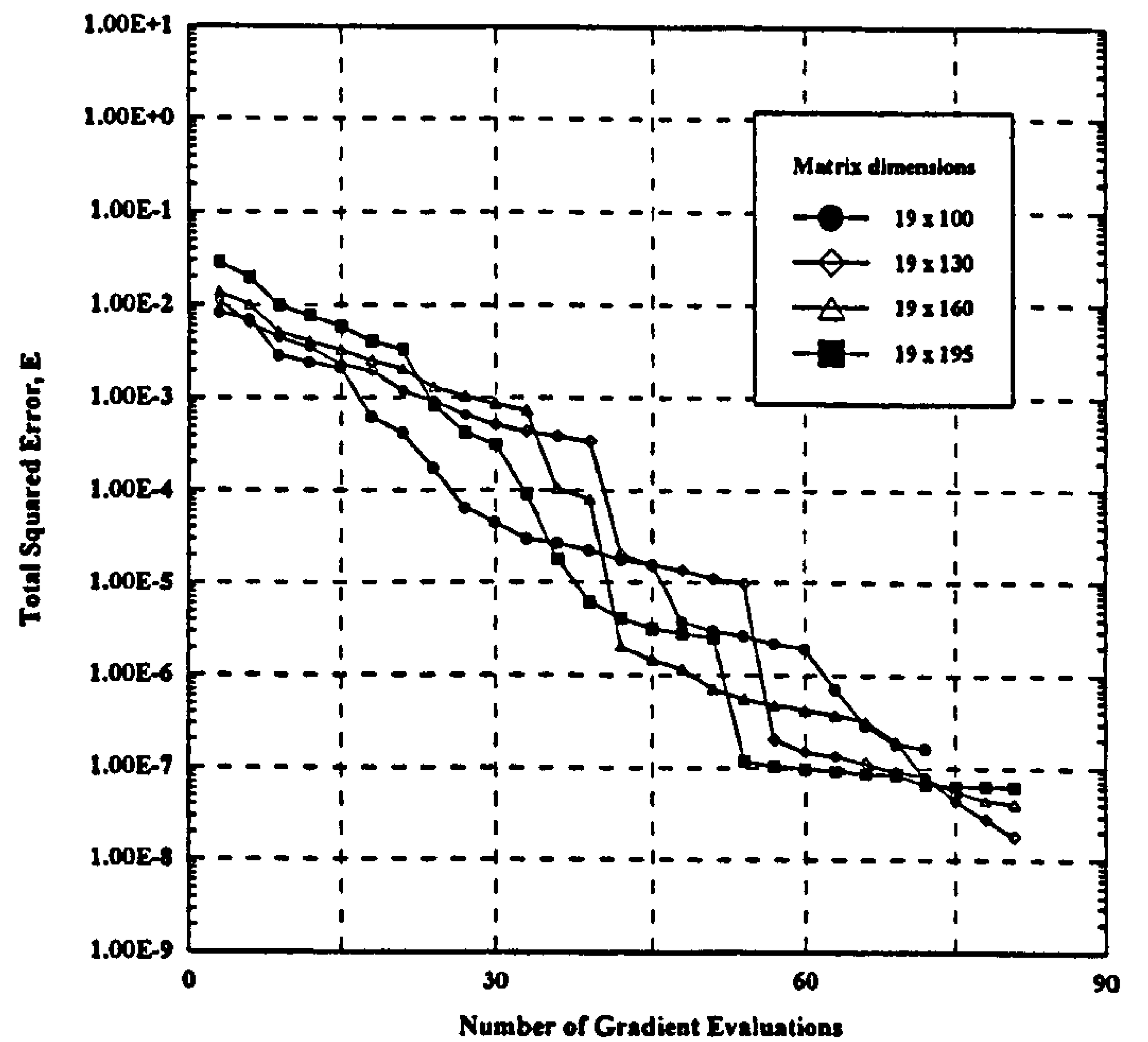
(a) Learning curves for BP method.



(b) Learning curves for SDLS method.



(c) Learning curves for CGFR method.



(d) Learning curves for CGPR method.

Figure 5.12 Effect of matrix size on the performance of four training methods used for the LU-decomposition of band matrices of same band-width.

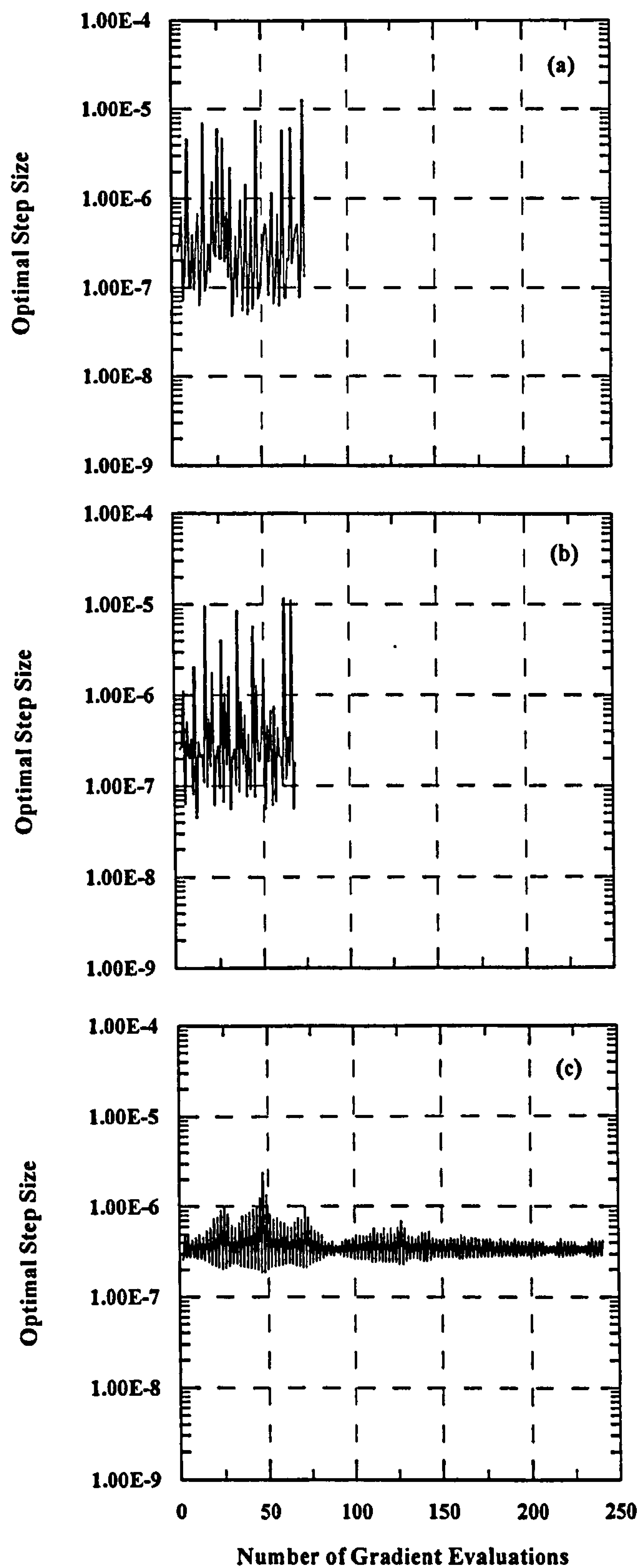


Figure 5.13 Illustration of optimal step size behavior for Case 5 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.

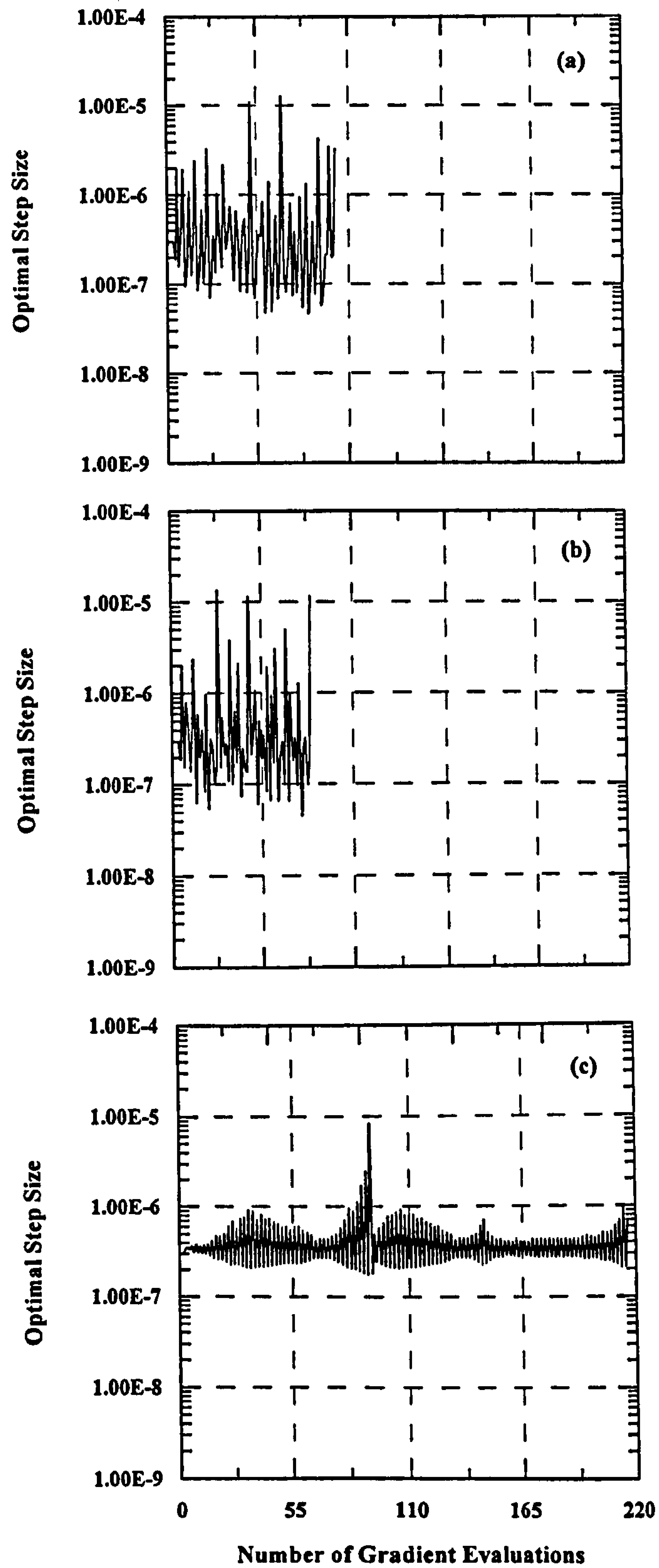


Figure 5.14 Illustration of optimal step size behavior for Case 6 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.

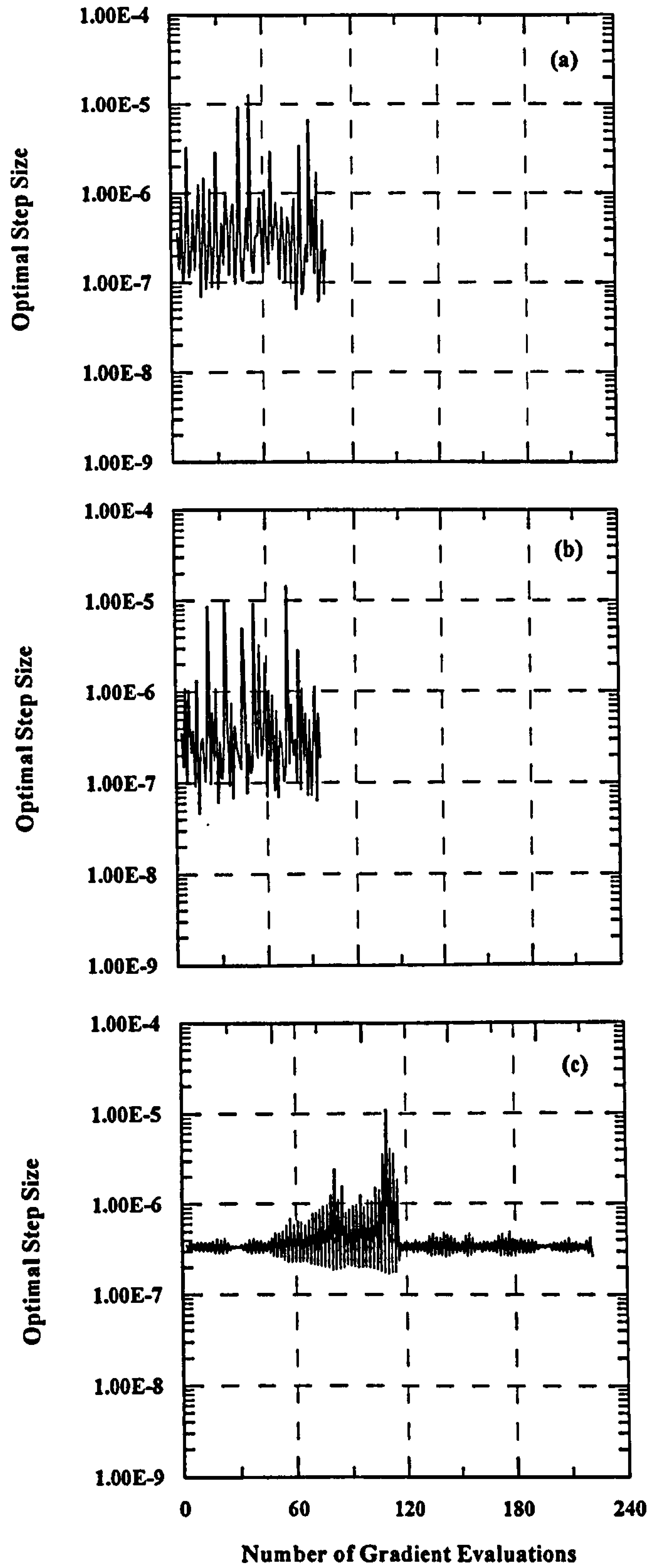


Figure 5.15 Illustration of optimal step size behavior for Case 7 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.

