

# C++ Implementation of Neural Networks Trainer

Hao Yu\* and Bogdan M. Wilamowski\*

\* Electrical and Computer Engineering, Auburn University, Alabama, US

hzy0004@auburn.edu wilam@ieee.org

**Abstract**—The paper is going to introduce a revised C++ version of neural network trainer (NNT) which is developed based on neuron by neuron computation. Besides traditional error back propagation (EBP) algorithm, two improved version of Levenberg Marquardt (LM) algorithm and a newly developing algorithm are also implemented. The software can handle not only conventional multilayer perceptron networks, but also arbitrarily connected neuron networks. Comparing with the original NNT developed based on MATLAB [18], the revised version can handle much larger networks and the training speed is also improved as 50 to 100 times faster. Several practical applications are presented to show the power of this training tool. The software is available for everyone on the website.

## I. INTRODUCTION

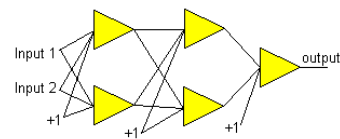
Neural networks have developed quickly in decades, since the invention of EBP algorithm [1]. There are lots of related industrial applications, especially in nonlinear control [2-4], VLSI design [5-7] and data classification [8-9].

Still, EBP algorithm is the most popular training method, however, it is also known as an inefficient one, for its slow convergence and low stability. In order to overcome those disadvantages, lots of improvements are developed based on EBP algorithm [10-12], and some of them, such as Quickprop algorithm [13] and Resilient EBP [14], really work well.

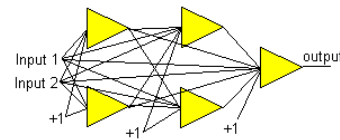
Second order algorithms, such as Newton algorithm and LM algorithm [15], are famous for their fast training. The main reason is that their training step sizes are “naturally” estimated in different directions according to instantaneous error surfaces, which makes them have a perfect match with gradient. As a successful combination of EBP algorithm and Newton algorithm, LM algorithm is regarded as the most efficient training method currently and there are also many good modifications [16-17] based on LM algorithm for better training.

Beside algorithms, neural structure is also an important aspect to measure the training efficiency. Normally, there is no defined method to find proper structures for specified problems and people have to try different networks before they get one solution. Although networks can be designed arbitrarily, the commonly used ones are showed in Fig. 1. Standard multilayer perceptron (MLP) networks (see Fig. 1(a), 15 weights) are broadly accepted in practical applications because they are regular and easy to realize by programming. MLP with full connections among layers (MLP-FCL) structure (see Fig. 1(b), 23 weights) are much more powerful than the standard MLP networks. For example, to deal with parity-N problems, it needs  $N+1$  neurons for standard MLP networks to get solutions, while MLP-FCL structures can solve it with

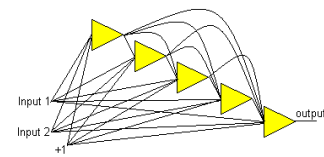
only  $N/2+1$  neurons (see Fig. 2). Fully connected neuron (FCN) networks (see Fig. 1(c), 25 weights) are the most complicated and powerful networks, but they are hard to training because of too many layers. Obviously, the latter two types of networks are better choices for efficient training, but they also require more challenging computation.



(a)  $2=2=1$  standard MLP network

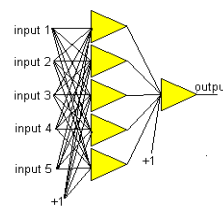


(b)  $2=2=1$  MLP-FCL structure

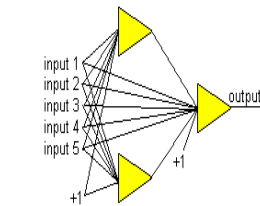


(c) 5 neurons in FCN network

Figure 1. Two inputs and five neurons in commonly used structures



(a)  $5=5=1$  MLP network



(b)  $5=2=1$  MLP-FCL structure

Figure 2. Smallest structures required for parity-5 problem

Currently, there are some excellent tools for neural networks training, such as the MATLAB Neural Network Toolbox (MNNT) and Stuttgart Neural Network Simulator (SNNS). The MNNT can do both EBP and LM computation, but only for standard MLP networks. Furthermore, it's also well known that MATLAB is very inefficient in executing “for” loop, which may slow down the training process. SNNS can handle FCN networks well, but the training methods it contains are all developed based on EBP algorithm, such as Quickprop algorithm and Resilient EBP, which makes the training still somewhat slow.

