Appl. Math. Inf. Sci. 6 No. 2S pp. 505S-511S (2012)

Applied Mathematics & Information Sciences
An International Journal

@ 2012 NSP
Natural Sciences Publishing Cor.

A Layer-by-Layer Levenberg-Marquardt algorithm for Feedforward Multilayer Perceptron

Young-Tae Kwak[1] and Heeseung Jo[1,2]

[1] Department of Information Technology Engineering, Chonbuk National University, 664-14, Deokjin-dong, Jeonju, 561-756, Republic of Korea, *Email: ytkwak@jbnu.ac.kr*

[2] *Corresponding author: Email: heeseung@jbnu.ac.kr*

Received July 1, 2011; Revised August 15, 2011; Accepted September 5 2011
Published online: 1 January 2012

Abstract: The error backpropagation (EBP) algorithm for training feedforward multilayer perceptron (FMLP) has been used in many applications because it is simple and easy to implement. However, its gradient descent method prevents EBP algorithm from converging fast. To overcome the slow convergence of EBP algorithm, the second order methods have adapted. Levenberg-Marquardt (LM) algorithm is estimated to be much faster than other algorithms if the size of FMLP is not large. However, it needs a lot of memory and expensive operations to calculate a Jacobian matrix and its inverse.

This paper proposes an improved LM algorithm that trains the weights of FMLP layer-by-layer. FMLP doesn't have full connections between each output node and between each hidden node. Therefore, our algorithm updates output weights with a Jacobian matrix reduced by its block diagonal matrix. Then we define a new error function for hidden layer derived from output layer's error signals. According to the new error function, we update hidden weights with hidden layer's block diagonal Jacobian matrix. The proposed method can save both memory required and expensive operations of LM algorithm by downsized Jacobian matrices. We tested an iris classification and a handwritten digit recognition for this work. As a result, we found that our method improved training speed and reduced the memory of Jacobian matrix by 30% in the classification and by 10% in the recognition.

Keywords: Error backpropagation, Feedforward multilayer perceptron, Levenberg-Marquardt algorithm, Jacobian matrix

1 Introduction

Since artificial neural network was devised, feedforward multilayer perceptron (FMLP) has been widely used in many applications because it is the first and simplest type of artificial neural network. In the 1980s, some researchers proposed the error backpropagation (EBP) algorithm with a gradient descent method [1,2]. However, the method can take a long time on flat error surfaces, which makes EBP algorithm slow. To speed the training of EBP up, different approaches such as momentum [3], dynamic learning rates [4] and Layer-by-Layer (LbL) algorithms [5-7], have been suggested. Even with all these methods, the convergence of EBP was not able to be improved considerably.

To overcome the slow convergence of EBP algorithm, the second derivatives of error functions have been used in Conjugate Gradient method [8], Quasi-Newton method [9], Gauss-Newton method and Levernberg-Marquardt (LM) algorithm [10,11]. These second order algorithms can train FMLP faster than EBP algorithm owing to approximating the second derivatives.

LM algorithm is estimated to be much faster than other algorithms if the size of FMLP is not very large. However, LM algorithm must calculate Jacobian matrix to obtain quasi-Hessian matrix. Saving the Jacobian matrix and calculating the inverse of quasi-Hessian matrix have been critical problems. To solve these problems, Costa restricted

the norm of weight vectors to speed LM algorithm up [12]. Lera proposed a way to train local nodes of FMLP to save both memory required and expensive operations [13]. Wilamowski took quasi-Hessian matrix directly from the gradient vector of each pattern, without Jacobian matrix multiplication and storage [14]. However, the size of his quasi-Hessian matrix was not reduced. Chan suggested both asynchronous and synchronous weight updating with block diagonal matrix to speed LM algorithm [15]. The experimental results showed that the asynchronous updating method with constraints gave the best improvement in the training time.

FMLP has been used in many applications more often than other networks, except fully connected cascade networks used in two-spiral task [16]. In such FMLP, There are no weights between each output node, in other words, the output layer is not fully connected. When we apply LM algorithm to the FMLP, the Jacobian matrix becomes a sparse matrix. Therefore, the proposed method can get rid of block matrices whose elements are 0 in the Jacobian matrix and reduce the Jacobian matrix for updating the output weights. Here, we use the inverse of block diagonal matrix for quasi-Hessian matrix.

Layer-by-Layer algorithms update the weights of output layer at first, and define a new error function at hidden layer and update the weights of hidden layer according to the defined error function. For our Layer-by-Layer LM (LbL LM) algorithm, we define a new hidden error function derived from the error signals of output layer. It is composed of the sum of squared errors between error signals and hidden outputs. Here, we can also obtain a block diagonal Jacobian matrix for hidden layer because the weights between each hidden node are also not connected. The reduced hidden Jacobian matrix is used for updating the weights of hidden layer. As a result, our LbL LM algorithm can not only improve training time but also reduce considerable memory through the inverse of block diagonal Jacobian matrices.