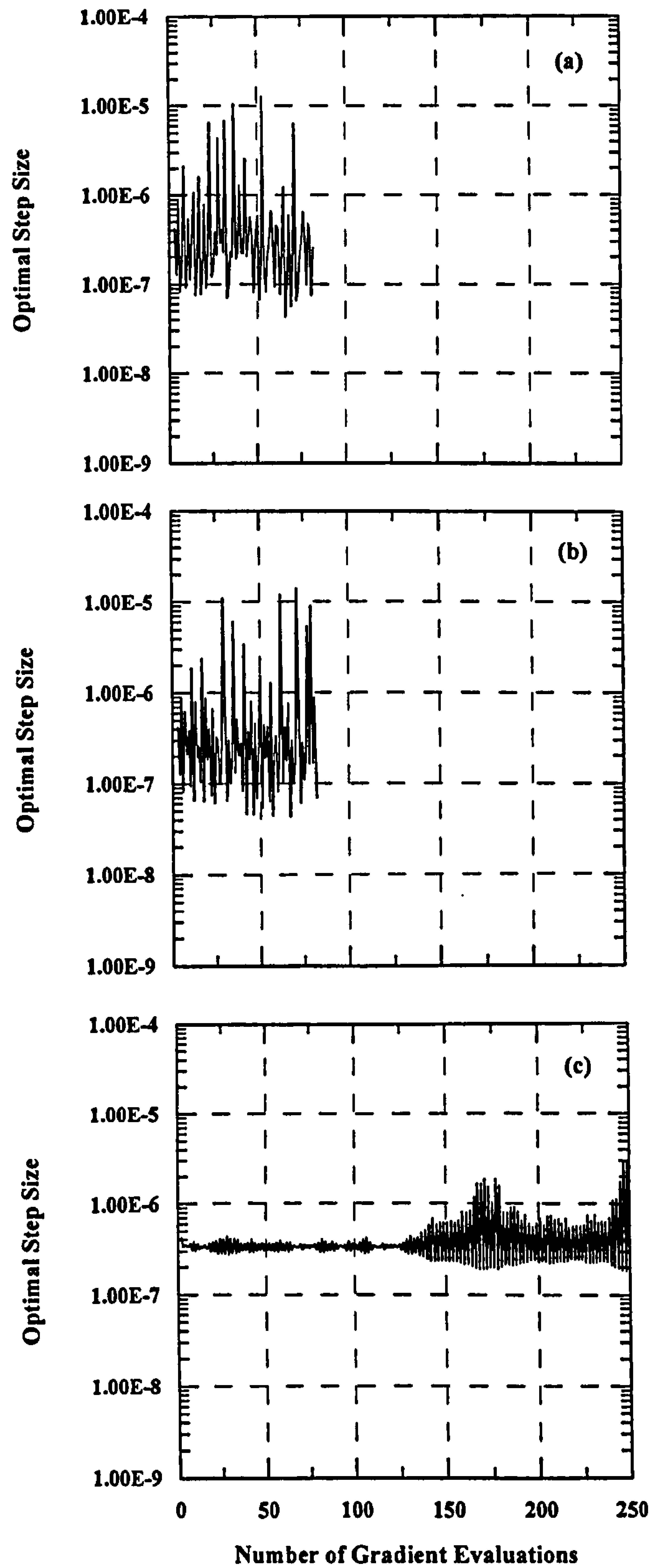
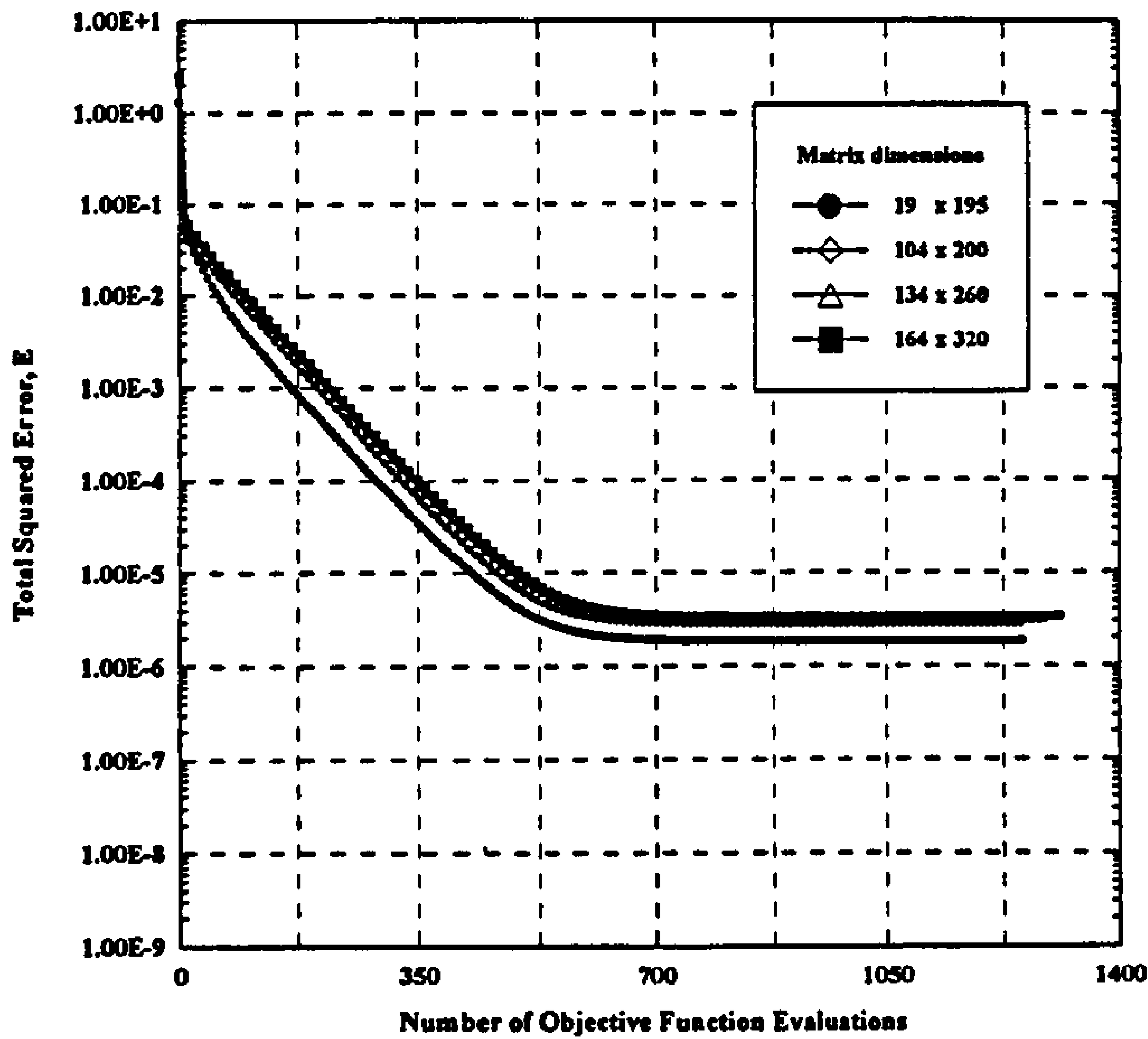
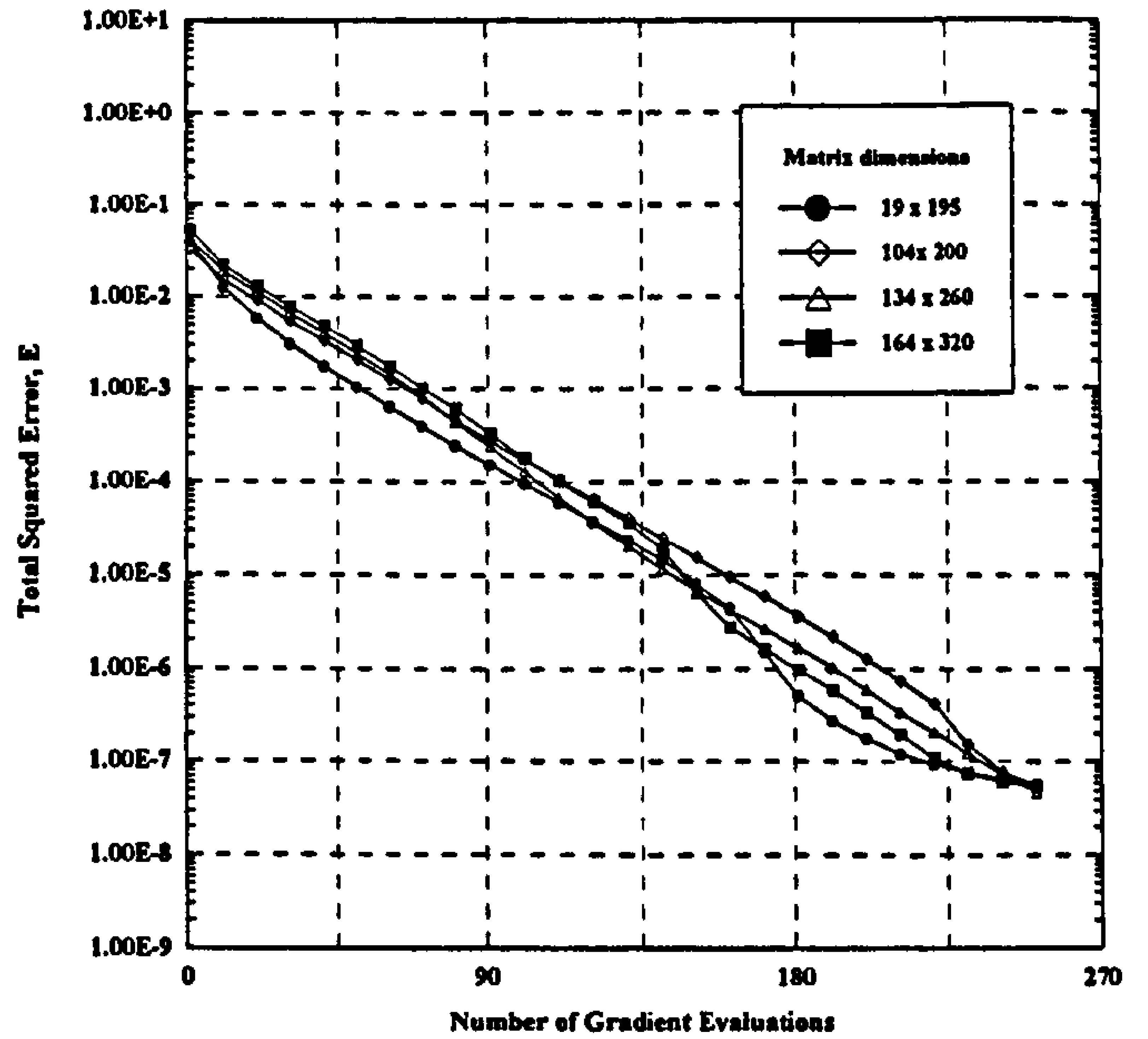


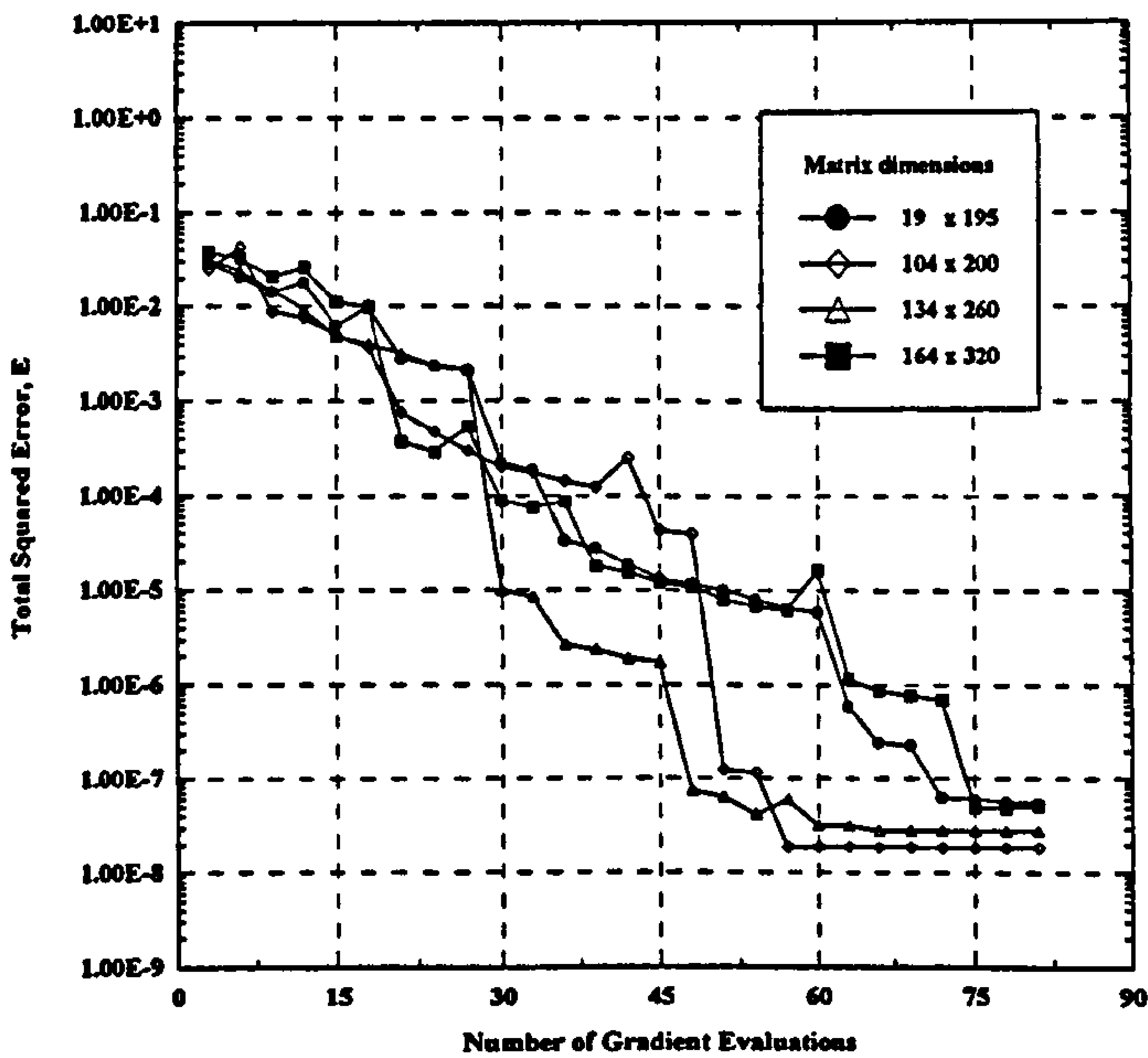
Figure 5.16 Illustration of optimal step size behavior for Case 8 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.



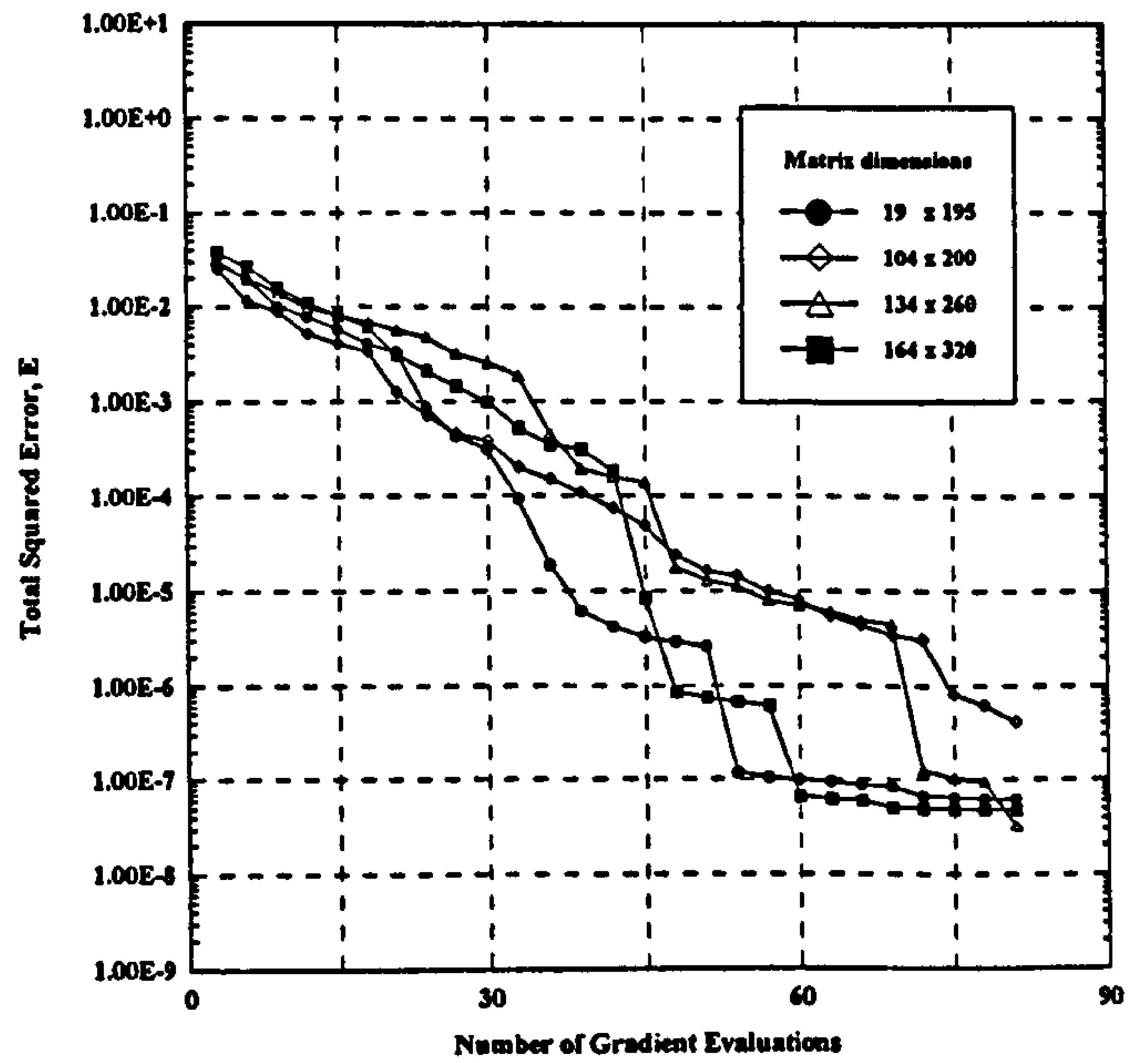
(a) Learning curves for BP method.



