Journal of the Korean Data & Information Science Society
2012, **23**(5), 1027–1035

http://dx.doi.org/10.7465/jkdi.2012.23.5.1027
한국데이터정보과학회지

An accelerated Levenberg-Marquardt algorithm for feedforward network

Young-Tae Kwak[1]

[1]Department of IT Engineering, Chonbuk National University

Received 7 August 2012, revised 26 August 2012, accepted 17 September 2012

Abstract

This paper proposes a new Levenberg-Marquardt algorithm that is accelerated by adjusting a Jacobian matrix and a quasi-Hessian matrix. The proposed method partitions the Jacobian matrix into block matrices and employs the inverse of a partitioned matrix to find the inverse of the quasi-Hessian matrix. Our method can avoid expensive operations and save memory in calculating the inverse of the quasi-Hessian matrix. It can shorten the training time for fast convergence. In our results tested in a large application, we were able to save about 20% of the training time than other algorithms.

Keywords: Error backpropagation, Levenberg-Marquardt algorithm, multilayer perceptrons.

1. Introduction

The multilayer perceptron (MLP) has been used often in many applications since it was developed. To train the MLP, some have used the error backpropagation (EBP) algorithm because it is easy and simple (Lippmann, 1987; Na and Kwon, 2010; Oh *et al.*, 2011). However, this algorithm has slow convergence due to a gradient descent method. Many attempts have been made to overcome that drawback (Oh and Lee, 1995; Vogl *et al.*, 1988; Yu *et al.*, 1995). However, EBP still suffers from a slow training speed (Buntine, 1994).

As an alternative to EBP, many studies have used a second-order method such as the conjugate gradient method (Charalambous, 1992), quasi-Newton method (Setiono and Hui, 1995), Gauss-Newton method or Levenberg-Marquardt (LM) algorithm (Hagan and Menhaj, 1994; Wilamowski and Yu, 2010). These methods are designed to be faster than EBP. The LM algorithm is estimated to be much faster than the other algorithms if the MLP is not very large. This algorithm is being used as the default in the Matlab Toolbox because of its fast convergence.

However, the LM algorithm must build a Jacobian matrix and calculate the inverse of a quasi-Hessian matrix. The two matrices are a large burden for the LM algorithm. To solve such problems, Lera and Pinzolas (2002) trained the local nodes of the MLP to save both memory and expensive matrix operations. Wilamowski and Yu took the quasi-Hessian matrix directly from the gradient vector of each pattern, without Jacobian matrix multiplication

[1] Associate professor, Department of Information Technology Engineering, Chonbuk National University, Jeonju 561-756, Korea. E-mail: ytkwak@jbnu.ac.kr

and storage. However, their method spends a lot of time in calculating the quasi-Hessian matrix as the number of training patterns increases. Chan and Szeto (1999) suggested both asynchronous and synchronous weight updating with block diagonal matrices. They used recurrent neural network that is different from our proposed method. It was difficult to set many parameters for asynchronous and synchronous weight updating. Until now, neither study attempted to improve the inverse of the quasi-Hessian matrix. Thus, we propose a new approach to effectively determine the inverse matrix.

Though MLP can form fully connected cascade networks for a specific application like a two-spiral task, ordinary MLP has no weights between output nodes, that is, the output layer is not fully connected. Under the structure of such an MLP, the Jacobian matrix becomes sparse, and the quasi-Hessian matrix is symmetric and includes a block diagonal matrix. Sparse and block matrices enable us to avoid repeated operations for finding its inverse. Therefore, we partition the Jacobian matrix into small block matrices and use them to calculate the inverse of the quasi-Hessian matrix. The inverse of a block diagonal matrix becomes another block diagonal matrix. This property can simplify the calculation process in finding the inverse of the quasi-Hessian matrix. Thus, our LM algorithm can reduce training time and the amount of memory by using the inverse of the partitioned quasi-Hessian matrix.

We tested handwritten digit recognition problem in our work. The simulations showed that our method was able to reduce training time by about 20%. Even though all of the algorithms took the same epochs, our method performed a fast and high-quality convergence with fewer matrix operations. The rest of this paper is organized as follows. Section 2 introduces the LM algorithm and the proposed method. The experimental results are shown in Section 3. Finally, we present our conclusion in Section 4.