In this paper, the revised version of neural network

trainer (NNT) is introduced as a powerful training tool. It contains EBP algorithm, LM algorithm, improved LM algorithm and a new algorithm. Based on neuron by neuron computation, all those algorithms can handle arbitrarily connected neuron (ACN) networks. Comparing with the former MATLAB version [18], the revised one is supposed to perform more efficient and stable training.

In the section II of this paper, the neuron by neuron (NBN) computation is described in details as the fundamental of the revised NNT. Section III introduces the functions of the new software. Some experimental results are presented in section IV as practical applications of the revised NNT.

## II. NEURON BY NEURON COMPUTATION

### A. Computational fundamentals

The update rule of EBP algorithm could be described as

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha \mathbf{g}_k \quad (1)$$

where  $\mathbf{w}$  is the weight vector,  $\alpha$  is the learning constant (step size),  $k$  is the iteration index,  $\mathbf{g}$  is gradient vector and its elements could be calculated by

$$g_i = \frac{\partial E}{\partial w_i} \quad (2)$$

where  $E$  is the sum squared error (SSE) and it's defined as

$$E(\mathbf{x}, \mathbf{w}) = \sum_{p=1}^P \sum_{m=1}^M (o_{pm} - d_{pm})^2 = \sum_{p=1}^P \sum_{m=1}^M e_{pm}^2 \quad (3)$$

where  $\mathbf{x}$ ,  $\mathbf{d}$ ,  $\mathbf{o}$  and  $\mathbf{e}$  are input vector, desired output vector, actual output vector and error vector respectively;  $P$  is the number of training patterns and  $M$  is the number of outputs.

Insert (3) into (2)

$$g_i = \frac{\partial E}{\partial w_i} = 2 \sum_{p=1}^P \sum_{m=1}^M \left( \frac{\partial e_{pm}}{\partial w_i} e_{pm} \right) \quad (4)$$

Derived from EBP algorithm and Newton algorithm, the update rule of LM algorithm is [15]

$$\mathbf{w}_{k+1} = \mathbf{w}_k - (\mathbf{H}_k + \mu \mathbf{I})^{-1} \mathbf{g}_k \quad (5)$$

where  $\mu$  is the combination coefficient,  $\mathbf{H}$  is the Hessian matrix and its element could be defined as

$$h_{ij} = \frac{\partial^2 E}{\partial w_i \partial w_j} \quad (6)$$

In order to avoid calculating the second order derivatives of SSE, Jacobian matrix  $\mathbf{J}$  is introduced

and its element could be computed by

$$J_{mpi} = \frac{\partial e_{mp}}{\partial w_i} \quad (7)$$

Combine (4) and (7), gradient vector could be calculated by

$$\mathbf{g}_k = 2 \mathbf{J}_k^T \mathbf{e}_k \quad (8)$$

Replace the SSE in (6) by (3)

$$h_{ij} \approx 2 \sum_{p=1}^P \sum_{m=1}^M \left( \frac{\partial e_{pm}}{\partial w_i} \frac{\partial e_{pm}}{\partial w_j} \right) \quad (9)$$

Combine (7) and (9), Hessian matrix could be computed by

$$\mathbf{H}_k \approx 2 \mathbf{J}_k^T \mathbf{J}_k \quad (10)$$

With the (8) and (10), it is clear that both gradient vector and Hessian matrix can be calculated by Jacobian matrix.

Integrate (8) with (1), the update rule of EBP algorithm could be modified as

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha \mathbf{J}_k^T \mathbf{e}_k \quad (11)$$

where the constant “2” in (8) is combined into the learning constant  $\alpha$ , and the same combination will be taken in the followed reasoning.

At the same time, insert both (8) and (10) into (5), the update rule of LM algorithm is modified as

$$\mathbf{w}_{k+1} = \mathbf{w}_k - (\mathbf{J}_k^T \mathbf{J}_k + \mu \mathbf{I})^{-1} \mathbf{J}_k^T \mathbf{e}_k \quad (12)$$

So far, the new update rules of both EBP algorithm (11) and LM algorithm (12) are obtained, and they are used for EBP and LM training in the revised version of NNT. Obviously, the computation of Jacobian matrix  $\mathbf{J}$  is the key challenge.