(b) Learning curves for SDLS method.



(c) Learning curves for CGFR method.



(d) Learning curves for CGPR method.

Figure 5.17 Effect of matrix size of the performance of training methods used for the LU-decomposition of band matrices of different bandwidth.

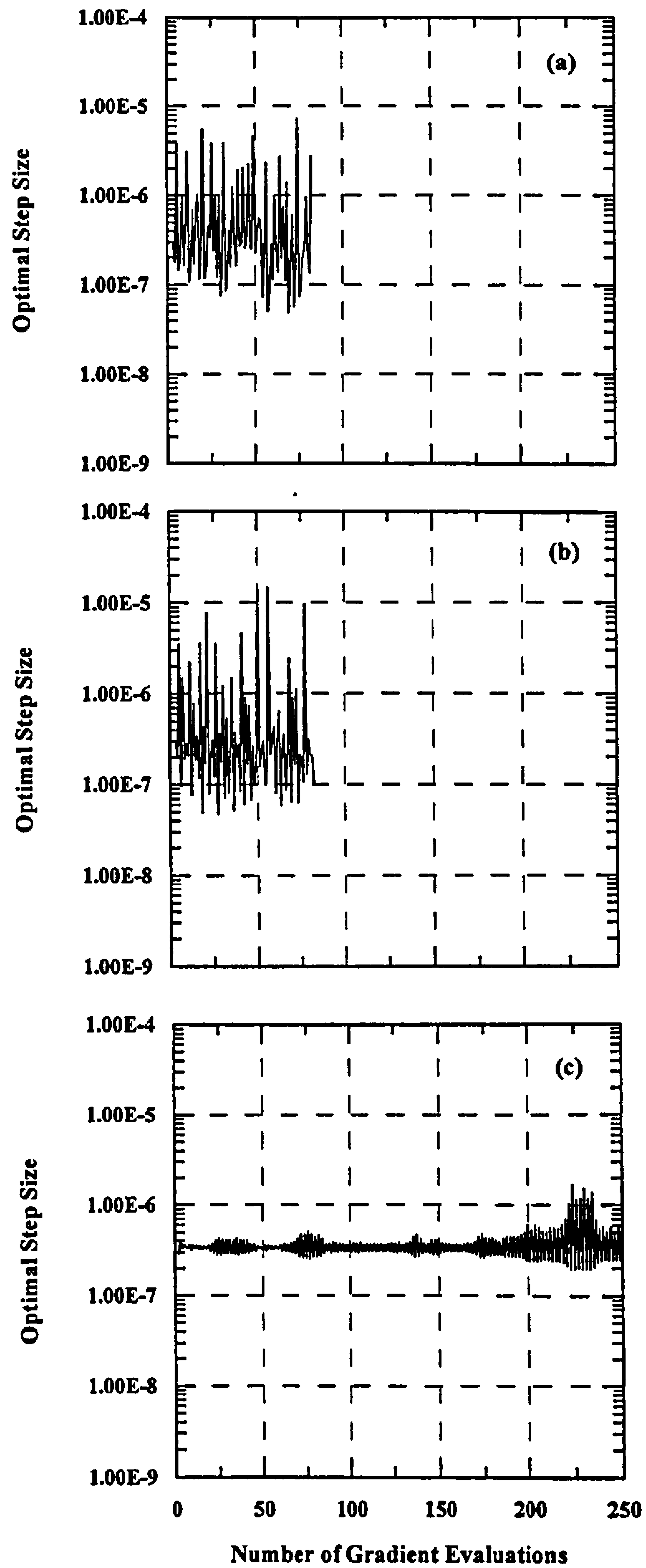


Figure 5.18 Illustration of optimal step size behavior for Case 9 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.

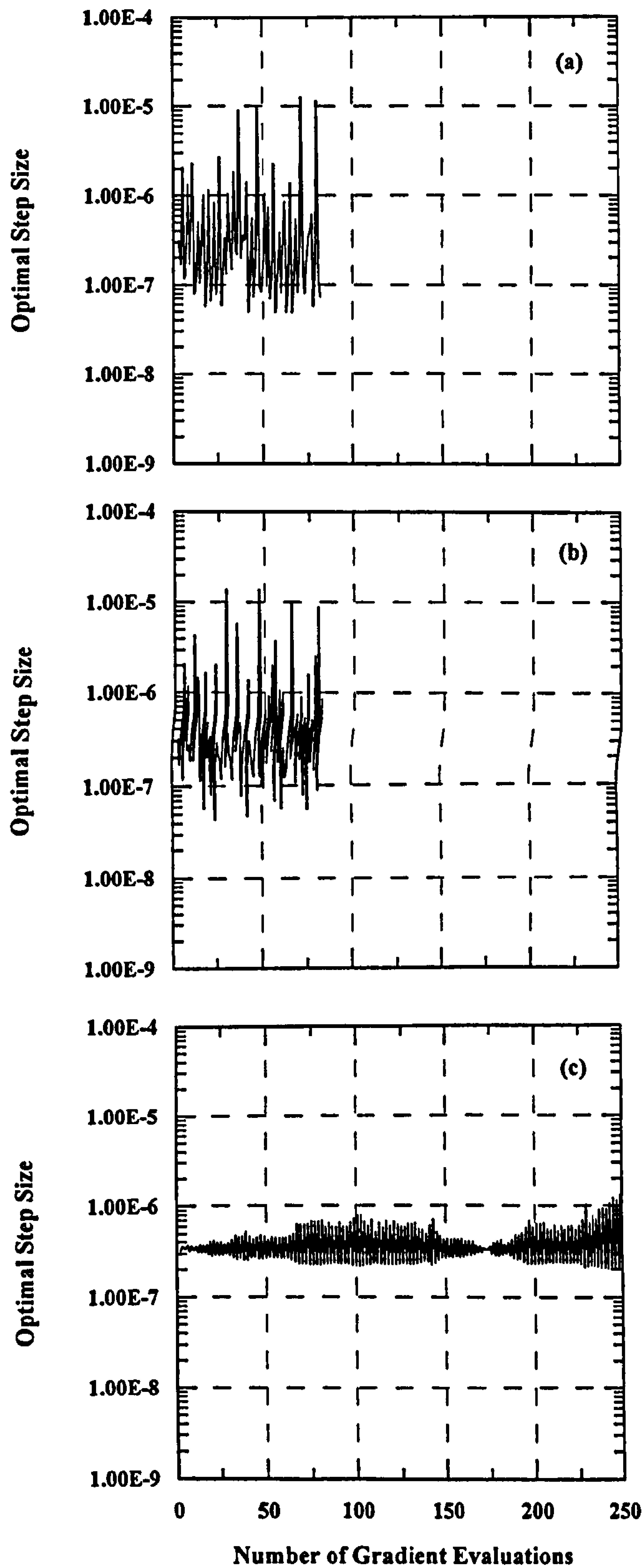


Figure 5.19 Illustration of optimal step size behavior for Case 10 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.

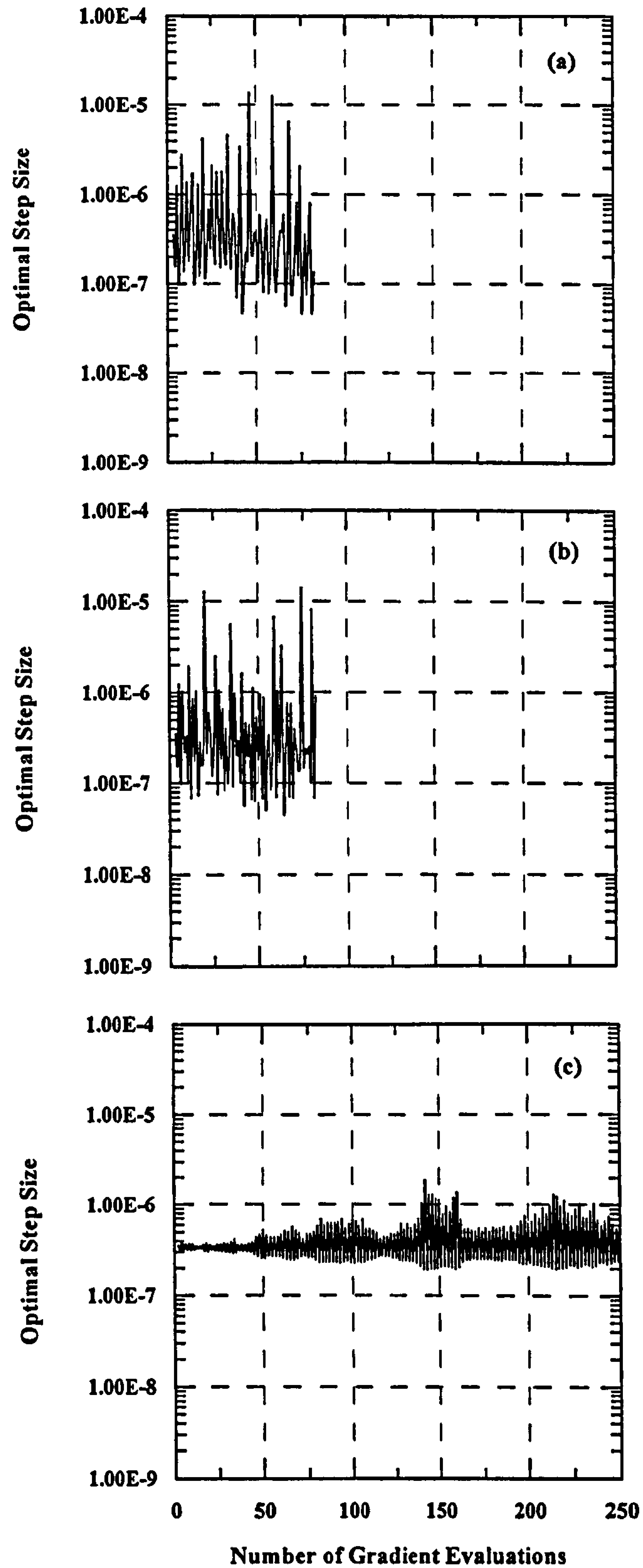
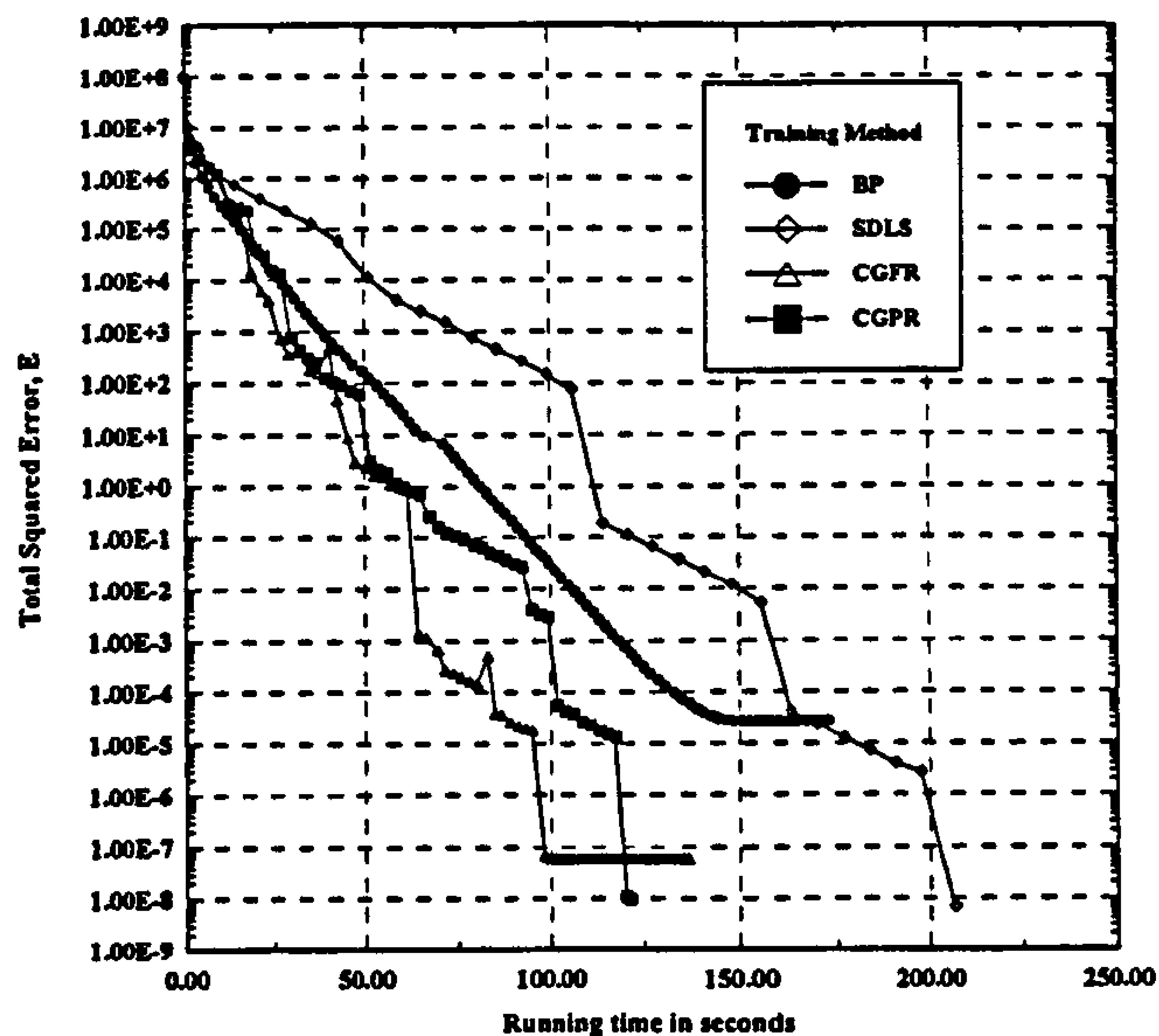
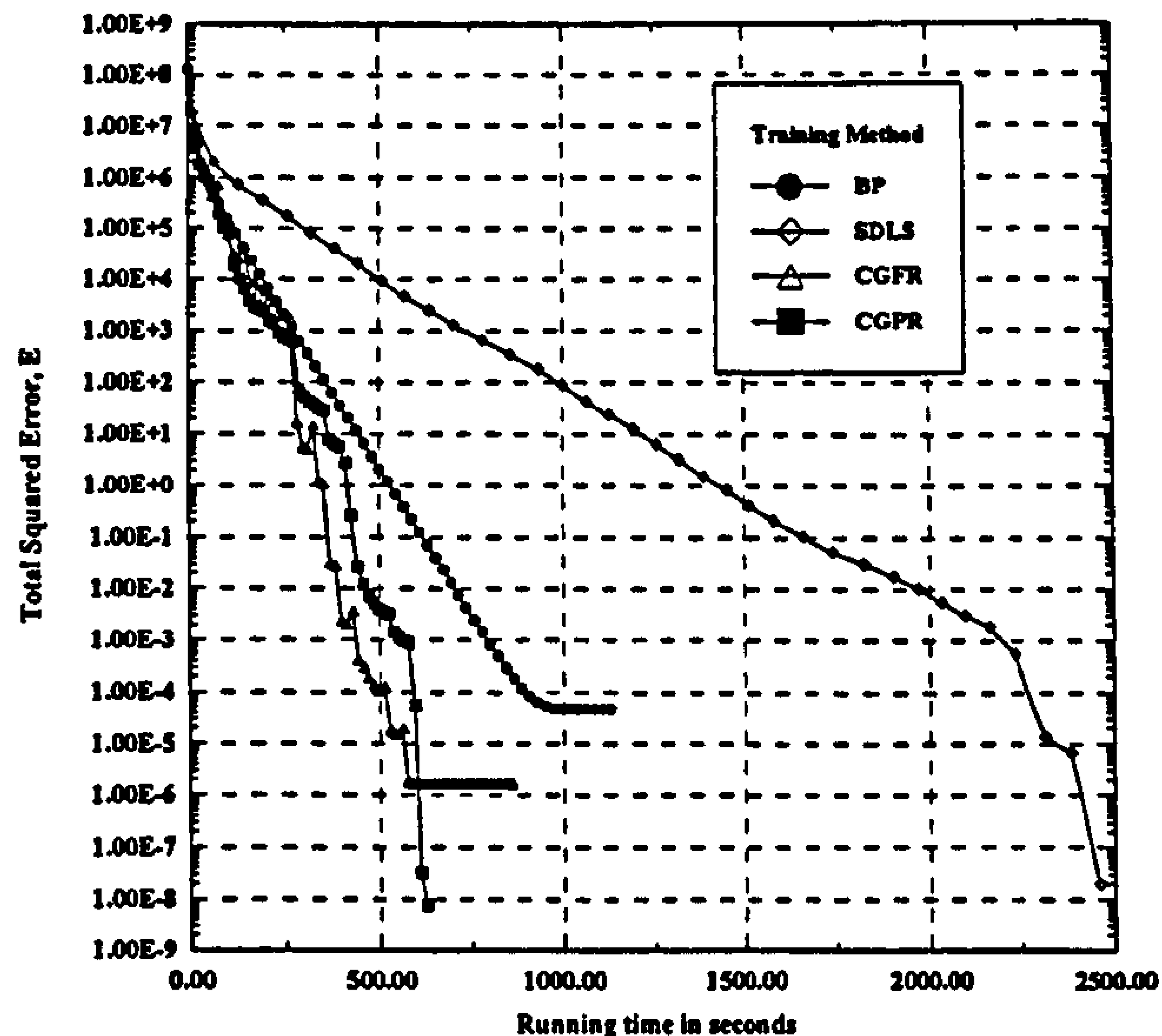