2. Accelerated LM algorithm

EBP algorithm utilizes a gradient descent method for minimizing mean squared error, and it can take a long time to escape from some flat error surfaces. To improve the training speed of EBP, some have proposed cross-entropy error instead of mean squared error. However, the cross-entropy error has poor generalization ability even though it can train fast. Oh and Lee (1995) suggested the modified error function that improves the cross-entropy function. However, their algorithm still uses a first-order method, that is, a gradient descent method, so it cannot avoid the slow training.

To overcome the slow convergence of EBP, various second-order methods have been proposed. The conjugate gradient method has to conduct golden section search and be reset after passing a fixed number of iterations. The LM algorithm must also calculate a Jacobian matrix to approximate the second derivatives. The equations from (2.1) to (2.4) express the LM algorithm that Hagan and Menhaj (1994) suggested. As you see, saving a quasi-Hessian matrix and calculating its inverse have been critical problems. Thus, we present a new algorithm to calculate them easily and fast.

We only consider a single hidden-layer perceptron because it is a universal approximator. Suppose $\mathbf{x}_p = [1, x_{1p}, x_{2p}, \cdots, x_{Ip}]^T (p = 1, \cdots, P)$ is a training pattern and $\mathbf{t}_p = [t_{1p}, t_{2p}, \cdots, t_{Kp}]^T$ is its target pattern. When a training pattern is presented to the input

layer, each hidden and output node is computed as

$$y_{jp} = \tanh(\sum_{i=0}^{I} v_{ji} x_{ip}), \qquad z_{kp} = \tanh(\sum_{j=0}^{J} w_{kj} y_{jp}) \tag{2.1}$$

where $y_{0p} = 1$ is a bias for the hidden layer, $\mathbf{V}_{(J,(I+1))}$ and $\mathbf{W}_{(K,(J+1))}$ are weight matrices for the hidden and output layers, respectively, P is the number of patterns, and K is the number of output nodes. The error function to be minimized in terms of the weight vector $\mathbf{s}$ is defined as

$$\begin{aligned} E(\mathbf{s}) &= \frac{1}{P}\sum_{k=1}^{K}\sum_{p=1}^{P}(t_{kp} - z_{kp})^2 = \frac{1}{P}\sum_{k=1}^{K}\sum_{p=1}^{P} e_{kp}^2 \\ &= \frac{1}{P}\mathbf{es}^T\mathbf{es}, \qquad \mathbf{es} = [e_{11}, \cdots, e_{1p}, e_{21}, \cdots, e_{2p}, \cdots, e_{K1}, \cdots e_{KP}]^T \end{aligned} \tag{2.2}$$

where $\mathbf{s} = [w_{10}, w_{11}, \cdots, w_{KJ}, v_{10}, v_{11}, \cdots, v_{JI}]^T_{(N,1)}$ includes all weights of the network, and $N(= K(J+1) + J(I+1))$ is the size of vector $\mathbf{s}$.

The LM algorithm updates the weight vector with a modification of the Gauss-Newton method.

$$\begin{aligned} \triangle\mathbf{s} &= -\left(\mathbf{Js}^T\mathbf{Js} + \mu\mathbf{I}\right)^{-1}\mathbf{Js}^T\mathbf{es} \\ &= -\left(\mathbf{qH}\right)^{-1}\mathbf{g} \end{aligned} \tag{2.3}$$

Here, $\mathbf{Js}_{((K\times P),N)}, \mathbf{qH}_{(N,N)}$ and $\mathbf{g}_{(N,1)}$ are a Jacobian matrix, a quasi-Hessian matrix, and a gradient vector, respectively, in relation to the weight vector $\mathbf{s}$, and $\mathbf{I}$ is an identity matrix. The damping parameter μ is adjusted depending on $E(\mathbf{s})$ using a decay rate λ. When $E(\mathbf{s})$ decreases, μ is multiplied by the decay rate $\lambda(0 < \lambda < 1)$, while it is divided by λ when $E(\mathbf{s})$ increases in a new iteration.