### B. Neuron by neuron computation

The purpose of NBN computation is to calculate Jacobian matrix, so as to perform EBP and LM training with (11) and (12) separately.

Instead of computing the derivatives of error vector in (7), a similar concept “sensitivity”, which is used in traditional computation of EBP algorithm, is introduced here to simplify the computation. The sensitivity vector  $\mathbf{S}$  is defined as

$$S_{im} = \frac{\partial o_m}{\partial net_i} \quad (13)$$

where  $S_{im}$  presents the propagation of error from the output of neuron  $m$  to the input of neuron  $i$ . if  $i$  is equal to

m, then  $S_{ii}=s_i$ , where  $s_i$  is the slope of neuron  $i$ .

Normally, error is defined as actual output minus desired output. In this case, with (7), for the output  $m$  at a given pattern, the row element of Jacobian matrix could be described as

$$J_{aim} = \frac{\partial e_m}{\partial w_{ai}} = \frac{\partial o_m}{\partial net_i} \frac{\partial net_i}{\partial w_{ai}} = S_{im} node_{ai} \quad (14)$$

where  $w_{ai}$  is the weight on the  $a$ th input of neural  $i$ , and  $node_i$  is a row vector defined to describe all the inputs of neuron  $i$ , so  $node_{ai}$  is the  $a$ th input of neuron  $i$ .

In order to get the sensitivity vector  $\mathbf{S}$ , both forward computation and backward computation are needed. Forward computation is used to propagating signal forwardly, while backward computation backpropagates error in a reversed direction. For better illustration, the FCN network showed in Fig. 3 is given as an example.

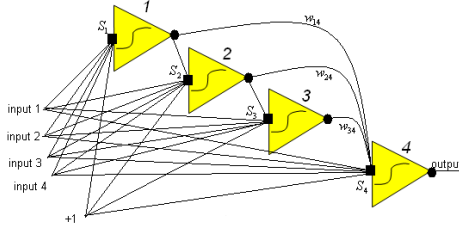


Figure 3. 4 neurons in FCN networks, with 4 inputs and 1 output

#### (1) Forward computation

In the forward calculation, the neurons which connect to the data inputs only are processed first, so their outputs can be used as the inputs for further computation. In this example, neuron 1 is the first. For neuron 1, compute its net, followed with the slope  $s_1$  and output  $o_1$ . For neuron 2, combine the output  $o_1$  into its inputs firstly, and then calculate slope  $s_2$  and output  $o_2$ . With the same process, analyzing neuron by neuron, all the slopes and outputs could be computed, as vectors  $\mathbf{s}=\{s_1, s_2, s_3, s_4\}$  and  $\mathbf{o}=\{o_1, o_2, o_3, o_4\}$ .

The forward computation is accorded with the direction of input signals transporting, and the elements of vector  $\mathbf{o}$  could be showed as the circles in Fig. 3 as the nodes for forward computation.

#### (2) Backward computation

The backward computation is designed to trace the feedback of errors, so as to get the elements of sensitivity vector  $\mathbf{S}$ , which are showed as rectangles in Fig. 3 as the nodes for backward computation.

Reversely, the backward computation is processed in a totally different direction from the forward computation. In the example, the order of forward computation is 1234, so backward computation should accord with the series 4321. For neuron 4, the sensitivity of neuron 4 itself is  $S_4=s_4$  and it's already obtained in forward computation. Then the sensitivity  $S_4$  is delivered to neuron 3, 2 and 1, through their interconnections. At the mean time, this sensitivity is propagated by related weights. After arrived neuron 3, the sensitivity  $S_4$  from neuron 4 is multiplied by the slope of neuron 3, so as to get the sensitivity at neuron 3 and it is  $S_3=S_4w_{34}s_3$ . With the same processing, all the elements of sensitivity vector  $\mathbf{S}=\{S_1, S_2, S_3, S_4\}$  could be calculated. One should take care that sensitivities at

neuron 3 and 4 are both delivered to neuron 2, so both of them should be calculated and summed when processing neuron 2, so as neuron 1.

After obtaining sensitivity vector  $\mathbf{S}$  by backward computation, and the output vector  $\mathbf{o}$  from forward computation, all the elements of Jacobian matrix could be calculated by (14). Furthermore, with Jacobian matrix, EBP training and LM training could be realized based on (11) and (12), separately.