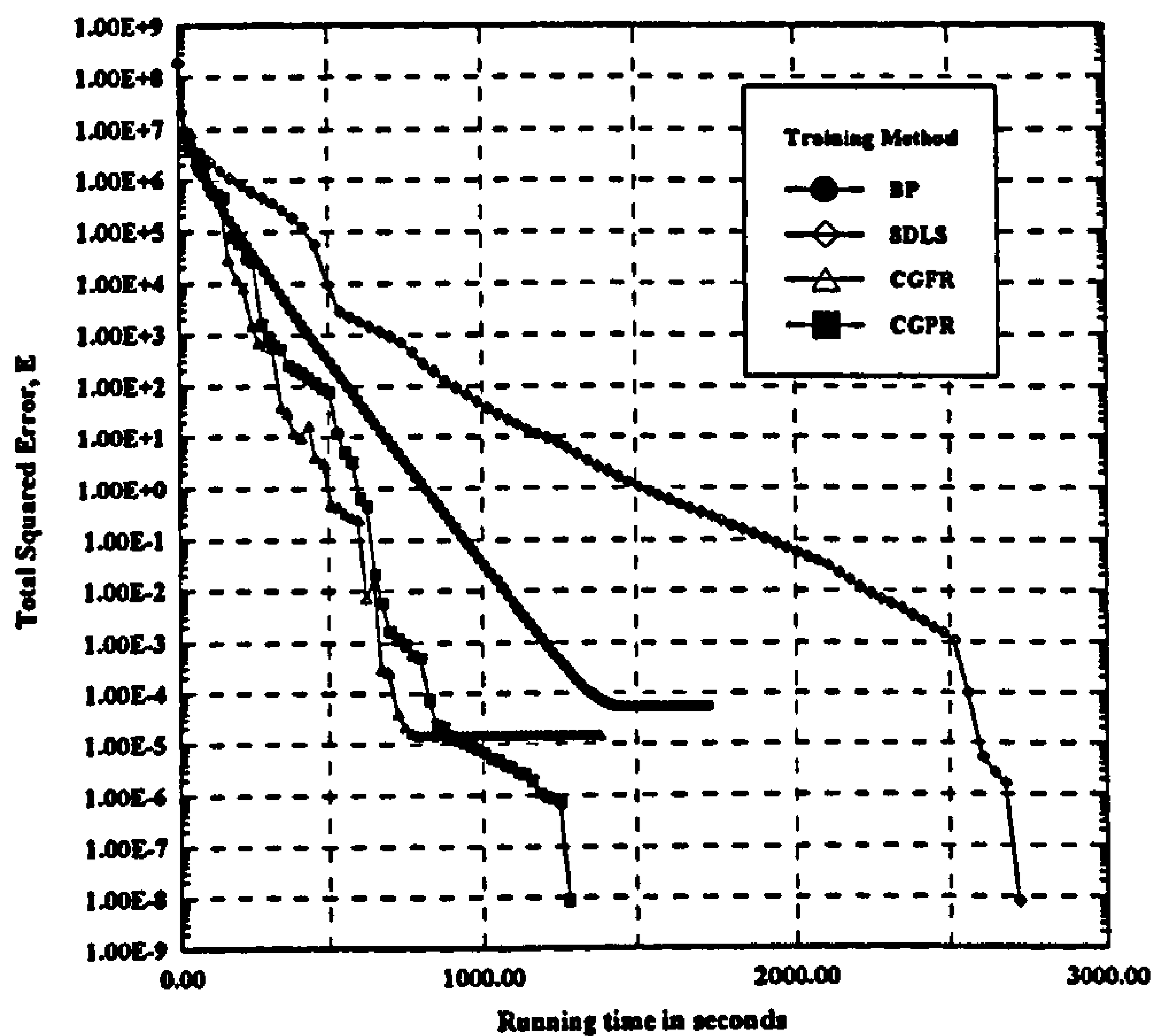
Figure 5.20 Illustration of optimal step size behavior for Case 11 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.



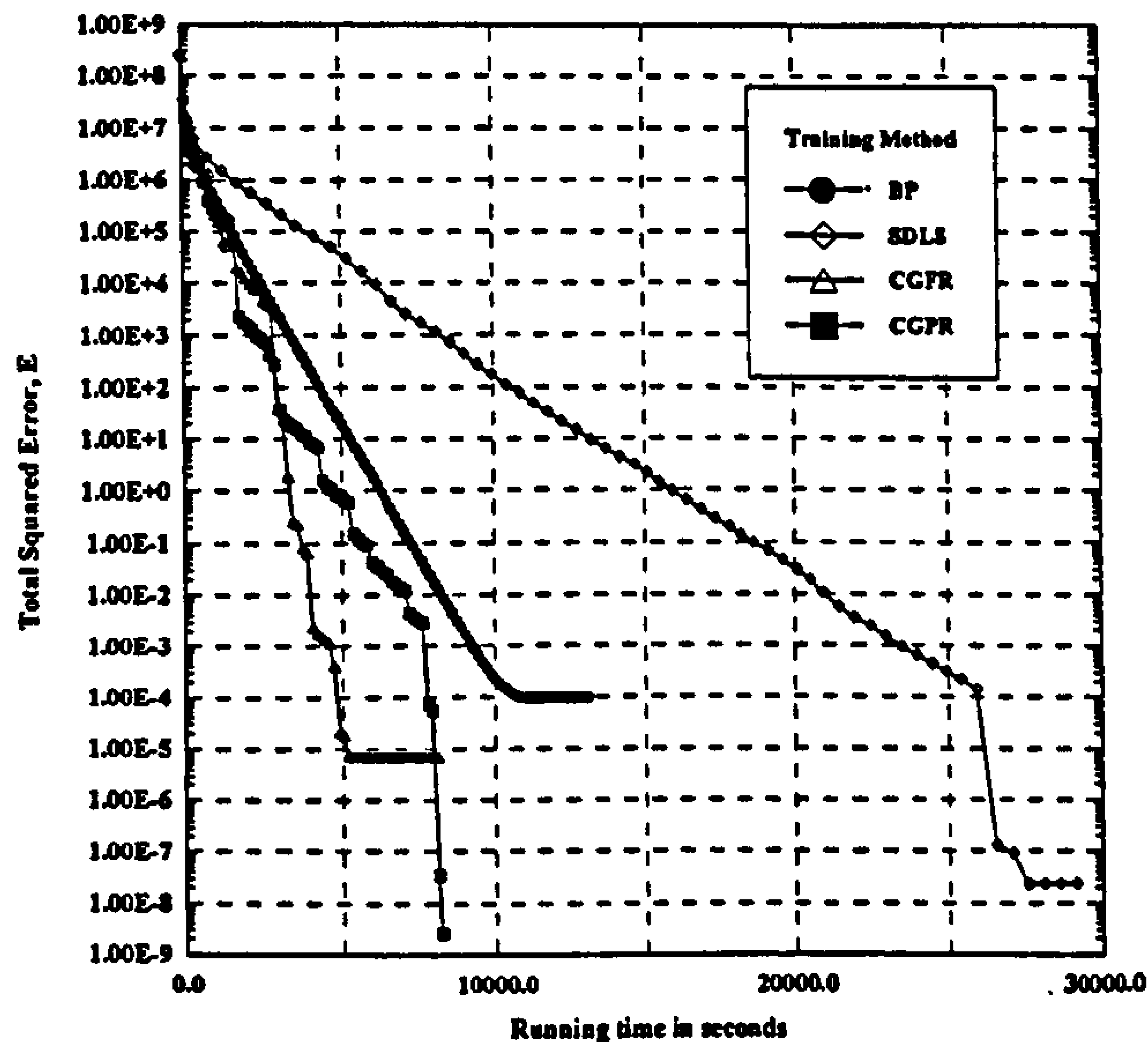
(a) Convergence history for Case 12.



(b) Convergence history for Case 14.



(c) Convergence history for Case 15.



(d) Convergence history for Case 17.

Figure 5.21 Performance of training methods as a function of time for the inversion of square matrices having different dimansions.

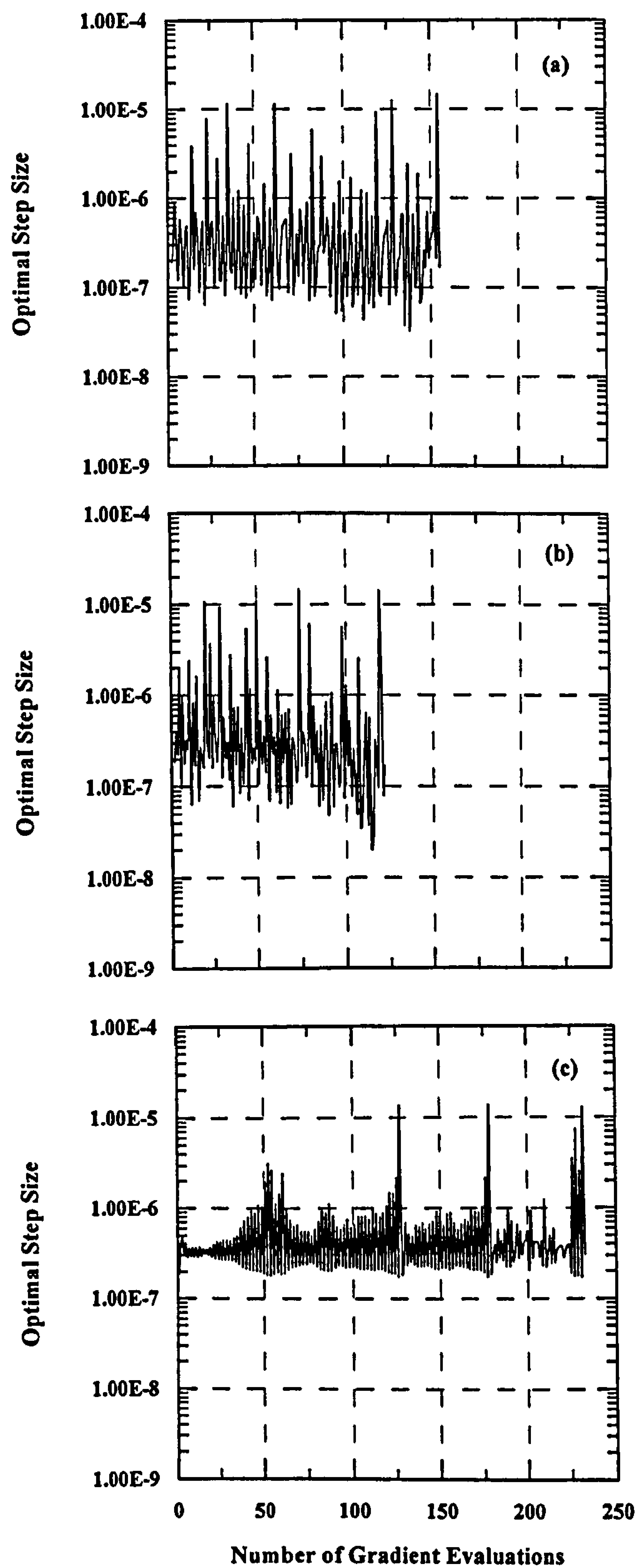


Figure 5.22 Illustration of optimal step size behavior for Case 12 under different training methods:(a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.

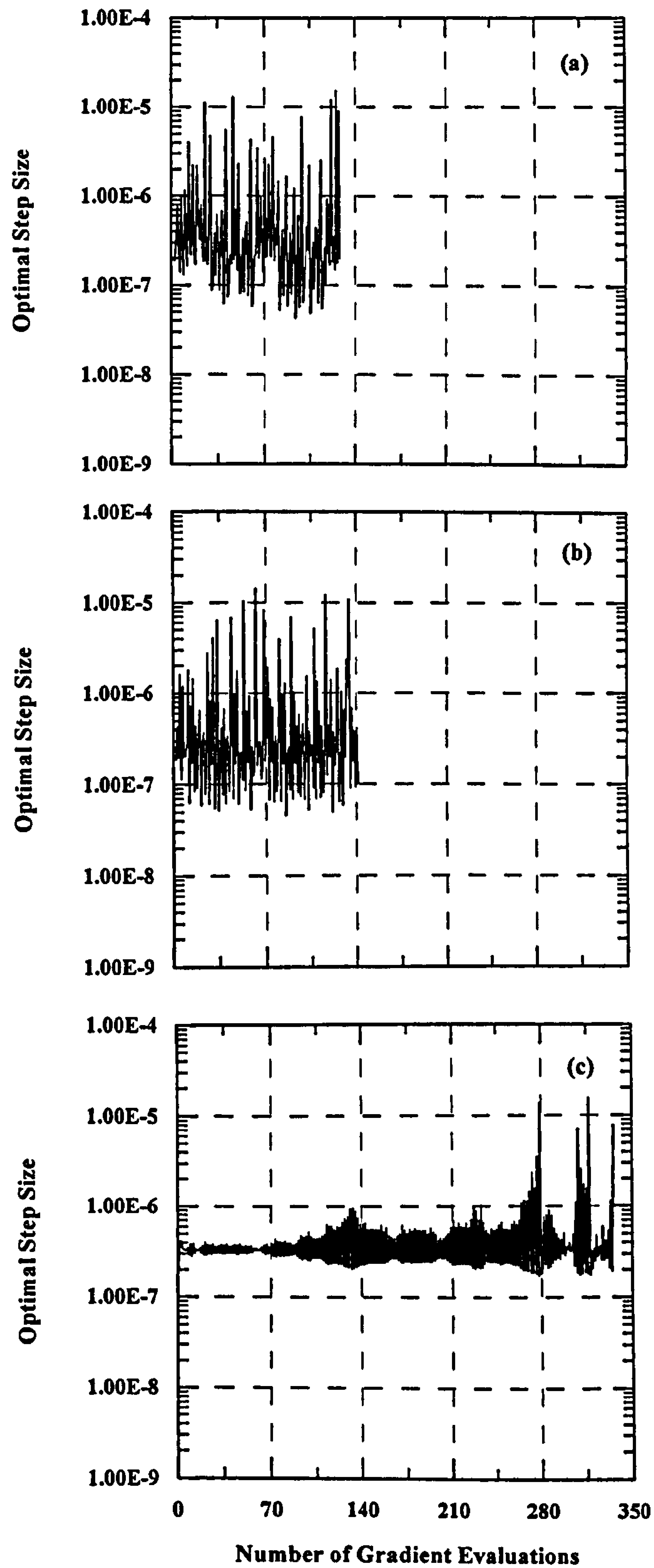


Figure 5.23 Illustration of optimal step size behavior for Case 13 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.