In the simulation, we tested both a simple classification and a complicated recognition. The results of simulations confirmed that our approach was able to improve training speed and save a lot of memory. Even in the problem with huge weights, our method performed a fast and high-quality convergence with smaller memory and operations. This paper is organized as follows. Section 2 introduces LM algorithm and proposes a Layer-by-Layer LM algorithm with a new hidden error function. The experimental results and analyses are shown in Section 3. Finally, we will give our conclusion in Section 4.

2 Layer-by-Layer LM Algorithm

2.1 FMLP and Notation

We use a single hidden-layer perception in the figure 2.1 because it is universal approximation theorem [17]. The notations of FMLP are as follow. The bold upper cases stand for matrices and the bold lower cases vectors.

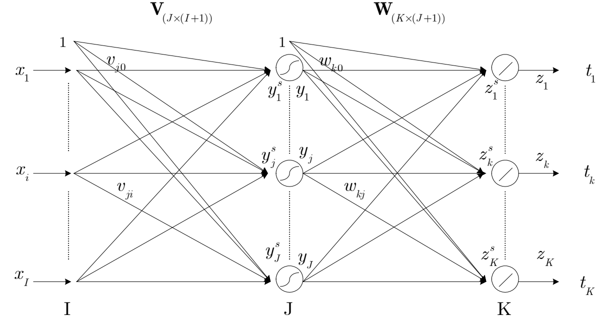

Figure 2.1: Feedforward single hidden-layer perception

$$\mathbf{x}_p = [1, x_{1p}, x_{2p}, \cdots, x_{Ip}]^T, \; p = 1, \cdots, P, \; \mathbf{X}_{((I+1),P)}$$

$$\mathbf{t}_p = [t_{1p}, t_{2p}, \cdots, t_{Kp}]^T, \quad \mathbf{T}_{(K,P)}$$

$$y_{jp}^s = \sum_{i=0}^{I} v_{ji} x_{ip}, \quad \mathbf{Y}_{(J,P)}^s = \mathbf{V}_{(J,(I+1))} \mathbf{X}_{((I+1),P)} \tag{2.1}$$

$$y_{jp} = f(y_{jp}^s), \quad \mathbf{Y}_{(J,P)} = f(\mathbf{Y}_{(J,P)}^s) \tag{2.2}$$

$$z_{kp}^s = \sum_{j=0}^{J} w_{kj} y_{jp}, \; \mathbf{Z}_{(K,P)}^s = \mathbf{W}_{(K,(J+1))} \mathbf{Y}_{((J+1),P)} \tag{2.3}$$

here, $\mathbf{y}_p = [1, y_{1p}, y_{2p}, \cdots, y_{Jp}]^T \leftarrow$ add a bias

$$z_{kp} = f(z_{kp}^s), \quad \mathbf{Z}_{(K,P)} = f(\mathbf{Z}_{(K,P)}^s),$$

$$\mathbf{z}_p = [z_{1p}, z_{2p}, \cdots, z_{Kp}]^T \tag{2.4}$$

For activation function, *tanh* function is used. t_{kp} is the target value of the kth output and the pth pattern, z_{kp} is the actual value of the kth output and the pth pattern. P is the number of patterns, and K is the number of the network outputs.

2.2 LM Algorithm and Block Diagonal Matrix

The error function to be minimized by LM algorithm, the sum of squared errors for a weight vector $\mathbf{s}$, is defined as

$$E(\mathbf{s}) = \frac{1}{P}\sum_{k=1}^{K}\sum_{p=1}^{P}(t_{kp} - z_{kp})^2 = \frac{1}{P}\sum_{k=1}^{K}\sum_{p=1}^{P} e_{kp}^2 \tag{2.5}$$

$$= \frac{1}{P}\mathbf{es}^T\mathbf{es}, \; \mathbf{es} = [e_{11}\cdots e_{1P}, e_{21}\cdots e_{2P}, \cdots, e_{K1}\cdots e_{KP}]^T$$

where $\mathbf{s} = [w_{10}, w_{11}, \cdots, v_{10}, v_{11}, \cdots v_{JI}]_{(N,1)}^T$ consists of all weights of the network. $N(= J\times(I+1) + K\times(J+1))$ is the number of the weights.