The whole process above is called NBN computation because all neurons are analyzed one by one, in both forward and backward computation.

### III. INTRODUCTION OF THE REVISED NNT

The revised NNT is developed based on Visual C++ 6.0 for neural network training. It's derived from the original MATLAB version, but it's improved a lot. As mentioned above, the low executing efficiency of "for" loop in MATLAB may slow down the training, but this problem is not existed at all in C++ programming. Let us have a simple example as the comparison. Training parity-3 problem with EBP algorithm, with 2 neurons in FCN network, the MATLAB code takes more than 2 minutes to be convergent, while the C++ code converges in less than 2 seconds.

Besides time efficient, the revised version also has a more reasonable user interface for easy operation (see Fig. 4), and it provides as much training information as possible.

Furthermore, some new algorithms and strategies for better training are introduced to strengthen the training ability of the revised NNT. Followed, new elements will be introduced, while for the similar information, it could be referred in [18].

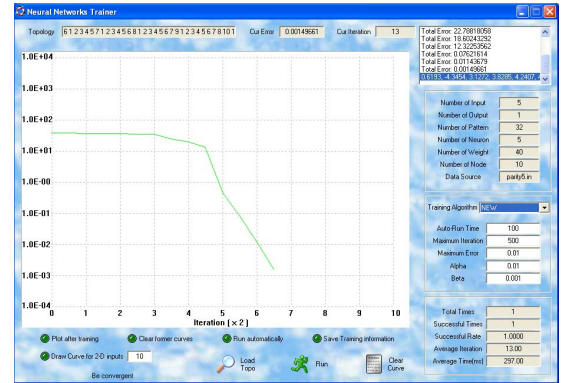


Figure 4. The user interface of the revised NNT

#### A. Implemented algorithms

There are four training algorithms and they are EBP algorithm, LM algorithm, NBN algorithm and a new developing algorithm named "NEW" temporarily. With the drop-list box, users can select required algorithms for training conveniently, at the same time, all training parameters will be initialed with the values used for last training. The four training algorithms are introduced as:

**EBP:** Improved EBP algorithm with momentum. The momentum here is designed to adjust weight vector using the information from the last iteration, so, based on (11), the update rule could be modified as

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha \mathbf{J}_k \mathbf{e}_k + \eta \mathbf{w}_{k-1} \quad (15)$$

where  $\eta$  is the momentum.

With the momentum, training becomes much faster and more stable. Table I presents the training results of EBP algorithm with different values of momentum. All neurons are in FCN networks and the iterations are averaged for 100 times.

TABLE I. Tests of EBP algorithm with momentum

Problems Parameters	Parity-3 problem		Parity-5 problem	
	$\eta=0$	$\eta=0.01$	$\eta=0$	$\eta=0.001$
Neurons	2	2	8	8
Iterations	1708.4	153.6	/	26261.2
Successful Rate	99%	100%	0%	20%

**LM:** original LM algorithm.

**NBN:** modified LM algorithm. It reduces the memory for Jacobian matrix storage largely by a newly developing decomposed computation for matrix multiplication based on (8) and (10). In this case, it is much more efficient on both training time and memory than LM algorithm, in large-sized pattern training. Especially, for problems with Jacobian matrix larger than 3GB, which is out of the limitation of Windows' compilers, this method can handle them well. Table II shows the comparison between LM algorithm and NBN algorithm. All neurons are in FCN networks and the time results are averaged for 100 times.

TABLE II. Training time and memory comparison between LM algorithm and NBN algorithm

Problems	Parity-7	Parity-11	Parity-13
Patterns	128	2048	8192
Neurons	5	13	14
Averaged computing time(ms)			
LM algorithm	113.9	26660.9	200531.8
NBN algorithm	110.9	16160.7	29356.3
Memory test (MB)			
LM algorithm	11.88	32.63	50.55
NBN algorithm	11.46	19.82	20.71

**NEW:** a newly developing algorithm derived from EBP algorithm and Newton algorithm. It's designed to do only one matrix inversion for each iteration during training. With this property, it's supposed to compute faster than LM algorithm. At the same time, by inheriting gradient searching ability from EBP algorithm, the NEW algorithm also can perform stable training.

### B. Strategies for better training

The purpose of the NNT is to make training convergent as quickly as possible. In order to achieve it, two strategies are introduced for better training.