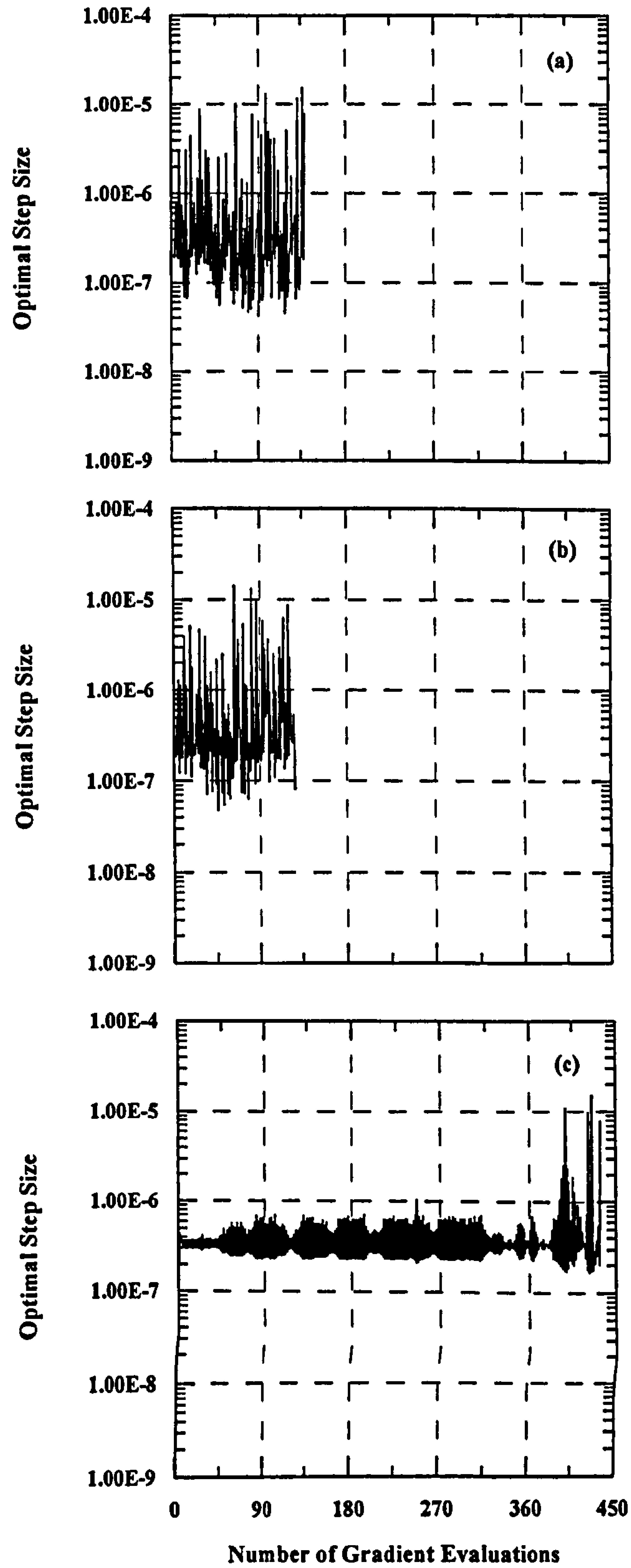


Figure 5.24 Illustration of optimal step size behavior for Case 14 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.

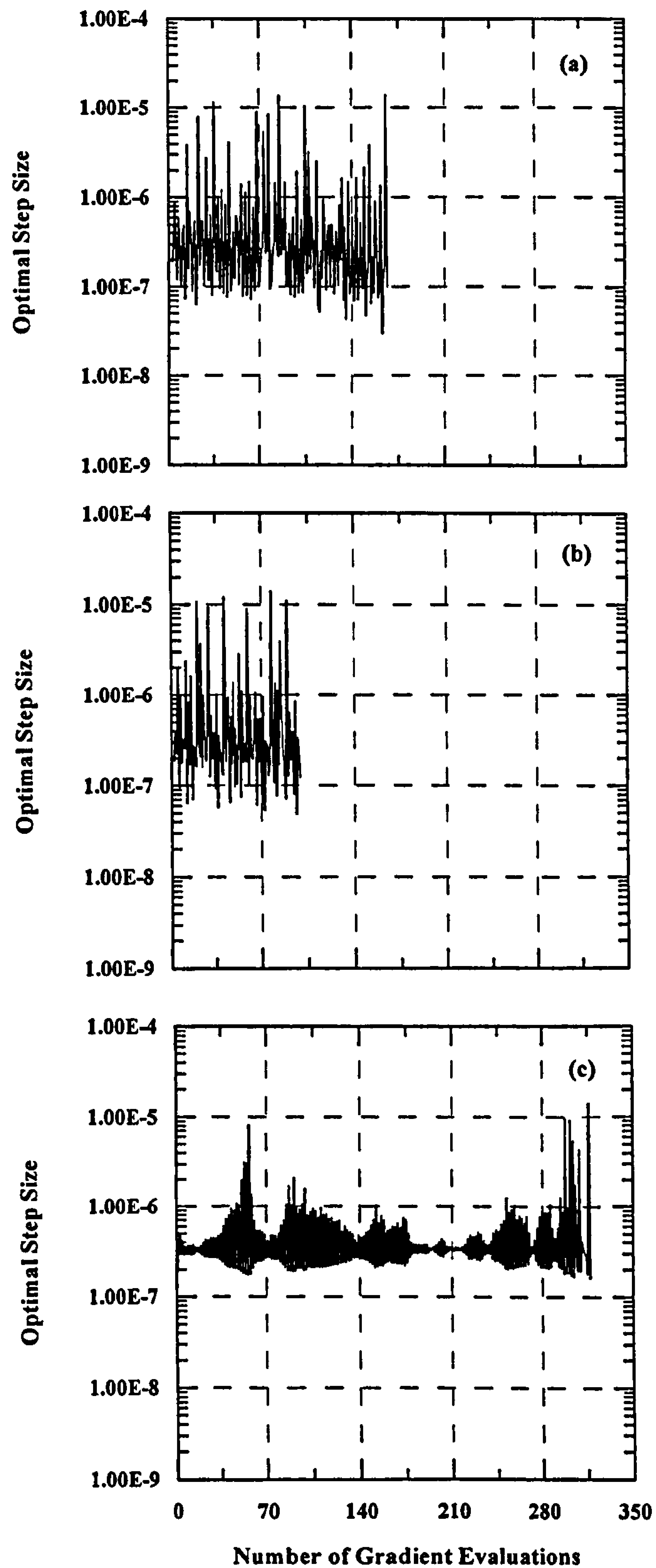


Figure 5.25 Illustration of optimal step size behavior for Case 15 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.

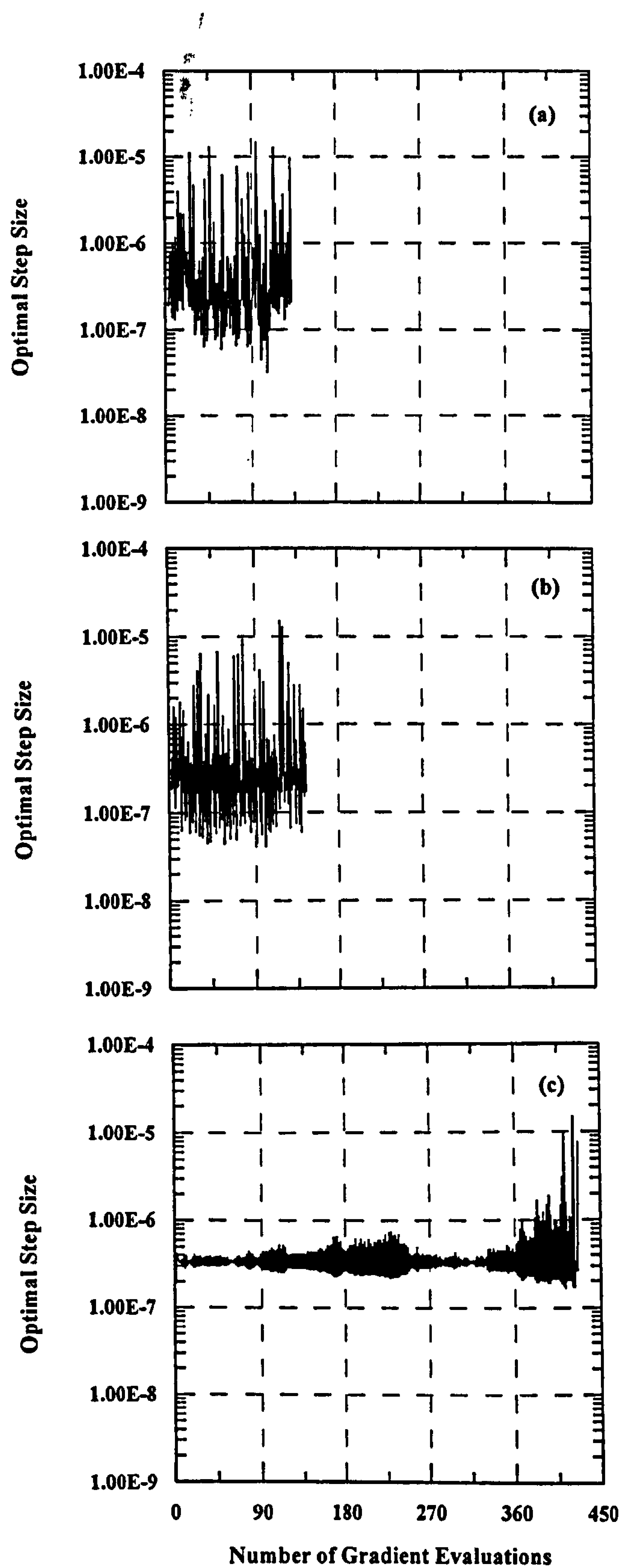


Figure 5.26 Illustration of optimal step size behavior for Case 16 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.

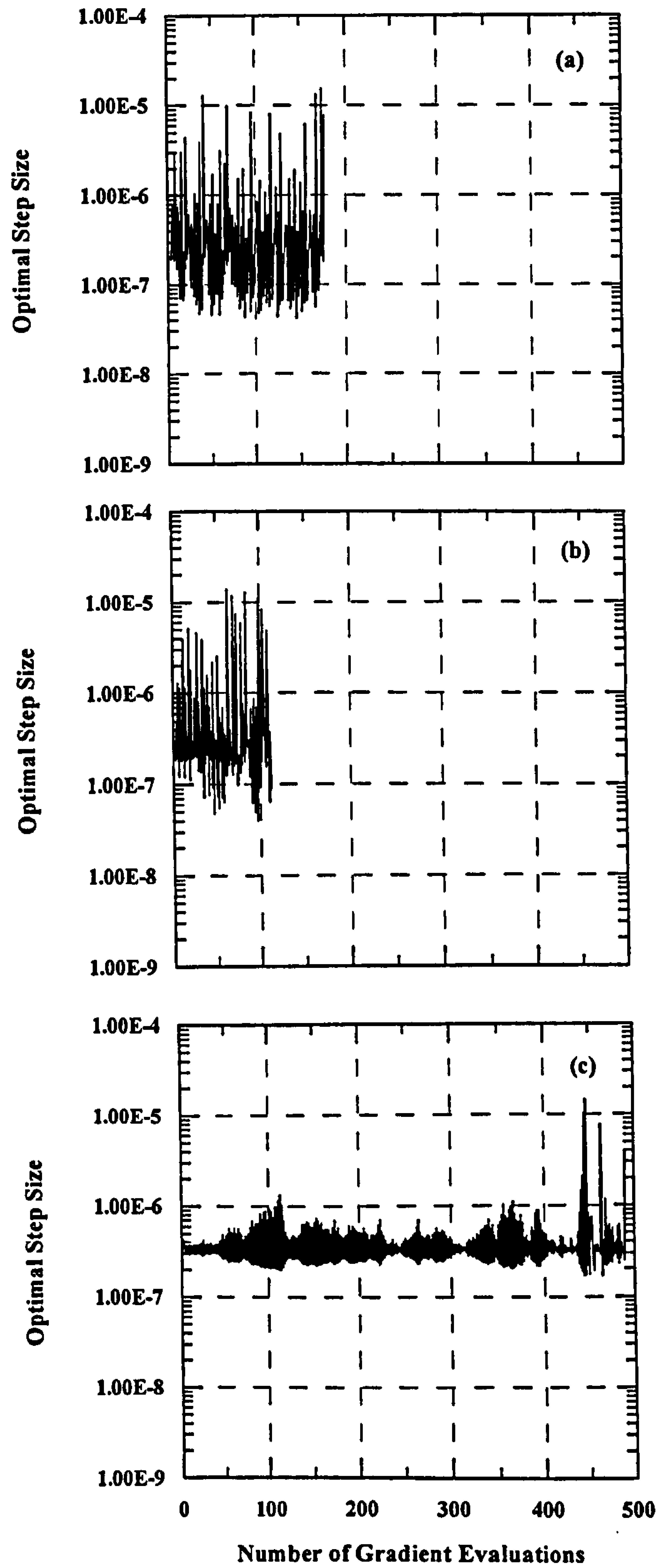
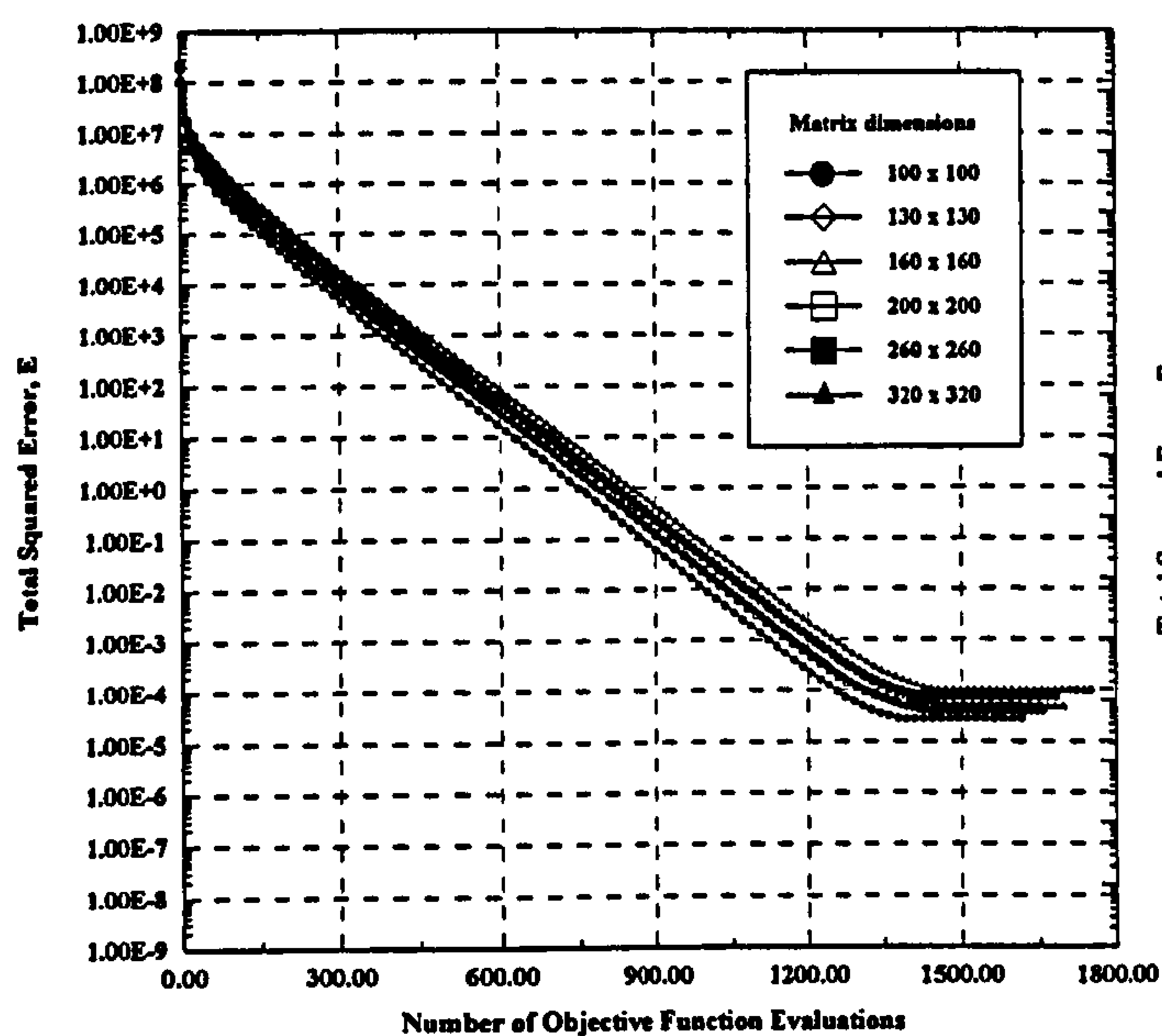
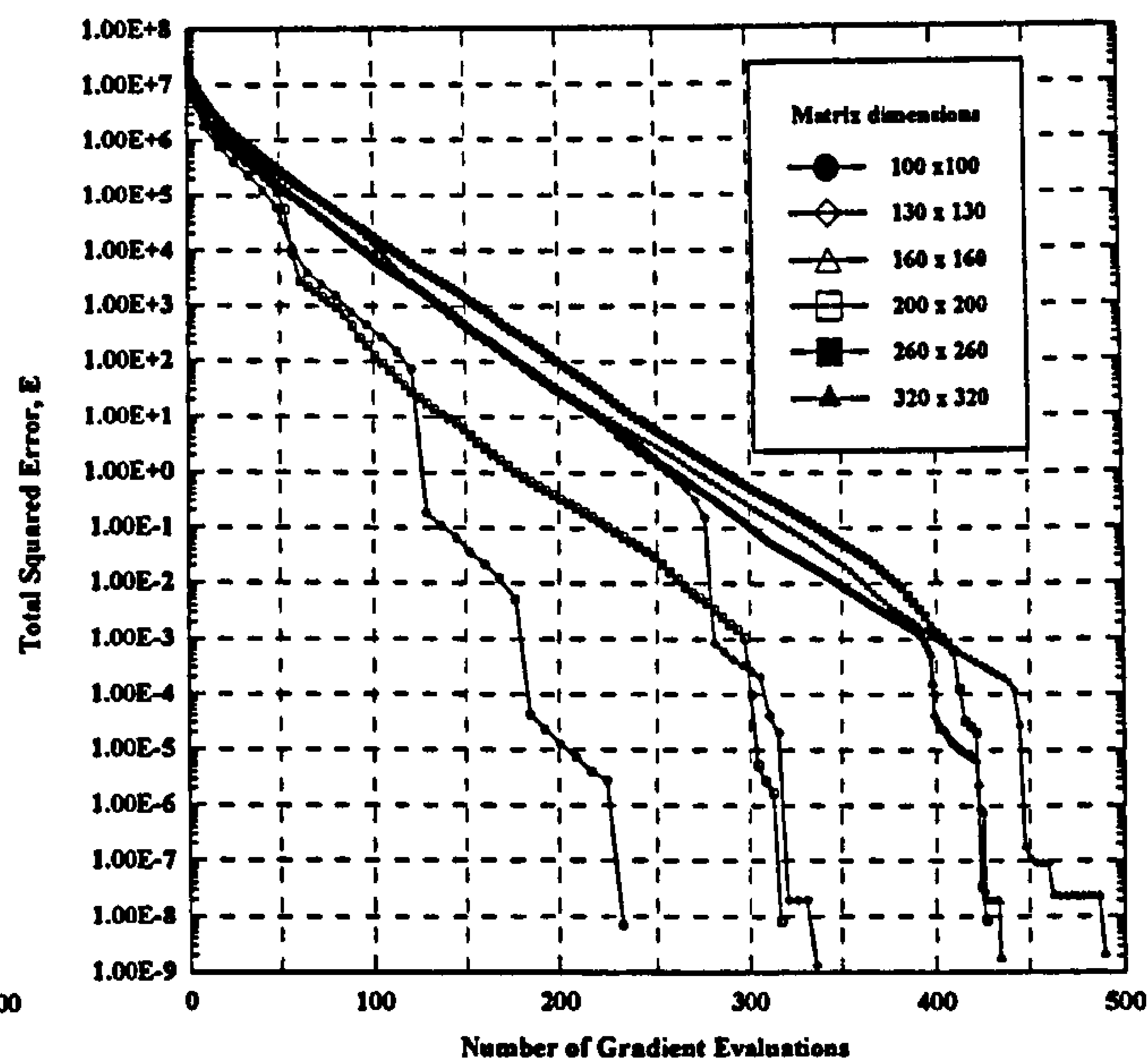