LM's weight updating is carried out from a

modification of the Gauss-Newton method.

$$\begin{aligned}\Delta \mathbf{s} &= -\left(\mathbf{Js}^T\mathbf{Js}+\mu\mathbf{I}\right)^{-1}\mathbf{Js}^T\mathbf{es} \\ &= -\left(\mathbf{qH}^T+\mu\mathbf{I}\right)^{-1}\mathbf{g}\end{aligned} \quad (2.6)$$

$\mathbf{Js}_{((K\times P),N)}$, $\mathbf{qH}_{(N,N)}$ and $\mathbf{g}_{(N,1)}$ are Jacobian matrix, quasi-Hessian matrix and a gradient vector respectively in relation to a weight vector **s**. μ is called a damping parameter and **I** is an identity matrix. The damping parameter is adjusted according to the evaluation of $E(\mathbf{s})$ in equation (2.5). If $E(\mathbf{s})$ is lower than a trial error $E(\mathbf{s}^{trial})$ that is calculated by a new weight, μ is increased by a factor λ (0< λ<1) called decay rate in equation (2.7). Reversely, if $E(\mathbf{s})$ is higher, μ is decreased by λ and the new updated weight is accepted.

$$\mu = \begin{cases} \mu/\lambda & \text{if } E(\mathbf{s}) \le E(\mathbf{s}^{trial}) \\ \mu\cdot\lambda & \text{if } E(\mathbf{s}) > E(\mathbf{s}^{trial}) \end{cases} \quad (2.7)$$

Each element of **Js** in equation (2.6) is equal to equation (2.8). However, all elements don't need to be computed like equation (2.9) because the weights between output nodes in FMLP are not fully connected. Then, **Js** becomes a sparse matrix and can be changed into equation (2.10) containing block matrices.

$$\mathbf{Js} = \begin{bmatrix} \frac{\partial e_{11}}{\partial w_{10}}, \frac{\partial e_{11}}{\partial w_{11}}, \cdots, \frac{\partial e_{11}}{\partial w_{KJ}}, \frac{\partial e_{11}}{\partial v_{10}}, \frac{\partial e_{11}}{\partial v_{11}}, \cdots, \frac{\partial e_{11}}{\partial v_{JI}} \\ \frac{\partial e_{12}}{\partial w_{10}}, \frac{\partial e_{12}}{\partial w_{11}}, \cdots, \frac{\partial e_{12}}{\partial w_{KJ}}, \frac{\partial e_{12}}{\partial v_{10}}, \frac{\partial e_{12}}{\partial v_{11}}, \cdots, \frac{\partial e_{12}}{\partial v_{JI}} \\ \vdots \\ \frac{\partial e_{1P}}{\partial w_{10}}, \frac{\partial e_{1P}}{\partial w_{11}}, \cdots, \frac{\partial e_{1P}}{\partial w_{KJ}}, \frac{\partial e_{1P}}{\partial v_{10}}, \frac{\partial e_{1P}}{\partial v_{11}}, \cdots, \frac{\partial e_{1P}}{\partial v_{JI}} \\ \frac{\partial e_{21}}{\partial w_{10}}, \frac{\partial e_{21}}{\partial w_{11}}, \cdots, \frac{\partial e_{21}}{\partial w_{KJ}}, \frac{\partial e_{21}}{\partial v_{10}}, \frac{\partial e_{21}}{\partial v_{11}}, \cdots, \frac{\partial e_{21}}{\partial v_{JI}} \\ \vdots \\ \frac{\partial e_{KP}}{\partial w_{10}}, \frac{\partial e_{KP}}{\partial w_{11}}, \cdots, \frac{\partial e_{KP}}{\partial w_{KJ}}, \frac{\partial e_{KP}}{\partial v_{10}}, \frac{\partial e_{KP}}{\partial v_{11}}, \cdots, \frac{\partial e_{KP}}{\partial v_{JI}} \end{bmatrix} \quad (2.8)$$

$$\frac{\partial e_{\alpha p}}{w_{\beta j}} = \begin{cases} \frac{\partial e_{kp}}{w_{kj}} & \text{if } \alpha=\beta, \quad \alpha,\beta = 1,\cdots,K \\ 0 & \text{otherwise} \end{cases} \quad (2.9)$$

$$\mathbf{Js} = \left[\overbrace{\begin{bmatrix} \mathbf{Jw}_1 & 0 & \cdots & 0 \\ 0 & \mathbf{Jw}_2 & \cdots & \vdots \\ \vdots & \vdots & \mathbf{Jw}_k & 0 \\ 0 & 0 & \cdots & \mathbf{Jw}_K \end{bmatrix}}^{\mathbf{JW}} \overbrace{\begin{bmatrix} \vdots \\ \vdots \\ \vdots \\ \vdots \end{bmatrix}}^{\mathbf{JV}}\right] \quad (2.10)$$