We can show each element of $\mathbf{Js}$ in (2.3) in the matrix (2.4). Since the weights between the output nodes of the MLP are not fully connected, all elements do not need to be computed, as in (2.5). Then, $\mathbf{Js}$ becomes a sparse matrix and is changed into (2.6) containing block matrices.

$$\mathbf{Js} = \begin{bmatrix} \frac{\partial e_{11}}{\partial w_{10}} & \frac{\partial e_{11}}{\partial w_{11}} & \cdots & \frac{\partial e_{11}}{\partial w_{KJ}} & \frac{\partial e_{11}}{\partial v_{10}} & \frac{\partial e_{11}}{\partial v_{11}} & \cdots & \frac{\partial e_{11}}{\partial v_{JI}} \\ \frac{\partial e_{12}}{\partial w_{10}} & \frac{\partial e_{12}}{\partial w_{11}} & \cdots & \frac{\partial e_{12}}{\partial w_{KJ}} & \frac{\partial e_{12}}{\partial v_{10}} & \frac{\partial e_{12}}{\partial v_{11}} & \cdots & \frac{\partial e_{12}}{\partial v_{JI}} \\ & & & & \vdots & & & \\ \frac{\partial e_{1P}}{\partial w_{10}} & \frac{\partial e_{1P}}{\partial w_{11}} & \cdots & \frac{\partial e_{1P}}{\partial w_{KJ}} & \frac{\partial e_{1P}}{\partial v_{10}} & \frac{\partial e_{1P}}{\partial v_{11}} & \cdots & \frac{\partial e_{1P}}{\partial v_{JI}} \\ \frac{\partial e_{21}}{\partial w_{10}} & \frac{\partial e_{21}}{\partial w_{11}} & \cdots & \frac{\partial e_{21}}{\partial w_{KJ}} & \frac{\partial e_{21}}{\partial v_{10}} & \frac{\partial e_{21}}{\partial v_{11}} & \cdots & \frac{\partial e_{21}}{\partial v_{JI}} \\ & & & & \vdots & & & \\ \frac{\partial e_{KP}}{\partial w_{10}} & \frac{\partial e_{KP}}{\partial w_{11}} & \cdots & \frac{\partial e_{KP}}{\partial w_{KJ}} & \frac{\partial e_{KP}}{\partial v_{10}} & \frac{\partial e_{KP}}{\partial v_{11}} & \cdots & \frac{\partial e_{KP}}{\partial v_{JI}} \end{bmatrix} \tag{2.4}$$

$$\frac{\partial e_{\alpha p}}{\partial w_{\beta j}} = \begin{cases} \dfrac{\partial e_{kp}}{\partial w_{kj}} & \text{if } \alpha = \beta, \quad \alpha, \beta = 1, \cdots, K \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

$$\mathbf{Js} = \begin{bmatrix} \mathbf{Jw}_1 & \mathbf{0} & \cdots & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{Jv}_1 \\ \mathbf{0} & \mathbf{Jw}_2 & \cdots & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{Jv}_2 \\ \vdots & \vdots & \ddots & \vdots & & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{Jw}_k & \cdots & \mathbf{0} & \mathbf{Jv}_k \\ \vdots & \vdots & & \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \cdots & \mathbf{Jw}_K & \mathbf{Jv}_K \end{bmatrix} \tag{2.6}$$

In (2.6), $\mathbf{Jw}_k$ is a block matrix related to each output node, and $\mathbf{Jv}_k$ is a block Jacobian matrix according to the weight $\mathbf{V}$. Each element of the block matrices can be obtained using (2.7).

$$\begin{aligned} \mathbf{Jw}_k &= \{jw_{\alpha\beta}\}, \quad jw_{\alpha\beta} = \frac{\partial e_{k\alpha}}{\partial w_{k\beta}} \\ & \qquad \alpha = 1, \cdots, P \quad \beta = 0, \cdots, J \\ \mathbf{Jv}_k &= \{jv_{(\alpha,(\beta\gamma))}\}, \quad jv_{(\alpha,(\beta\gamma))} = \frac{\partial e_{k\alpha}}{\partial v_{\beta\gamma}} \\ & \qquad \alpha = 1, \cdots, P \quad \beta = 0, \cdots, J \quad \gamma = 0, \cdots, I \end{aligned} \tag{2.7}$$

The matrix $\mathbf{qH}$ in (2.3) is expanded with $\mathbf{Js}$ and $\mathbf{I}$ as in (2.8), and it produces the 2×2 partitioned matrix in (2.9).