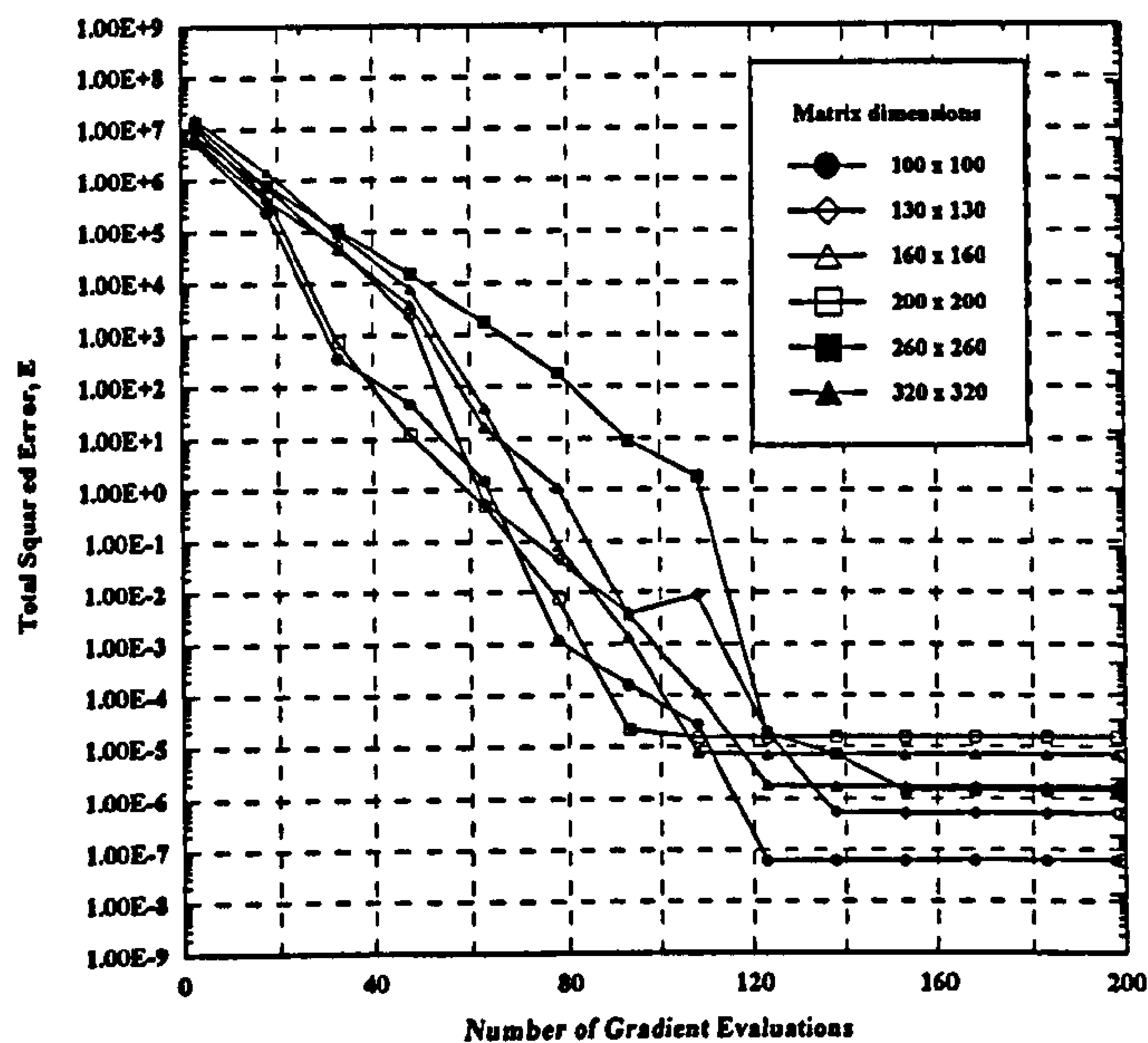
Figure 5.27 Illustration of optimal step size behavior for Case 17 under different training methods: (a) CGPR (r = 3); (b) CGFR (r = 3); and (c) SDLS.



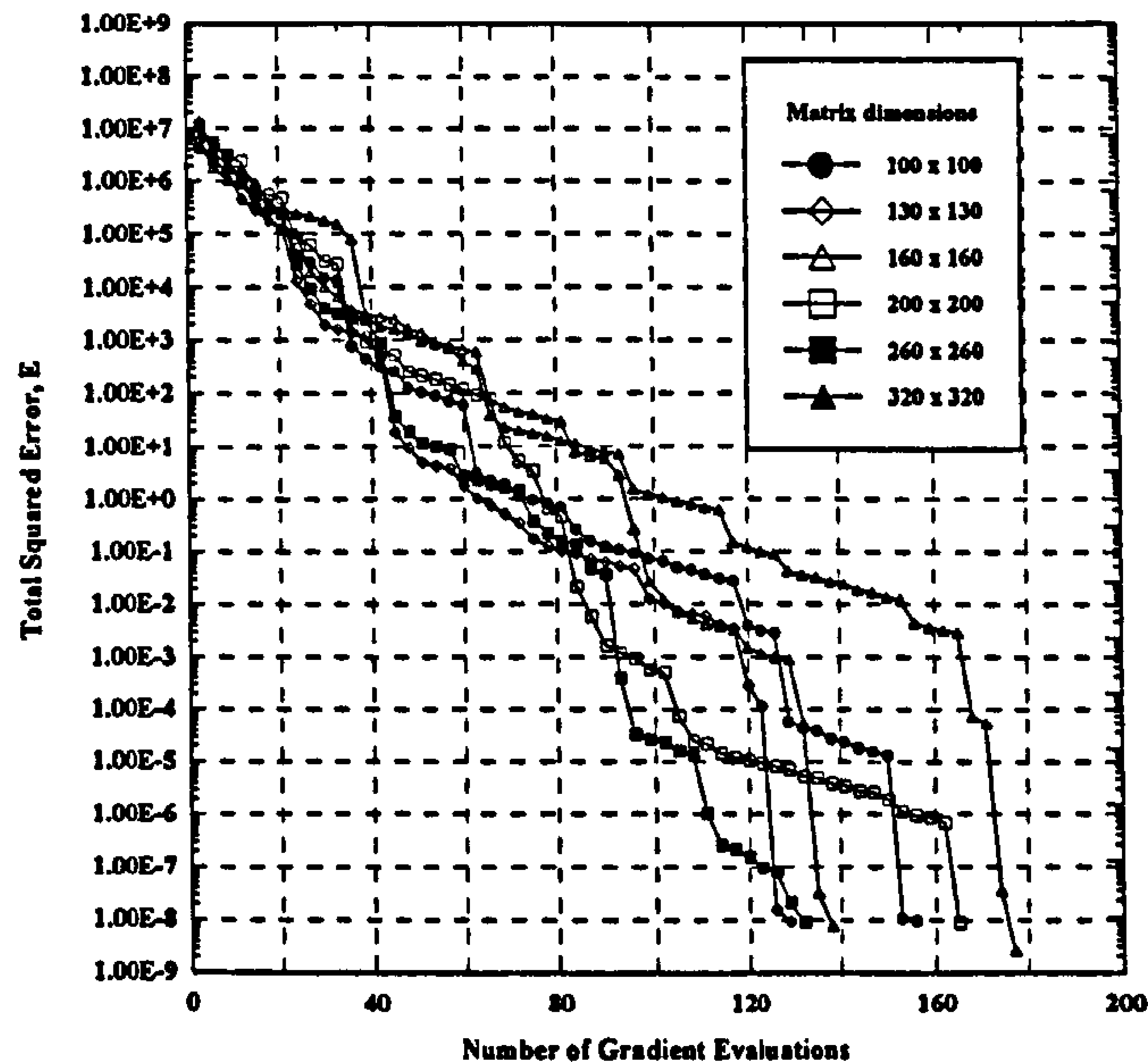
(a) Learning curves for BP method.



(b) Learning curves for SDLS method.

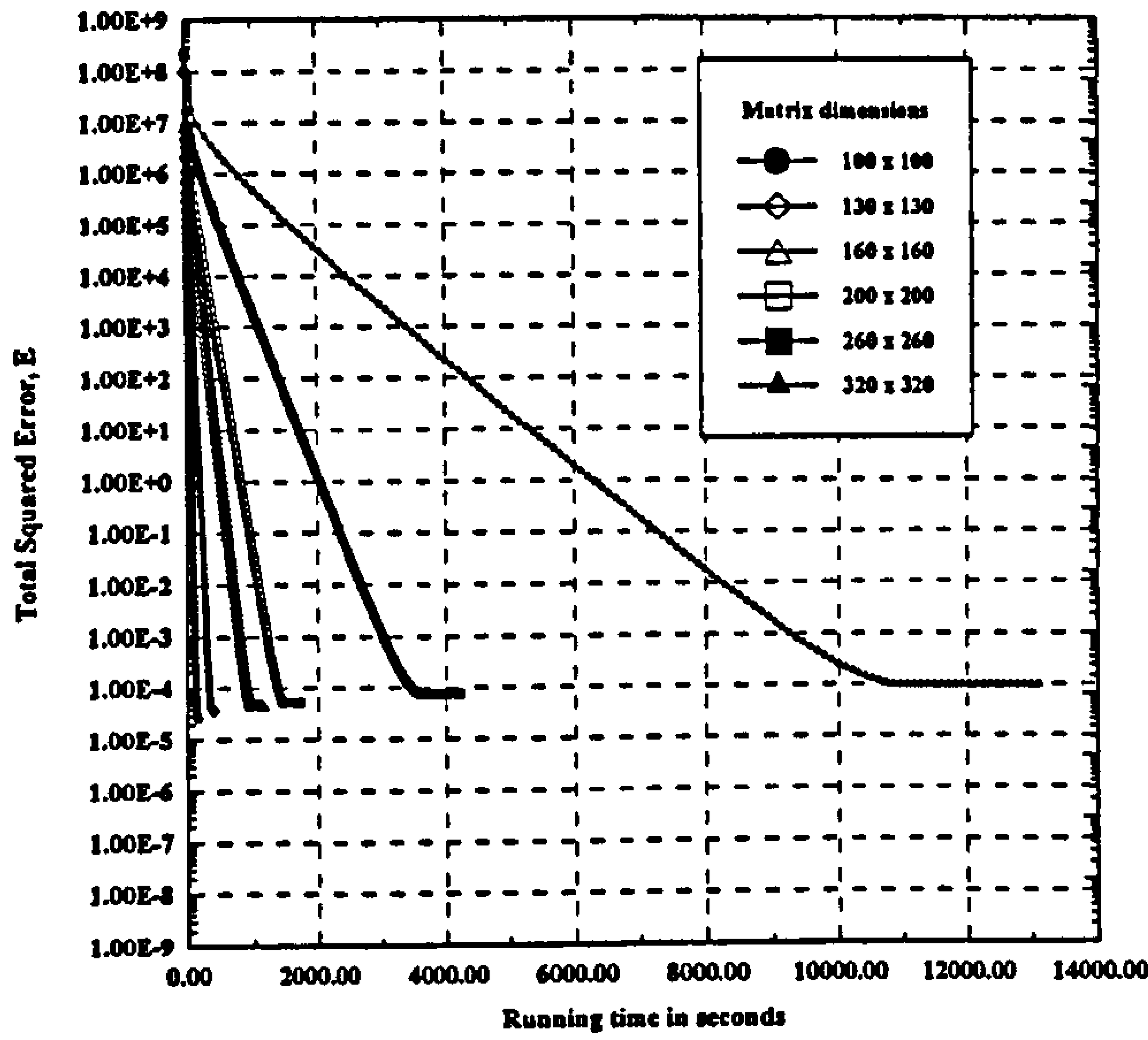


(c) Learning curves for CGFR method.