In equation (2.10), $\mathbf{Jw}_k$ are block matrices related to each output node in equation (2.11). **JV** is a block Jacobian matrix according to weight **V**.

$$\mathbf{Jw}_k = \left\{ jw_{\alpha\beta} \right\}, \ jw_{\alpha\beta} = \frac{\partial e_{k\alpha}}{\partial w_{k\beta}}, \quad \alpha = 1,\cdots,P \ \ \beta = 0,\cdots,J \quad (2.11)$$

When **Js** is used to approximate quasi-Hessian matrix **qH**, the block matrix **JV** affects **qH** on the whole. The inverse of **qH** is also affected by **JV**. However, if we apply a Layer-by-Layer algorithm to FMLP, the block **JV** can be disregarded because one layer is only considered. Therefore we can use the following property of block diagonal matrix in equation (2.12): The inverse of a block diagonal matrix is another block diagonal matrix, composed of the inverse of each block.

$$\begin{bmatrix} \mathbf{A}_1 & 0 & \cdots & 0 \\ 0 & \mathbf{A}_2 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \mathbf{A}_n \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}_1^{-1} & 0 & \cdots & 0 \\ 0 & \mathbf{A}_2^{-1} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \mathbf{A}_n^{-1} \end{bmatrix} \quad (2.12)$$

2.3 Layer-by-Layer LM Algorithm

Our LbL LM algorithm updates the weights of output layer through such error function as defined in equation (2.5), and defines a new error function for hidden layer and updates the weights of hidden layer according to the new error function. When the Jacobian matrix of equation (2.5) is computed with respect to the only weight **W**, it shows the block diagonal matrix **JW** like equation (2.10). Each element of $\mathbf{Jw}_k$ can be computed by equation (2.13). The proposed method uses the inverse of a block diagonal matrix for removing 0 elements. Therefore we can accomplish the weight updating of output layer like equation (2.14).

$$\mathbf{Jw}_k = \left\{ \mathbf{jw}_p \right\}, \ \mathbf{jw}_p = -f'(z^s_{kp})\mathbf{y}^T_p, \ p = 1,\cdots,P \quad (2.13)$$

$$\Delta\mathbf{w}_k = -(\mathbf{Jw}_k^T\mathbf{Jw}_k + \mu_{\mathbf{w}}\mathbf{I})^{-1}\mathbf{Jw}_k^T\mathbf{ew}_k, k = 1,\cdots,K \quad (2.14)$$
$$,\mathbf{ew}_k = [e_{k1}, e_{k2}, \cdots, e_{kP}]^T$$

To update the hidden layer's weights, we have to define a new error function. The error function is derived from the error signals of output layer like equation (2.15). This way is similar to a gradient descent method. In equation (2.16), the new error function is composed of the sum of squared errors between error signals and hidden outputs.

$$\delta_{jp} = \sum_{k=1}^{K} w_{kj} f'(z^s_{kp})(t_{kp} - z_{kp}) \quad (2.15)$$

$$E^{hid}(\mathbf{v}) = \frac{1}{P}\sum_{j=1}^{J}\sum_{p=1}^{P}\left(e^h_{jp}\right)^2 = \frac{1}{P}\left(\mathbf{ev}^h\right)^T\mathbf{ev}^h \quad (2.16)$$

$$e^h_{jp} = \delta_{jp} - y_{jp}, \ \mathbf{ev}^h = [e^h_{11}\cdots e^h_{1P}, \cdots, e^h_{J1}\cdots e^h_{JP}]^T$$

Since Layer-by-Layer algorithm considers only one layer, we just pay attention to the error function

E^{hid} and weight **V**. The weights between hidden nodes at hidden layer are also not connected. So we can obtain a block diagonal Jacobian matrix **Jv** like equation (2.17). The matrix in equation (2.17) has the same shape of output layer's **Jw**.

$$\mathbf{Jv} = \begin{bmatrix} \mathbf{Jv}_1 & 0 & \cdots & 0 \\ 0 & \mathbf{Jv}_2 & \cdots & \vdots \\ \vdots & \vdots & \mathbf{Jv}_j & 0 \\ 0 & 0 & \cdots & \mathbf{Jv}_J \end{bmatrix} \quad (2.17)$$

Our algorithm follows the repetitive processes of the output layer for obtaining **Jv** and hidden layer's updating. Each element of $\mathbf{Jv}_j$ is computed by equation (2.18) and the weight updating of hidden layer is accomplished like equation (2.19).