$$\mathbf{qH} = \left[\begin{array}{ccccc|c} \mathbf{Jw}_1^T\mathbf{Jw}_1 + \mu\mathbf{I} & \cdots & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{Jw}_1^T\mathbf{Jv}_1 \\ \vdots & \ddots & \vdots & & \vdots & \vdots \\ \mathbf{0} & \cdots & \mathbf{Jw}_k^T\mathbf{Jw}_k + \mu\mathbf{I} & \cdots & \mathbf{0} & \mathbf{Jw}_k^T\mathbf{Jv}_k \\ \vdots & & \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \cdots & \mathbf{0} & \cdots & \mathbf{Jw}_K^T\mathbf{Jw}_K + \mu\mathbf{I} & \mathbf{Jw}_K^T\mathbf{Jv}_K \\ \hline \mathbf{Jv}_1^T\mathbf{Jw}_1 & \cdots & \mathbf{Jv}_k^T\mathbf{Jw}_k & \cdots & \mathbf{Jv}_K^T\mathbf{Jw}_K & \sum_{\alpha=1}^{K}\mathbf{Jv}_\alpha^T\mathbf{Jv}_\alpha + \mu\mathbf{I} \end{array}\right] \tag{2.8}$$

$$= \left[\begin{array}{c|c} \mathbf{D} & \mathbf{P}_{12} \\ \hline \mathbf{P}_{21} & \mathbf{JV} \end{array}\right] \quad \begin{cases} \mathbf{D} = diag(\mathbf{D}_1, \cdots, \mathbf{D}_K), \\ \qquad \mathbf{D}_k = \mathbf{Jw}_k^T\mathbf{Jw}_k + \mu\mathbf{I} \\ \mathbf{JV} = \sum_{\alpha=1}^{K}\mathbf{Jv}_\alpha^T\mathbf{Jv}_\alpha + \mu\mathbf{I} \\ \mathbf{P}_{12} = [\mathbf{Jw}_1^T\mathbf{Jv}_1, \cdots, \mathbf{Jw}_K^T\mathbf{Jv}_K]^T \\ \mathbf{P}_{21} = [\mathbf{Jv}_1^T\mathbf{Jw}_1, \cdots, \mathbf{Jv}_K^T\mathbf{Jw}_K] \end{cases} \tag{2.9}$$

where $\mathbf{D}$ is a block diagonal matrix. Since $\det(\mathbf{D})\det(\mathbf{JV}) \neq 0$, $\mathbf{D}$ and $\mathbf{JV}$ are invertible, $\mathbf{qH}$ is nonsingular. And $\mathbf{P}_{21}$ is the transpose of $\mathbf{P}_{12}$. Now the complicated matrix $\mathbf{qH}$ becomes a simplified and partitioned matrix.

Next, we calculate the inverse of the quasi-Hessian matrix. The matrix in (2.8) has the following inverse form (Saad, 2003):

$$\mathbf{qH}^{-1} = \begin{bmatrix} \mathbf{D}^{-1} + \mathbf{D}^{-1}\mathbf{P}_{12}\mathbf{S}^{-1}\mathbf{P}_{21}\mathbf{D}^{-1}, & -\mathbf{D}^{-1}\mathbf{P}_{12}\mathbf{S}^{-1} \\ -\mathbf{S}^{-1}\mathbf{P}_{21}\mathbf{D}^{-1}, & \mathbf{S}^{-1} \end{bmatrix} \quad (2.10)$$
$$\mathbf{S} = \mathbf{JV} - \mathbf{P}_{21}\mathbf{D}^{-1}\mathbf{P}_{12}$$

where both $\mathbf{D}$ and $\mathbf{S}$ are assumed to be nonsingular. In addition the matrix $\mathbf{qH}$ is symmetric, so we can simplify it as the following matrix.

$$\mathbf{qH}^{-1} = \begin{bmatrix} \mathbf{D}^{-1} + \mathbf{T}\mathbf{P}_{21}\mathbf{D}^{-1}, & -\mathbf{T} \\ -\mathbf{T}^{T}, & \mathbf{S}^{-1} \end{bmatrix} \quad \begin{array}{l} \mathbf{S} = \mathbf{JV} - \mathbf{P}_{21}\mathbf{D}^{-1}\mathbf{P}_{12} \\ \mathbf{T} = \mathbf{D}^{-1}\mathbf{P}_{21}\mathbf{S}^{-1} \end{array} \quad (2.11)$$