#### (1) Improved slope

The famous "flat spot" problem is that if the slope of the neuron is very small, while the error is huge, then the training will be pushed into the saturate region and get stuck. In order to avoid being trapped in saturate region, an equivalent slope is introduced instead of the derivative of activation function (see Fig. 5), and it is calculated by

the followed algorithm.

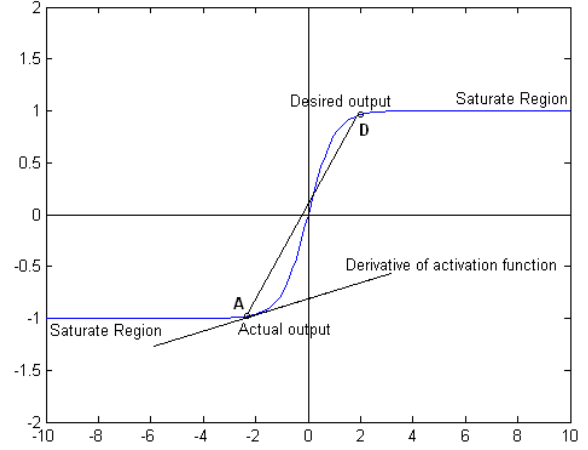


Figure 5. The "flat spot" problem in sigmoidal activation function

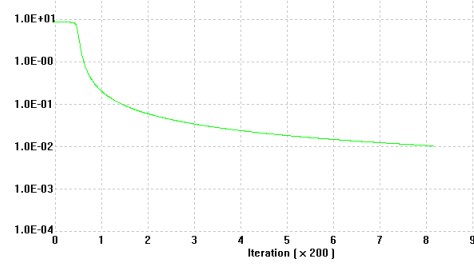
Algorithm for slope improvement [20]:

```

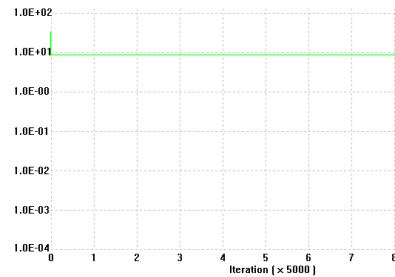
{
    Compute derivative of activation function;
    Compute the slope of line AD  $S_{AD}$ ;
    If  $derivate > S_{AD}$ 
        improved slope = derivative;
    else
        improved slope =  $S_{AD}$ ;
}

```

In order to test the improvement above, the "worst case" training is performed. The "worst case" training is to use a proper training result as the initial status to train the same patterns with all outputs reversed. For parity-3 problem, the training result of "worst case" is showed in Fig. 6.



(a) A proper training result



(b) "Worst case" training without slope improved



(c) “Worst case” training with slope improved

Figure 6. Test the improved slope by “worst case” training

From the results showed in Fig. 6, it is clear that the improved slope works well for “worst case” training.

## (2) Self-aware

Local minima problem is an unavoidable problem for gradient based training algorithms. The algorithms in the revised NNT will all failed in training, if they are trapped in local minima, even for the NEW algorithm which has the best gradient searching ability in the four. In order to improve this situation, a self-aware function is added in the NEW algorithm. By monitoring the relationship of Jacobian matrix and error vector, it can be aware if the training is trapped into a local minimum. If it's trapped, the training will be reset by randomly generating a group of new weights. Technically, it is not a scientific solution, but it can make the software stronger. In the software, this function is optional.

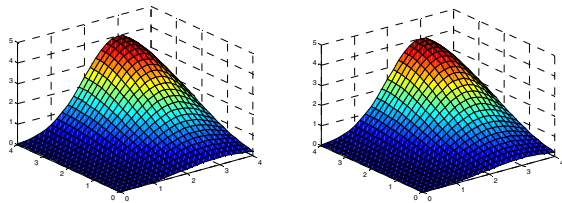
## IV. PRACTICAL APPLICATIONS

As introduced in the first section, neural networks are broadly used in various fields, and several practical applications are presented below to test the training ability of revised NNT.