$$\mathbf{Jv}_j = \{\mathbf{jv}_p\}, \quad \mathbf{jv}_p = -f'(y^s_{jp})\mathbf{x}^T_p, \quad p = 1,\cdots,P \quad (2.18)$$

$$\begin{aligned} \Delta\mathbf{v}_j &= -\left(\mathbf{Jv}_j^T\mathbf{Jv}_j + \mu_{\mathbf{v}}\mathbf{I}\right)^{-1}\mathbf{Jv}_j^T\mathbf{ev}_j, \quad j = 1,\cdots,J \\ \mathbf{ev}_j &= [e^h_{j1}, e^h_{j2},\cdots,e^h_{jP}]^T \end{aligned} \quad (2.19)$$

Each LM algorithm runs in each layer. Then the damping parameters ($\mu_{\mathbf{w}}$, $\mu_{\mathbf{v}}$) are adjusted according to each error function. Our LbL LM algorithm can be summarized as follows:

1. Initialize the weights and parameters $\mu_{\mathbf{w}}$, $\mu_{\mathbf{v}}$ and λ (for example, set $\mu_{\mathbf{w}} = \mu_{\mathbf{v}}$ =0.01 and λ =0.1)
2. Stop if the number of iteration exceeds the maximum iteration or $E(\mathbf{s})$ in equation (2.5) is below a desired error
3. Compute $E(\mathbf{s})$ in equation (2.5) after passing all input patterns
4. Compute each Jacobian matrix $\mathbf{Jw}_k$ in equation (2.13)
5. Solve equation (2.14) to get the weight changes $\Delta\mathbf{w}_k$
6. Recompute $E(\mathbf{s}^{trial})$ using $\mathbf{w}^{trial} = \mathbf{w} + \Delta\mathbf{w}_k$ over all patterns
   IF $E(\mathbf{s}^{trial}) > E(\mathbf{s})$ THEN
      $\mu_{\mathbf{w}} = \mu_{\mathbf{w}} / \lambda$
      go back to step 5
   ELSE
      $\mu_{\mathbf{w}} = \mu_{\mathbf{w}} \cdot \lambda$
      $E(\mathbf{s}) = E(\mathbf{s}^{trial})$, $\mathbf{w} = \mathbf{w}^{trial}$
7. Evaluate $E^{hid}(\mathbf{v})$ in equation (2.16) with the updated **w** and **Y**
8. Compute each Jacobian matrix $\mathbf{Jv}_j$ in equation (2.18)
9. Solve equation (2.19) to get the weight changes $\Delta\mathbf{v}_j$
10. Recompute $E^{hid}(\mathbf{v}^{trial})$ using $\mathbf{v}^{trial} = \mathbf{v} + \Delta\mathbf{v}_j$ over all patterns
   IF $E^{hid}(\mathbf{v}^{trial}) > E(\mathbf{s})$ THEN
      $\mu_{\mathbf{v}} = \mu_{\mathbf{v}} / \lambda$
      go back to step 9
   ELSE
      $\mu_{\mathbf{v}} = \mu_{\mathbf{v}} \cdot \lambda$
      $E(\mathbf{s}) = E^{hid}(\mathbf{v}^{trial})$, $\mathbf{v} = \mathbf{v}^{trial}$
11. go back to step 2

For saving Jacobian matrix, LM algorithm needs *KPN* memory but our Layer-by-Layer algorithm has *PN* memory so that it can save *(K-1)PN*. In quasi-Hessian matrix operation, LM algorithm allocates N^2 memory. On the other hand, our method only requires $K(J+1)^2+J(I+1)^2$ memory. Through saving these memories, we can not only train LM algorithm fast but also reduce a lot of memory.

3 Experiments

For our simulation, we tested both an iris classification and a handwritten digit recognition [18,19]. Our LbL LM algorithm was implemented with Matlab scripts not using Matlab NN Toolbox. To compare fairly with EBP and LM methods, we equally initialized the initial weights. In addition, our simulation was run in standard alone without any running program to measure exact time. We tried to train FMLP for 20 runs according to each number of hidden nodes to get more general results.

3.1 Iris Classification

Iris classification is that FMLP gets 4 input data and classifies three types of iris. So FMLP consisted of 4 input nodes, 3 output nodes and 7-9 hidden nodes. We used 150 input patterns and the error function in equation (2.5). The criterion to stop training was that the maximum iteration was 300 or equation (2.5) was less than 0.1. If the iteration of a training exceeded the maximum, we considered the training as a failure.

Table 1 is the iris results averaged after trying 20 runs. EBP algorithm failed in all trials but we showed them to compare with other methods. In the table, *Iteration*, *Total time*, *Time/Epoch* stand for the number of epochs, the whole time taken to finish a run, the time per epoch, respectively. Memory cost is the memories to store Jacobian matrices. Here, Nil means that any Jacobian matrix is not needed in both EBP and Wilamowski algorithm. Wilamowski's method calculates quasi-Hessian matrix directly from each training pattern. Then it doesn't have Jacobian matrix but stores quasi-Hessian matrix not reduced into small size. So we will compare our method with LM algorithm. *LbL* rows are our method. *WM* rows are the results of Wilamowski method.