Here, we need to compute just two inverses of $\mathbf{D}^{-1}$ and $\mathbf{S}^{-1}$, and the other operations are matrix multiplications. The computation time is shorter than the time for finding the inverse of the entire $\mathbf{qH}$. The matrix $\mathbf{D}$ in (2.9) is a block diagonal matrix. The inverse of a block diagonal matrix becomes another block diagonal matrix that is composed of the inverse of each block as in (2.12).

$$\mathbf{D}^{-1} = diag(\mathbf{D}_1^{-1}, \cdots, \mathbf{D}_K^{-1}) \quad (2.12)$$

This property results in additional training time savings when using our algorithm. Therefore, when we update the weight vector in (2.3) with both (2.11) and (2.12), our method can reduce the training time considerably. In a large application, it is important and necessary to reduce the computation time per iteration as well as to reduce the number of iterations.

3. Simulation results

For a complex and large application, we tested the proposed method on a handwritten digit recognition database called CEDAR (Hull, 1994). The training pattern has 1000 digits with 100 patterns for each digit. A digit consists of 12×12 pixels that have one of the hexadecimal digits. We simulated our LM algorithms with Matlab scripts but did not use the Neural Network Toolbox. For a more accurate comparison, we used equivalent environments such as initial weights and did not run any other programs in the background. The criterion to stop training was that the maximum number of iterations was greater than 300 or the result of (2.2) was less than 0.1. When the number of iterations of a training process exceeded the maximum, we considered it to be a failure.

Table 3.1 Mean results after 20 runs

Hidden nodes	Method	Iteration	MSE	Training time		Failure	Memory (KByte)
				mean	std (σ)		
15	hgLM	23.05	0.063	103.92	23.09	0	41.59
	alLM	23.05	0.063	83.10	18.06	0	41.42
	ptLM	23.05	0.063	88.48	20.14	0	41.59
	mEBP	300	0.893	11.92	0.33	20	Non
20	hgLM	22.75	0.059	193.12	52.31	0	73.79
	alLM	22.75	0.059	148.80	40.28	0	73.48
	ptLM	22.75	0.059	150.93	41.75	0	73.79
	mEBP	300	0.636	12.68	0.37	20	Non

Table 3.1 shows the mean results obtained after we trained each algorithm for 20 trials. In the table, hgLM, alLM, ptLM, and mEBP stand for the conventional LM algorithm in (Hagan and Menhaj, 1994), our proposed LM algorithm using (2.11) and (2.12), the partitioned LM algorithm using (2.11), and a modified EBP algorithm (Oh and Lee, 1995) respectively. hgLM, alLM, and ptLM have the same number of iterations and mean squared errors. However, there is a large difference in the training times. The proposed methods (alLM and ptLM) save about 17% of the training time in the 15 hidden nodes and about 23% in the 20 hidden nodes. They show that our method can eliminate much of the computation time. In terms of the memory cost, hgLM and ptLM used the same amount of memory, but alLM used slightly less memory as reduced by (2.12). Even though mEBP trained quickly, it did not reach the training criterion and failed in all trials. These results show that mEBP did not exceed the limit of a gradient descent method because it still uses a first-order method.

Table 3.2 ANOVA tables for the training time

Hidden nodes	Sourc	DF	Sum of Squares	Mean square	F value	Pr > F
15	Model	2	4671.60857	2335.80428	5.54	0.0063
	Error	57	24047.59783	421.88768		
	Corrected total	59	28719.20640			
20	Model	2	24994.5558	12497.2779	6.14	0.0038
	Error	57	115955.6191	2034.3091		
	Corrected total	59	140950.1748			

We performed one-way analysis of variance (ANOVA) for the training time of each method in Table 3.2. Here, mEBP was not included because it entirely failed in 20 trials. The results show that the means are not the same for hidden nodes 15 and 20 with p-values of 0.0063 and 0.0038, respectively. The following is the result of DUNCAN multiple comparison for both hidden nodes 15 and 20.

$$\text{hgLM} > \text{ptLM, alLM}$$

Therefore, we can notice that our methods spend less time than the conventional method.