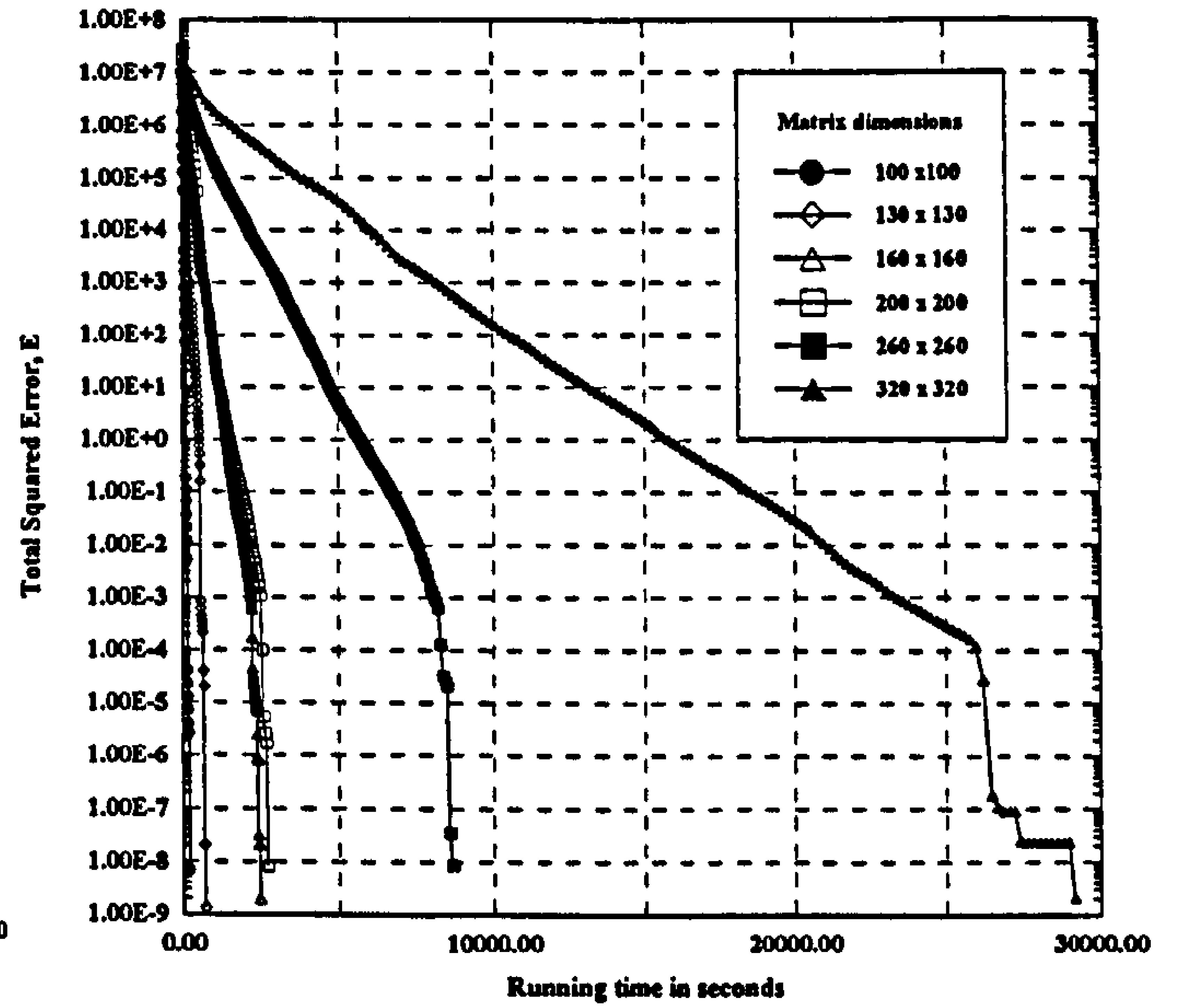


(d) Learning curves for CGPR method.

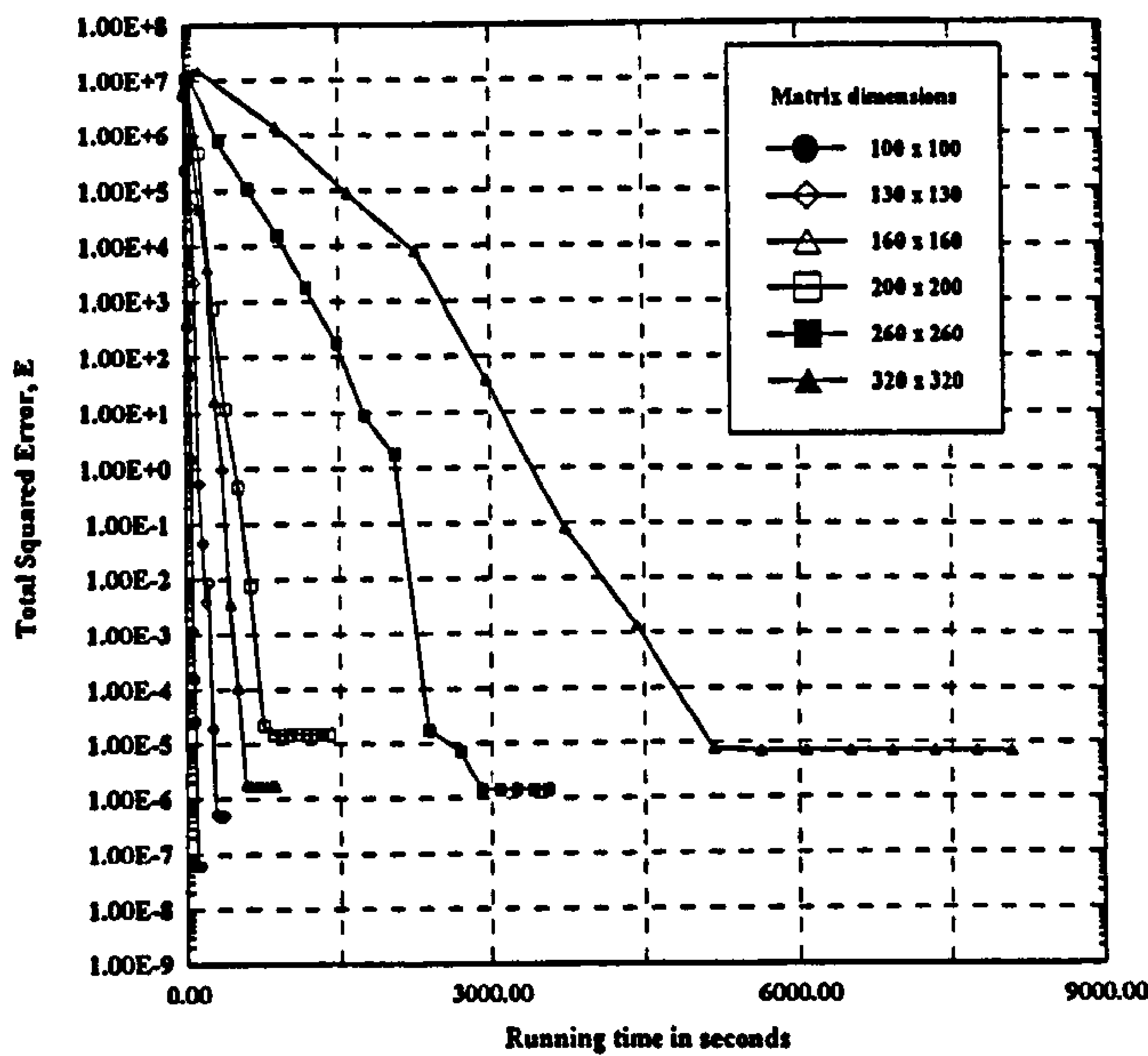
Figure 5.28 Effect of matrix size on the performance of training methods used for the inversion of square matrices.



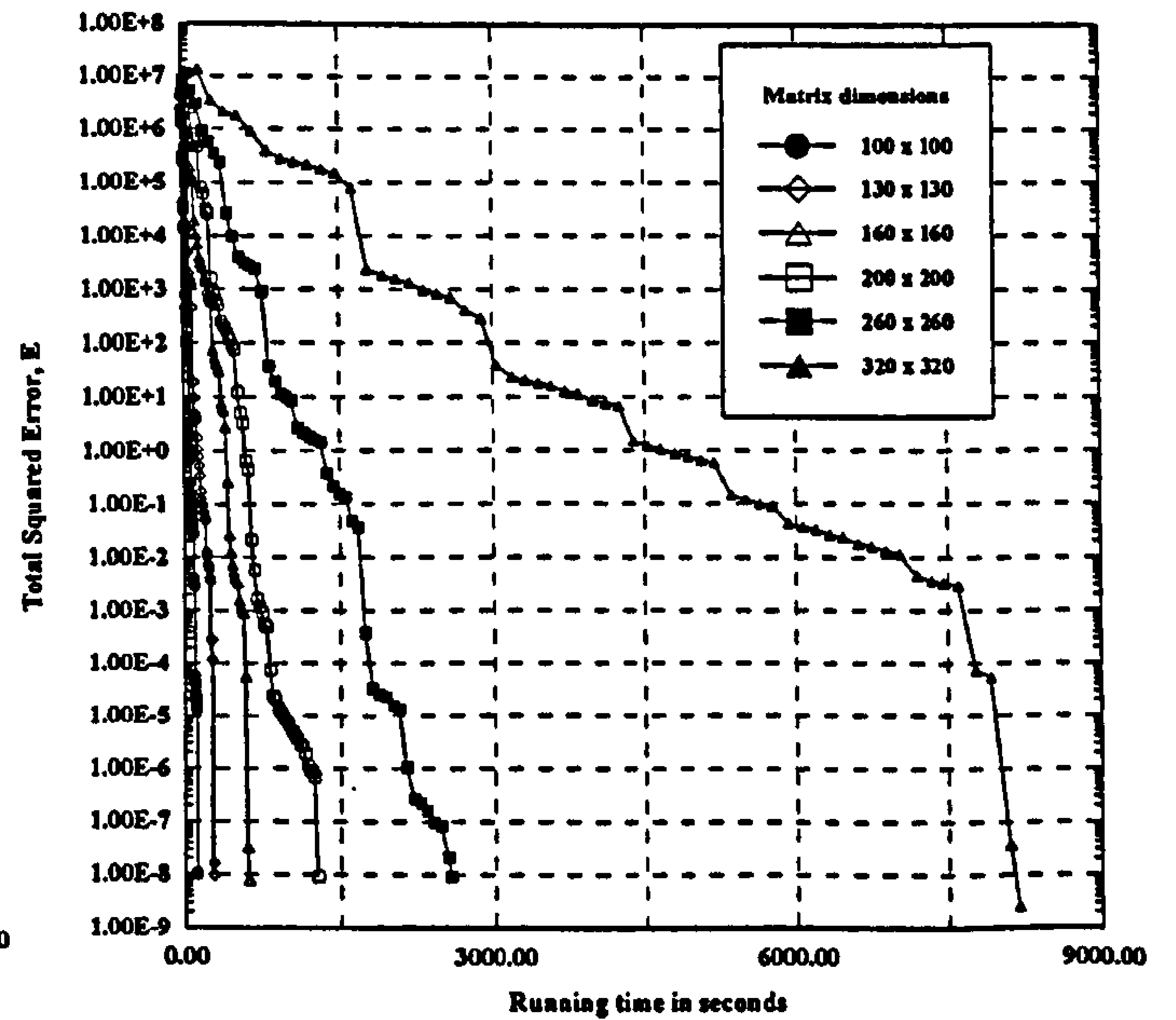
(a) Learning curves for BP method.



(b) Learning curves for SDLS method.



(c) Learning curves for CGFR method.



(d) Learning curves for CGPR method.

Figure 5.29 Effect of matrix size on the convergence history of training methods used for the inversion of square matrices.

CHAPTER 6

Conclusions and Recommendations

The utility of feedforward neural networks (FNN) for matrix computations has been established in this study via the development of new efficient and accurate FNN training methods based on steepest descent method and conjugate gradient methods. This FNN approach has been assessed on only two matrix problems; namely, the LU-decomposition of both band and square ill-conditioned unsymmetric matrices and the inversion of square ill-conditioned unsymmetric matrices. However, the basic idea of this approach can be easily extended to a lot of other matrix algebra problems. This basic idea is: first, represent a given problem by FNN; then, train this network to match some desired patterns; finally, obtain the solution to the problem from the weights of the resulting FNN.

Although the present FNNs have some similarities with conventional FNNs, e.g., both are parallel networks composed of simple elements connected together, and both use training to match the desired patterns, they are different in some fundamental aspects. First, in conventional FNN approaches, only input-output mappings are of interest and the weights are not directly utilized. In the present networks, however, it is the weights, which are of prime interest, i.e., the final converged weights give the solution to the problem. Second, conventional FNNs perform non-linear transformations, whereas the present networks are linear networks. Since what we require are the final converged weights, not the input-output mapping, the present multi-layer networks cannot be simplified into single-layer networks. Third, in conventional FNNs the weights are usually unconstrained; in the present networks, however, the weights are usually strongly constrained, so that we need to construct complicated overlapped networks and develop new training algorithms. In summary, conventional neural network research concentrates on external properties (mappings) of the networks, while the present network research focuses on internal properties (weights) of the network.

Several techniques have been presented in developing the present approach including; line search method to determine the optimal step size as training proceeds, network implementations of some matrix computation problems, first- and second-order minimization methods for updating the weights of the FNN, and restart criteria for second-order methods to insure correct convergence directions. The theoretical analysis presented in this study not only uncovers the nature of the training methods for the FNN, but also provides important insight into developing efficient second-order training methods for training FNNs. The second-order training methods developed herein (i.e., CG methods) have solved the major drawbacks of the standard back-propagation algorithm and of the steepest descent method.

In this study, two performance indexes have been considered; namely, learning speed and convergence accuracy. Extensive computer simulations have been carried out using different training methods; namely, the steepest descent with line search (SDLS) method, conventional back propagation (BP) algorithm, and conjugate gradient (CG) methods; specifically, *Fletcher-Reeves* conjugate gradient (CGFR) method and *Polak-Ribière* conjugate gradient (CGPR) method. The performance comparisons between these minimization methods have demonstrated that CG methods give an approximate increase in the rate of convergence of about an order of magnitude as compared to both SDLS method and BP algorithm when applied to matrix computations. Simulation results have also shown that the CG methods have higher convergence accuracy than both SDLS method and BP algorithm do. The simulation results have also shown that the present second-order training methods have quadratic convergence speed; hence, they, unlike the conventional back propagation algorithm, avoid the phenomena of being trapped in the flat region of error surface. This property generates a higher learning speed and probabilities of locating better local minima than the conventional back propagation.

Further improvement to the CG methods has been achieved by using *Powell*'s restarting criteria. This is because these criteria overcome the problem of wrong convergence directions usually encountered in pure CG training methods. In general, conjugate gradient methods with restarts, especially the *Polak-Ribière* conjugate gradient method, have shown the best performance among all other methods in training the FNN for *LU*-decomposition and matrix inversion.

Accordingly, it is concluded that, of the minimization methods examined herein, the CG methods are by far the superior with respect to learning time; they offer speed-ups of anything between 3 and 4 over SDLS depending on the severity of the error goal chosen and the size of the problem. In fact, they are at least an order of magnitude faster than other training methods and they give better convergence accuracy.

Therefore, the present CG methods could be considered good candidates for training FNN of matrix computations, in particular, *Polak-Ribière* conjugate gradient with *Powell*'s restart criteria, which has shown promising results. Even though, CG methods can be viewed as BP algorithm with momentum, they are still much better. This is because BP with momentum requires high level of expertise to choose the momentum factor that could produce best training results. It is worth mentioning here that the present second-order training methods have two important advantages: Firstly, the explicit computations of the second-order derivative in weight space are not needed and; Secondly, no heavy computational and storage burden is necessary. Accordingly, the convergence of the training process is significantly accelerated and the overall running time for the training procedure is consequently reduced to a great extent.

The effect of the use of line search method on the performance of BP algorithm is clear from the comparison between learning curves of SDLS method and those of BP algorithm. The simulation results show that the rate of convergence for the SDLS method is half of that for the BP method; on the other hand, the convergence accuracy of BP method is much worse than that of the SDLS method. Consequently, the advantage of getting good accuracy when using SDLS method is hindered by the increase in training time as a result of performing line search to determine the optimal learning rate (LR). Of course, the other alternative would be the use of a trial and error approach commonly used for choosing the LR in conventional BP algorithm. This practice usually results in slow convergence, inaccurate results, and may lead to a trapping in a local minimum. Evidently, this practice is no longer required and, consequently, the dependence of the training convergence upon the initial weight values has been eliminated when using SDLS method.