As Table 1 shows, the speed difference between LM and LbL LM algorithm is not large because the iris problem is simple. The total time and iteration mostly depend on the number of failures. However, we should notice that LbL LM algorithm is shorter

Table 1 Comparison of Iris results

Hidden nodes /Method		Iteration	Time/Epoch (10^{-2})	Total time	Number of failures	Memory cost (KByte)
7	LM	32.1	6.04	1.75	0	207
	WM	37.5	5.91	1.75	1	Nil
	LbL	28.2	4.28	0.93	1	70
	EBP	300	2.44	7.35	20	Nil
8	LM	48.3	4.77	1.83	2	235
	WM	21.4	5.93	1.24	0	Nil
	LbL	13.5	2.66	0.35	0	79
	EBP	300	2.99	8.99	20	Nil
9	LM	20	4.6	0.92	0	263
	WM	28.9	6.06	1.58	0	Nil
	LbL	41.5	2.91	1.08	2	89
	EBP	300	2.28	6.87	20	Nil

than LM algorithm in the time per epoch and it can save about 70% memory for storing Jacobian matrix. In the case of 9 hidden nodes, the total time of LbL LM is similar to the LM's total time even though LbL LM takes two times more iteration, because the Time/Epoch of LbL LM is shorter. On the whole, we can see that LbL LM algorithm outperformed other methods in training speed as well as saving memory.

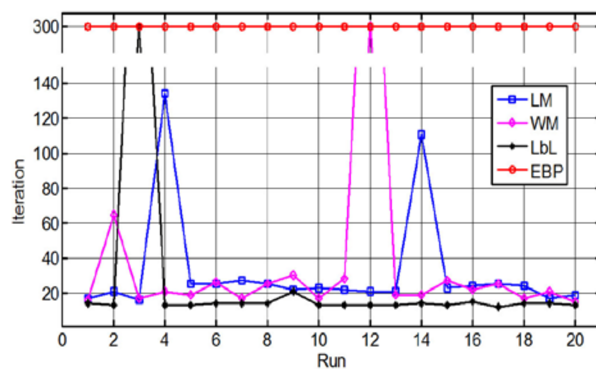

Figure 3.1: Comparison of iterations in iris results trained with 7 hidden nodes

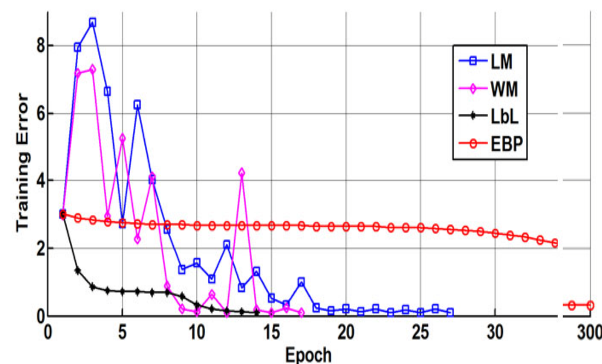

Figure 3.2: Training errors of 7th run among iris results trained with 7 hidden nodes

Figure 3.1 shows the training iterations obtained by 7 hidden nodes for 20 runs. We magnified the important part to distinguish the figure clearly. Through this figure, we can find that our method has faster training speed than others. Figure 3.2 illustrates the comparison of training errors on the 7th run among the results of figure 3.1. Our method and EBP algorithm reduced its training error smoothly, while LM algorithm made frequent oscillations. It was caused by adapting the damping parameter to seek target weights. These oscillations made LM algorithm slower.

3.2 Handwritten Digit Recognition

To test a more complicated problem and get more general results, we applied our approach to a handwritten digit recognition. We used 1000 input patterns, 100 patterns of each digit. Each digit was 12x12 pixels and each pixel had a hexadecimal value by gray level. FMLP had 144 input nodes and 10 output nodes, and we trained it by changing 20-22 hidden nodes to simulate FMLP having a huge weight space. These FMLPs cost a lot of time because of huge weights.

Table 2 shows the results of CEDAR data. All EBP trials also failed in this test. This table tells us that our method obtained better results than other methods in training time and memory storage. In the total time, our results were shorter than LM algorithm's results and even EBP algorithm's results. They took about 15% of LM algorithm's total time. In this test, LbL LM algorithm took much more iteration than LM algorithm. However, because its Time/Epoch is even shorter, LbL LM algorithm was able to train faster than other algorithms. We must pay attention to the memory cost that our method needed the 10% of LM algorithm for saving Jacobian matrix. This reduced memory made LbL LM algorithm train faster than others.