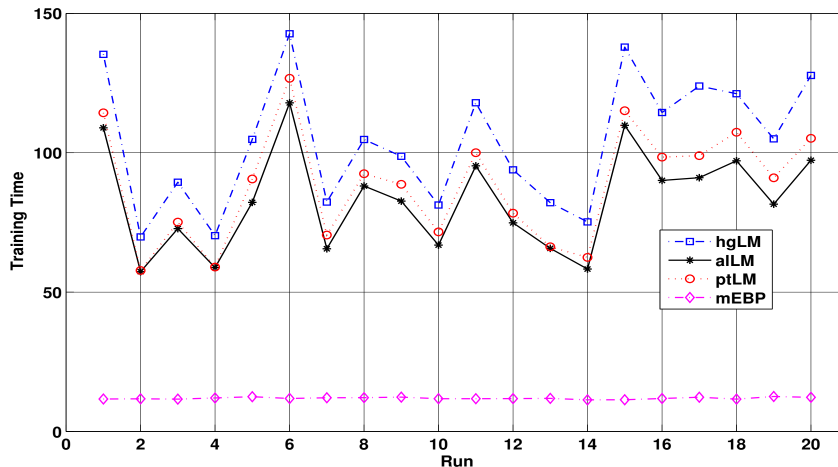

Figure 3.1 Training time with 15 hidden nodes

Figure 3.1 displays the training time for each run when 15 hidden nodes were used for MLP. In the figure, we can see that the proposed methods follow the training process of the traditional LM algorithm, but they reduce the training time more than the LM algorithm. In the case of 16-20 run, using (2.11) and (2.12) saved more training time than using (2.11) alone.

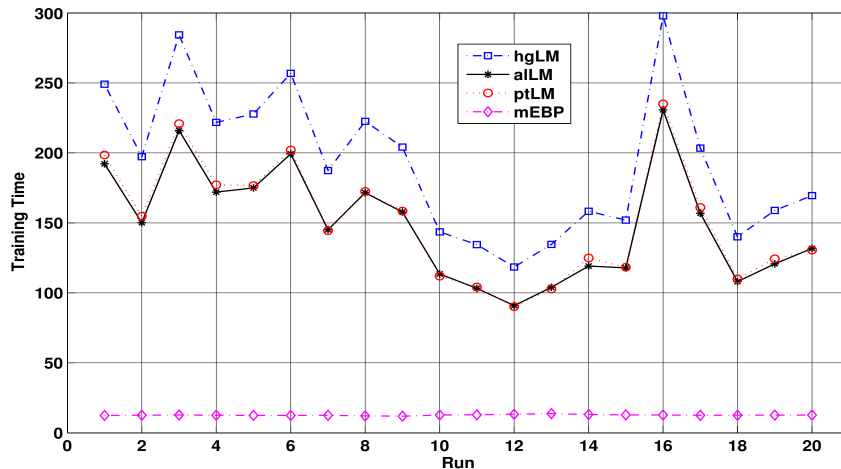

Figure 3.2 Training time with 20 hidden nodes

Figure 3.2 shows the results of 20 hidden nodes, where alLM and ptLM achieved similar results. However, they required less training time than hgLM. In this test, mEBP again failed in all runs. Our methods complete training more quickly than the LM algorithm because the training time per iteration is shorter.

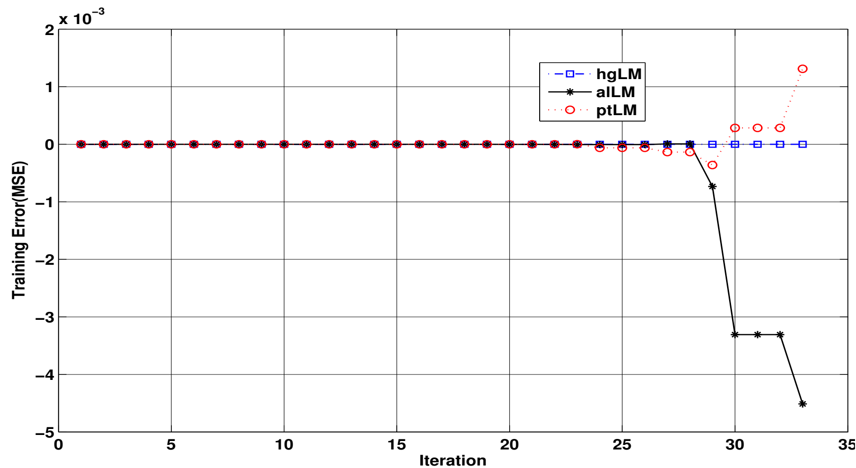

Figure 3.3 Difference in training error in the second run with 20 hidden nodes

Slight differences in the training error sparsely occurred between the traditional LM algorithm and our methods because the size of a matrix whose inverse is being calculated