The effect of network size, the degree of accuracy with which the line searches are performed, and the number of training vectors on both iteration time and optimal step size is investigated. Simulation results have shown that, for all test cases, the iteration time of all methods increases as the size of the network (matrix dimensions) increases. Whereas, the rate of convergence decreases for all methods as the size of the problem increases.

Simulation results have shown that the optimal step sizes of the CG methods are on average much larger than those of the SDLS method. It is interesting to note that the size of the *LU*-decomposition network has no effect on the behavior of the optimal step size for the CG methods. Again, this indicates the superiority of these methods over the SDLS method. Moreover, the results have shown that, for square matrices, the SDLS optimal step size behavior deteriorates as the size of the network increases. Thus, it can be stated that the performance of the SDLS method for the *LU*-decomposition of square matrices slows down as the size of the problem increases.

As compared with existing methods for matrix algebra, the present approach in case of *LU*-decomposition involves $2N^3/3$ flops (floating point operations) like the Gaussian elimination formulations presented in reference [1]. Whereas, it involves N^3 flops in the case of matrix inversion. However, it has some very important advantages. First, the present approach uses parallel architectures (i.e., FNN) to represent the problems, and can use parallel algorithms to train the networks, so that it is suitable for VLSI realizations. Second, no divisions are involved in any of the calculations, so that the new approach is free of the divide-by-zero problem. Third, the present approach is so simple and straightforward that no complicated matrix algebra knowledge is required in order to understand and to use this new method. Fourth, the basic idea of the present approach is quite general and can be used for most matrix algebra problems, i.e., the new approach presents a general framework for solving a large variety of matrix algebra problems. Accordingly, the present approach represents an attractive alternative to standard matrix computation algorithms.

Formal mathematical analysis on the convergence speed and accuracy of the present second-order training methods protected by the restart criteria should be a subject of future work. Furthermore, to further reduce the training times, parallelisation of the training methods is suggested. It should be emphasized here that this FNN parallelisation is achieved

through the hardware implementations. This subject needs to be investigated and demonstrated on different training methods to find out the speed-ups when implemented on various matrix computation problems.

References

1. Golub , G.H. and Van Loan , C.F., *Matrix Computations,* 3rd edn, Johns Hopkins, 1996.
2. Dongarra J., Bunch, F.R., Moler, C.B. and Stewart, G.W., *LINPACK Users Guide,* SIAM Publications, Philadelphia, 1978.
3. The MathWorks Inc., *PCMATLAB User's Guide,* 1989.
4. Swartzlander, E.E., Jr., ed., *Systolic Signal Processing Systems,* Marcel Dekker, INC., 1987.
5. Kung, S.Y., Arun, K.S., Gai-Ezer, R.J. and Bhaskar Rao, D.V., "Wavefront Array Processor: Language, Architecture, and Applications," *IEEE Transactions on Computers,* Vol. C-31, Nov. 1982, pp.1054-1066.
6. Kung, S.Y. and Bhaskar Rao, D.V., "Highly Parallel Architectures to Solving Linear Equations," *Proc. ICASSP,* Atlanta, 1981, pp.39-42.
7. Widrow, B. and Stearns, S.D., *Adaptive Signal Processing,* Prentice-Hall, 1985.
8. Kay, S. M. and Shaw, A. K., Frequency estimation by principal AR spectral estimation method without eigendecomposition. *IEEE Trans. Acoustics Speech Signal Process.* Vol. 36(1), 1988, pp. 95-101.
9. Hopfield, J.J., "Neural Networks and Physical Systems with Emergent Collective Computational Ability", *Proceedings of the National Academy of Science,* USA (Biophysics), Vol. 79, 1982, pp. 2554-2558.
10. Hebb, D.O., *The Organization of Behavior.* New York: Wiley, 1949.
11. Hopfield, J.J., "Neurons with graded response have collective computational properties like those of the two-state neurons," *Proceedings of the National Academy of Science.,* Vol. 81, May 1984, pp. 3088-3092.
12. Tank, D.W. and Hopfield, J.J., "Simple Neural Optimization Networks: A/D Converter, Signal Decision Circuit and a Linear Programming Circuit," *IEEE Transactions on Circuits and Systems,* Vol. 33, 1986, pp. 533-541.
13. Wang, J. and Wu, G. "Recurrent Neural Networks for LU Decomposition and Cholesky Factorization", *Mathematical and Computer Modeling,* Vol. 18, 1993, pp. 1-8.

14. Jang, J., Lee, S., and Shin, S., "An Optimization Network for Matrix Inversion", in *Neural Information Processing Systems*, Anderson, D.Z., [Ed.], American Institute of Physics, New York, 1988, pp. 397-401.

15. Luo, F.L. and Zheng B., "Neural Network Approach to Computing Matrix Inversion," *Applied Mathematics and Computation*, Vol. 47, 1992, pp. 109-120.

16. Wang, J., "A Recurrent Neural Network for Real-Time Matrix Inversion", Applied *Mathematics and Computations*, Vol. 55, 1993, pp. 89-100.

17. Wang, J. and Li, H. "Solving Simultaneous Linear Equations Using Recurrent Neural Networks", *Information Sciences*, Vol. 76, 1993, pp. 255-277.

18. Wang, J. "Electronic Realisation of Recurrent Neural Network For Solving Simultaneous Linear Equations", *Electronics Letters,* Vol. 28, No. 5, 27th February 1992, pp. 493-495.

19. De Carvalho, L.A.V. and Barbosa, V.C., " Fast Linear System Solution by Neural Networks," *Operation Research Letters*, Vol. 11, 1992, pp. 141-145.

20. Cichocki, A. and Unbehauen, R., "Neural Networks for Solving Systems of Linear Equations and Related Problems," *IEEE Transactions on Circuits and Systems*, Vol. 39, No. 2, 1992, pp. 124-138.

21. Wang, L. X. and Mendel, J., "Structured Trainable Networks for Matrix Algebra," in *Proceedings of International Joint Conference on Neural Networks*, Vol. II, 1990, pp. 125-132.

22. Wang, L. X. and Mendel, J., "Matrix Computations and Equation Solving Using Structured Networks and Training," in *Proceedings of. IEEE Conference on Decision and Control*, 1990, pp. 1747-1750.

23. Wang, L. X. and Mendel, J., "Parallel Structured Networks for Solving a Wide Variety of Matrix Algebra Problems," *Journal of Parallel and Distributed Computing*, Vol. 14, 1992, pp. 236-247.

24. Wang, L. X. and Mendel, J., "Three-Dimensional Structured Networks for Matrix Equation Solving," *IEEE Transactions on Computers*, Vol. 40, No. 12, 1991, pp. 1337-1347.

25. Parker, D., "Learning-logic," Invention Report S81-64, File 1, Office of Technology Licensing, Stanford University, Stanford, CA, Oct. 1982.

26. Webos, P., *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Ph. D. thesis, Harvard University, Cambridge, MA, Aug. 1974.

27. Parker, D., "Learning-logic," Technical Report TR-47, Center for Computational Research in Economics and Management Science, MIT, April. 1985.

28. D.E.Rumelhart, G.E.Hinton and R.J.Williams, "Learning Internal Representations by Error Propagation," in *Parallel Distributed Processing*, Vol.1, Ch. 8, D.E. Rumelhart and J.L. McCelland, Eds., Cambridge, MA: MIT Press, 1986.

29. Jacobs, R.A., "Increased Rates of Convergence Through Learning Rate Adaptation," *Neural Networks*, Vol. 1 (4), 1988, pp. 295-308.

30. Vogel, T.P., Mangis, J.K., Rigler, A.K., Zink, W.T., and Alkon, D.L., "Accelerating the Convergence of the Backpropagation Method," *Biological Cybernetics*, Vol. 54, 1988, pp. 257-263.

31. Allred, L.G., and Kelly, G.E., "Supervised Learning Techniques for Back-propagation Networks," in *Proceedings of International Joint Conference on Neural Networks*, Vol. I, 1990, pp. 702-709.

32. Hush, D.R., and Salas, J.M., "Improving the Learning Rate of Back-propagation with the Gradient Reuse Algorithm," in *Proceedings of IEEE Conference on Neural Networks*, Vol. I, 1990, pp. 441-447.

33. Weir, M.K., "A Method for Self-determination of Adaptive Learning Rates in Back-propagation," *Neural Networks*, Vol. 4, 1991, pp. 371-379.

34. Yu, X.-H., Chen, G.-A., and Cheng, S.-X., "Dynamic Learning Rate Optimization of the Back-propagation Algorithm," *IEEE Transactions on Neural Networks*, Vol. 6(3), 1995, pp. 669-677.

35. Yu, X.-H and Chen, G.-A., "Efficient Back-propagation Learning Using Optimal Learning Rate and Momentum," *Neural Networks*, Vol. 10 (3), 1997, pp. 517-527.

36. Becker, S. and leCun, Y., "Improving the Convergence of back-propagation Learning with Second order Methods," in *Proceedings of the 1988 Connectionist Models Summer School*, 1988, pp. 29-37.

37. B Battiti, P., "First- and Second-order Methods for Learning: Between Steepest Descent and Newton's Method," *Neural Computation*, Vol. 4, 1992, pp.141-166.

38. Webb, A.R., Lowe, D., and Bedworth, M.D., "A Comparison of Nonlinear Optimization Strategies for Feed-forward Adaptive Layered Networks," *Royal Signals & Radar Establishment*, Memorandum No.4157, 1988.

39. Singhal, S. and Wu, L., "Training Feed-forward Networks with the Extended Kalman Algorithm," In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, Scotland, 1989, pp. 1187-1190.

40. Pushkorius, G.V. and Feldkamp, L.A., "Decoupled Extended Kalman Training of Feed-forward Layered Networks," in *Proceedings of the International Joint Conference on Neural Networks*, Vol. I, Seattle, 1991, pp. 771-777.

41. Kollias, S. and Anastrassiou, D., "An Adaptive Least Square Algorithm for the Efficient Training of Artificial Neural Networks," *IEEE Transactions on Circuits and Systems*, Vol. 36, 1989, pp. 1092-1101.

42. Monhandes, M., Codrington. C.W., and Gelfand, S.B., "Two Adaptive Stepsize Rules for Gradient Descent and Their Application to the Training of Feed-forward Artificial Neural Networks," in *Proceedings of the IEEE International Conference on Neural networks*, Vol. I, Orlando, 1994, pp. 555-560.

43. Hush, D.R. and Home, B.G., "Progress in Supervised Neural Networks," *IEEE Signal Processing Magazine*, Vol. 10 (1), 1993, pp. 8-39.

44. Xu L., Klasa S., and Yuille A., "Recent Advances on Techniques of Static Feed-forward Networks with Supervised Learning," *International Journal of Neural Systems*, Vol.3, 1992, pp. 253-290.

45. Kolen, J.F. and Pollack, J.B., "Back-propagation Is Sensitive to Initial Conditions," in *Advances in Neural Information Processing Systems*, Lippmann, R.P., Moody, J.E., and Tpuretzky, D.S. [Eds.], Morgan Kaufmann, San Mateo, CA, Vol. 3, 1991, pp. 860-867.

46. Lee Y., Oh S. H., and Kim M. W., "An Analysis of Premature Saturation in Back-propagation Learning," *Neural Networks*, Vol. 6, 1993, pp. 719-728.

47. Nguyen, D. and Widrow, B., "Improving the Learning Speed of 2-layer Neural Networks by Choosing Initial Values of the Adaptive Weights," in *Proceedings of the international Joint Conference on Neural Networks*, Vol. III, 1990, pp. 21-26.

48. Wessels L. F. A. and Barnard E., "Avoiding False Local Minima by Proper Initialization of Connections," *IEEE Transactions on Neural Networks*, Vol. 3, 1992, pp. 899-905.

49. Alpsan, D., Towsey, M., Ozdamar, O., Tsoi, A. C., and Ghista, D. N., " Efficacy of Modified Backpropagation and Optimisation Methods on a Real-world Medical Problem", *Neural Networks*, Vol. 8 (6), 1995, pp. 945-962.

50. Hornik K., Stinchcombe, M., and White, H., "Multilayer Feedforward Networks are Universal Approximators," *Neural Networks*, Vol. 2, 1989, pp. 359-366.

51. Blum, E. K. and Li, L. K.,"Approximation Theory and Feedforward Networks," *Neural Networks*, Vol. 4, 1991, pp. 511-515.

52. Siegelmann, J H. T. and Sontag, E. D., "On the Computational Power of Neural Nets," *Journal of Computer and System Sciences*, Vol. 50(1), 1995, pp. 132-150.

53. Magoulas, G.D., Vrahatis, M.N., and Androulakis, G.S., "Effective Backpropagation Training with Variable Step Size", *Neural Networks*, Vol. 10(1), 1997, pp. 69 -82.

54. Munro, J P. W., "Repeat Until Bored: A Pattern Selection Strategy," in *Advances in Neural Information Processing Systems*, J. E. Moody, S. J. Hanson, and R. P. Lippmann, [eds.], Vol. 4, Morgan Kaufman, San Mateo, CA, 1994, pp. 1001-1008.

55. Cachin, C., "Pedagogical Pattern Selection Strategies," *Neural Networks*, Vol. 7(1), 1994, pp. 175-181.

56. Winter, R., Stevenson, M., and Widrow, B. "Sensitivity of Feedforward Neural Networks to Weight Errors," *IEEE Transactions on Neural Networks*, Vol. 1(1), 1990, pp. 71-80, March.

57. Haftka, R.T. and Gurdal, Z., *Elements of Structured* Optimization, 3rd edn, Kluwer Academic, Boston, MA, 1992.

58. Minai, A.A. and Williams, R.D., "Acceleration of Backpropagation through learning rate and momentum adaptation," in *Proceedings of the international Joint Conference on Neural Networks*, Washington, DC, Vol. I, 1990, pp. 676-679.

59. Van Der Smagt, P.P, "Minimization Methods for Training Feedforward Neural Networks," *Neural Networks*, 7(1), 1994, pp.1-11.

60. Beveridge , G. S. G. and Schechter, R. S., *Optimization: Theory and Practice*. McGraw-Hill, New York, 1970.

61. Battiti, R. and Tecchiolli, G., "Learning with First, Second and no Derivatives: A Case Study in High Energy Physics," *Neurocomputing*, Vol. 6, 1994, pp. 181-206.

62. Foulds , L. R., *Optimization Techniques: An Introduction.* Springer-Verlag, New York, 1981.

63. Aylward, S., St Clair, D.C., Bond, W., Flachsbart, B., and Rigler, A.K., "One-dimensional Search Strategies for Conjugate gradient Training of Backpropagation Neural Networks," *Proceedings of the Artificial Neural Networks in Engineering (ANNIE' 92) Conference*, St. Louis, MO, Vol. 2, 1992, pp. 197-202.

64. De Groot, C. and Wurtz, D., " 'Plain Backpropagation' and Advanced Optimization Algorithm: A Comparative Study," *Neurocomputing*, Vol. 6, 1994, pp.153-161.

65. Barnard, E., "Optimisation for Training Neural Nets, *IEEE Transactions on Neural Networks,* Vol. 3, 1992, pp.323-240.

66. Barnard, E. and Holm, J.E., "A Comparative Study of Optimization Techniques for Backpropagation," *Neurocomputing*, Vol. 6, 1994, pp. 19-30.

67. Beale, E.M.L., "A Deviation of Conjugate Gradients," *in Numerical Methods for Nonlinear Optimization*, Lootsma, F.A. [Ed.], London, Academic Press, 1972, pp. 39-43.

68. Powell, M.J.D., "Restart Procedures for the Conjugate Gradient Methods," *Mathematical Programming*, Vol. 12, 1977, pp. 241-254.

69. Moller, M.F., "A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning," *Neural Networks,* Vol. 6, 1993, pp. 525-533.

70. Irwin, G.W. and Mcloone, S., "Fast Gradient Based Off-Line Training of Multilayer Perceptron," *in Dynamic Control Systems*, Hunk, K., Irwin, G., and Warwick, K., [Eds.], 2nd Edn, Springer-Verlager, London, UK, 1996, pp. 179-200.