Figure 3.3 is the results of CEDAR used with 20 hidden nodes. Figure 3.4 represents the training errors of the 10th run among figure 3.3. Figure 3.3

Table 2 Comparison of CEDAR results

Hidden nodes /Method		Iteration	Time/Epoch	Total time	Number of failures	Memory cost (MByte)
20	LM	16.8	4.64	78.9	0	237.2
	WM	14.15	4.36	18.4	0	Nil
	LbL	58.4	0.25	12.1	1	23.7
	EBP	300	0.04	12.9	20	Nil
21	LM	14.4	5.05	73.3	0	249.0
	WM	16.6	0.44	11.6	0	Nil
	LbL	43.3	0.26	9.09	1	24.9
	EBP	300	0.04	13.4	20	Nil
22	LM	13.2	5.43	72.2	0	260.9
	WM	14	5.76	13.4	0	Nil
	LbL	51.5	0.25	11.4	1	26.0
	EBP	300	0.04	13.5	20	Nil

shows that LbL LM algorithm required more iteration number than LM algorithm. However, when we refer the Time/Epoch in Table 2, we can know that our method took the less total time than other methods. Like the iris classification, Figure 3.4 displays that LM algorithm was still oscillating, while our way had a fast and smooth convergence.

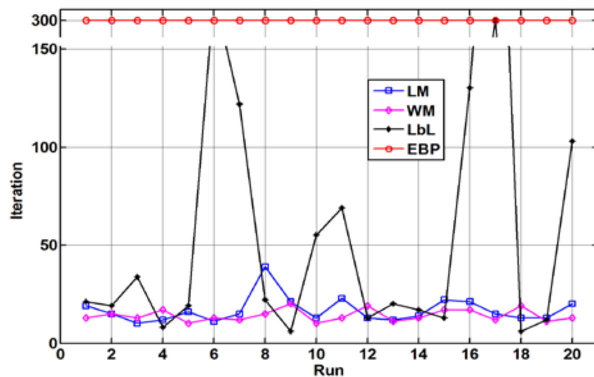

Figure 3.3: Comparison of iterations in CEDAR results trained with 20 hidden nodes

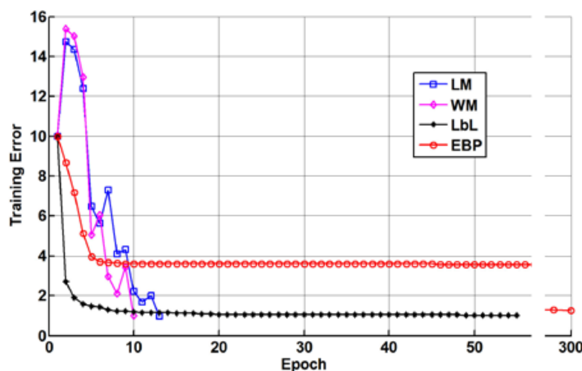

Figure 3.4: Training errors of 10th run among CEDAR results trained with 20 hidden nodes

As Table 1 and Table 2 show, the rate of failure of the proposed LbL LM algorithm is higher than the LM algorithm. That is why our method makes a partial Jacobian matrix in each layer. Since each layer's Jacobian matrix doesn't have the Jacobian matrix of other layer, our method can't calculate more accurate weight direction than the LM algorithm. However, the rate of the failure is so rare that the LbL LM algorithm can bring better benefit such as saving memory and time in practical applications.

4 Conclusion

LM algorithm is estimated to be much faster than other algorithms if the size of network is not very large. However, it must calculate and save Jacobian matrix for quasi-Hessian matrix. The manipulations and operations of Jacobian matrix have been critical problems. Then we proposed a Layer-by-Layer LM algorithm using block diagonal matrix to converge FMLP faster and save memory for Jacobian matrix.

FMLP have no weights between output nodes because they are not fully connected. Each element of Jacobian matrix related to different output nodes become zero. So our LM algorithm updated output weights with a reduced block diagonal Jacobian matrix. For hidden layer, we defined a new error function derived from output layer's error signal. According to the error function, we updated hidden weights layer-by-layer by the hidden layer's block diagonal Jacobian matrix, because each hidden node is also not fully connected. The proposed method took a less training time and storage than other algorithms through downsized Jacobian matrices.

We were able to confirm our algorithm through the simulations. Even though our method took a little longer iteration time, we were able to train FMLP faster than others because its time per epoch was even shorter. We also reduced Jacobian matrix's memory by 30% in iris classification and by 10% in CEDAR recognition. Our approach has the merit of converging FMLP faster and spending

memory less than LM algorithm when it is used for an application having huge weight space.