varied in (2.3), (2.11), and (2.12), due to a machine error. In Figure 3.3, we displayed the training error that made the largest difference among our 40 trials. The figure highlighted each difference on the basis of the hgLM training error; for example, hgLM=hgLM-hgLM and alLM=alLM-hgLM. The errors occurred in the last training session at a 10^{-3} training error rate. However, the errors were so much smaller than those using the training criterion that we can consider them trivial.

In the training the MLP, it is important to reduce the number of iterations. However, it is much more important to shorten the computation time per iteration. Therefore, our methods that reduce training time by about 20% are very useful in large applications.

4. Conclusion

For an application in which the LM algorithm is used, computing the inverse of the quasi-Hessian matrix is critical. As the size of the MLP grows, the LM algorithm requires more training time. Therefore, we proposed a new LM algorithm to reduce training time and memory. We partitioned the Jacobian matrix into block matrices related to the weights of the output node and the hidden node. To determine the inverse of the quasi-Hessian matrix, we simplified the process of finding the inverse through the block diagonal matrix of the output node and the symmetry of the quasi-Hessian matrix.

The handwritten digit recognition test showed that our method reduced training time by more than 20% compared to that of the conventional LM algorithm. In addition, the proposed method did not produce any difference in the number of iterations and the training error compared to those of the other method. In the training the LM algorithm, improving the inverse of the quasi-Hessian matrix means improvement of the LM algorithm itself. Therefore, our method will have better performance when it is applied to a large MLP.

References

Buntine, W. L. and Weigend, A. S. (1994). Computing second derivatives in feed-forward networks: A review. *IEEE Transactions on Neural Networks*, **5**, 480-488.

Chan, L.-W. and Szeto, C.-C. (1999). Training recurrent network with block-diagonal approximated Levenberg-Marquardt algorithm. In *International Joint Conference on Neural Networks*, 1521-1526.

Charalambous, C. (1992). Conjugate gradient algorithm for efficient training of artificial neural networks. *IEEE Proceedings*, **139**, 301-310.

Hagan, M. T. and Menhaj, M. B. (1994). Training feedforward networks with the Marquardt algorithm. *IEEE Transactions on Neural Networks*, **5**, 989-993.

Hull, J. J. (1994). A database for handwritten text recognition research. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **16**, 550-554.

Lera, G. and Pinzolas, M. (2002). Neighborhood based Levenberg-Marquardt algorithm for neural network training. *IEEE Transactions on Neural Networks*, **13**, 1200-1203.

Lippmann, R. (1987). An introduction to computing with neural nets. *IEEE ASSP Magazine*, **4**, 4-22.

Na, M. W. and Kwon, Y. M. (2010). Alternative optimization procedure for parameter design using neural network without SN. *Journal of the Korean Data & information Science Society*, **21**, 211-218.

Oh, K. J., Kim, T. Y., Jung, K. and Kim, C. (2011). Stock market stability index via linear and neural network autoregressive model. *Journal of the Korean Data & information Science Society*, **22**, 335-351.

Oh, S.-H. and Lee, Y. (1995). A modified error function to improve the error back-propagation algorithm for multi-layer perceptrons. *ETRI Journal*, **17**, 11-22.

Saad, Y. (2003). *Iterative methods for sparse linear systems*, SIAM, Philadelphia.

Setiono, R. and Hui, L. C. K. (1995). Use of a quasi-Newton method in a feedforward neural network construction algorithm. *IEEE Transactions on Neural Networks*, **6**, 273-277.

Vogl, T., Mangis, J., Rigler, A., Zink, W. and Alkon, D. (1988). Accelerating the convergence of the back-propagation method. *Biological Cybernetics*, **59**, 257-263.

Wilamowski, B. M. and Yu, H. (2010). Improved computation for Levenberg-Marquardt training. *IEEE Transactions on Neural Networks*, **21**, 930-937.

Yu, X.-H., Chen, G.-A., and Cheng, S.-X. (1995). Dynamic learning rate optimization of the backpropagation algorithm. *IEEE Transactions on Neural Networks*, **6**, 669-677.