Acknowledgements
This paper was supported by research funds of Chonbuk National University in 2010.

References
[1] R. P. Lippman, An Introduction to Computing with Neural Nets, IEEE ASSP Magazine, Vol. 4, No. 2, (1987), 4-22.

[2] D. E. Rumelhart and J. L. McClelland, Parallel Distributed Processing, MIT Press, Cambridge, MA, (1986), 318-362.

[3] T. P. Vogl, J. K. Mangis, A. K. Zigler, W. T. Zink and D. L. Alkon, Accelerating the Convergence of the Back-Propagation method, Biological Cybernetics, Vol. 59, (1988), 256-263.

[4] Xiao-Hu Yu, Guo-An Chen and Shi-Xin Cheng, Dynamic Learning Rate Optimization of the Backpropagation Algorithm, IEEE Trans. on Neural Networks, Vol. 6, No. 3, (1995), 669-677.

[5] S. Ergezinger and E. Thomsen, An accelerated learning algorithm for multilayer perceptrons optimization Layer by Layer, IEEE Tarns. on Neural Networks, Vol. 6, No. 1, (1995), 31-42.

[6] Gou-Jen Wang and Chih-Cheng Chen, A fast multilayer neural-network training algorithm based on the layer-by-layer optimizing procedures, IEEE Tarns. on Neural Networks, Vol. 7, No. 3, (1996), 768-775.

[7] Sang-Hoon Oh and Soo-Young Lee, A New Error Function at Hidden Layers for Fast Training of Multilayer Perceptrons, IEEE Tarns. on Neural Networks, Vol. 10, No. 4, (1999), 960-964.

[8] C. Charalambous, Conjugate gradient algorithm for efficient training of artificial neural networks, IEEE Proceedings, Vol. 139, No. 3, (1992), 301-310.

[9] Rudy Setiono and Lucas Chi Kwong Hui, Use of a Quasi-Newton Method in a Feedforward Neural Network Construction Algorithm, IEEE Tarns. on Neural Networks, Vol. 6, No. 1, (1995), 273-277.

[10] Martin T. Hagan and Mohammad B. Menhaj, Training feedforward networks with the Marquardt algorithm, IEEE Trans. on Neural Networks, Vol. 5, No. 6, (1994), 989-993.

[11] Martin. T. Hagan, Howard B. Demuth and Mark Beale, Neural Network Design, PWS Publishing Company, (1995).

[12] Marcelo Azevedo Costa, Antonio de Padua Braga and Benjamin Rodrigues de Menezes, Improving generation of MLPs with sliding mode control and the Levenberg-Marquardt algorithm, Neurocomputing, Vol. 70, (2007), 1342-1347.

[13] G. Lera and M. Pinzolas, Neighborhood Based Levenberg-Marquardt Algorithm for Neural Network Training, IEEE Trans. on Neural Networks, Vol. 13, No. 5, (2002), 1200-1203.

[14] Bogdan Mo Wilamowski and Hao Yu, Improved Computation for Levenberg-Marquardt Training, IEEE Trans. Neural Networks, Vol. 21, No. 6, (2010), 930-937.

[15] Lai-Wan Chan and Chi-Cheong Szeto, Training Recurrent Network with Block-Diagonal Approximated Levenberg-Marquardt Algorithm, Proc. IJCNN, Vol. 3, (1999), 1521-1526.

[16] J. R. Alvarez-Sanchez, Injecting Knowledge into the Solution of the Two-Spiral Problem, Neural Computing & Applications, Vol. 8, (1999), 256-272.

[17] Kur Hornik, Maxwell Stinchcombe and Halber White, Multilayer Feedforward networks are universal approximators, Neural Networks, Vol. 2, (1989), 359-366.

[18] D. Aha, A. Asuncion and D. Newman, UCI Machine Learning Repository, Availabe: http://archive.ics.uci.edu/ml/index.html

[19] Jonathan J. Hull, A Database for Handwritten Text Recognition Research, IEEE Tarns. on Pattern Analysis and Machine Intelligence, Vol. 16, No. 5, (1994), 550-554.

Young-Tae Kwak received the B.S., M.S., and Ph.D. degrees in computer engineering from the Chungnam National University, Republic of Korea, in 1993, 1995, and 2001, respectively. He joined the faculty of the Chonbuk National University in 2002. His research interests include pattern recognition and neural networks.

Heeseung Jo received the B.S. degree in computer science from the Sogang University, Republic of Korea, in 2000, and the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), in 2010. He joined the faculty of the Chonbuk National University in 2010. His research interests cover computer hardware and system software.