### A. Function approximation

Function approximation is usually used in nonlinear control realm of neural networks, for control surface prediction. In order to approximate the function showed below, 25 points are picked out from 0 to 4 as the training patterns. With only 4 neurons in FCN networks, the training result is presented in Fig. 7.

$$z = 4 \exp(-0.15(x-4)^2 - 0.5(y-3)^2) + 10^{-9} \quad (16)$$



(a) Desired surface

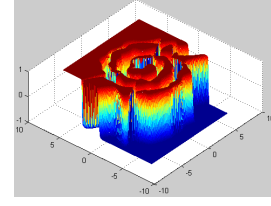
(b) Neural prediction

Figure 7. Function approximating result with SSE = 0.00515954

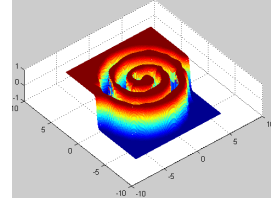
### B. Two-spiral problem

It is known that two-spiral problem is presented as an evaluation of both training algorithms and neural structures [19]. With 16 neurons in 2=5=5=5=1 MLP-FCL structure, two-spiral problem is solved by all 4 algorithms in NNT. Actually, the other three second order

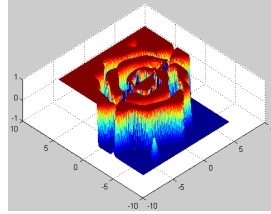
algorithms can handle the problem with less neurons and the training results are presented in Fig. 8.



(a) EBP algorithm with 2=5=5=5=1 MLP-FCL structure



(b) NBN algorithm with 2=5=5=5=1 MLP-FCL structure



(c) NEW algorithm with 8 neurons in FCN networks

Figure 8. Solutions of Two-spiral problem

For EBP algorithm, it spends nearly 80000 iterations to get the solution, while other three algorithms only need less than 400 iterations to get solutions. For NEW algorithm, the least neuron number needed for two-spiral problem is 8 (see Fig. 8(c)). So, it is clear that second order algorithms are exactly more powerful than EBP algorithm (with momentum).

### C. Parity-n problem

Parity-n problem is one of the hardest training problems in neural networks. With the self-aware ability, the NEW algorithm can handle it well (see Table III). All the neurons are in FCN networks and testing results are averaged in 100 times.

TABLE III. Training results of parity-n problems with NEW algorithm

Problems Parameters	Parity 3	Parity 5	Parity 7	Parity 8	Parity 11
Neurons	2	3	4	5	6
Iterations	7.8	19.51	33.18	101.32	100.12
Time (ms)	4.98	23.35	159.78	1152.56	15371.35
Successful rate	100%	98%	99%	99%	98%

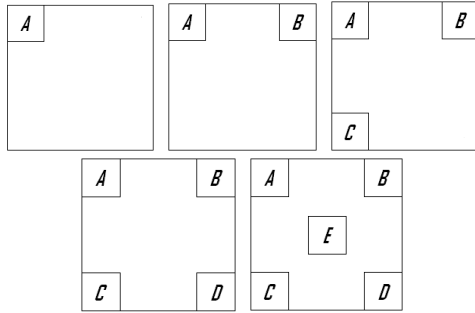
The results in the table above are tested in the strictest situation, which means the number of neurons is just enough for each problem. The more neurons are used, the better the results will be.

### D. Multi-output problems

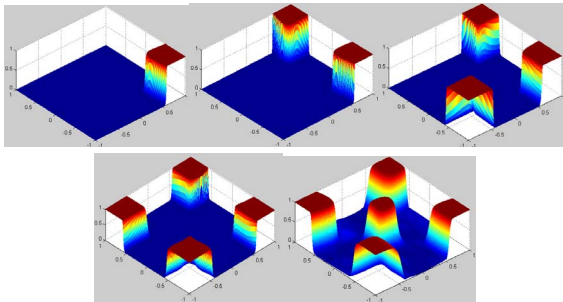
Several multi-output problems are showed in Fig. 9(a), and their purposes are to separate the different parts



marked by different letters. Since the number of letters is equal to the number of outputs, so those problems are used to test the ability of the revised NNT for multi-output problems.



(a) Problems with 1-5 outputs



(b) Training results of related problems in (a)

Figure 9. Solutions of multi-output training with SSE < 0.001

## V. CONCLUSION

In this paper, a neuron by neuron computation is introduced to implement both first order and second order algorithms. Based on the computation, the revised NNT is developed by C++ for neural network training. Although derived from the former MATLAB version, the revised NNT is injected with lots of new elements and shows its advantages for fast and stable training. As an all-round training tool, the revised NNT contains both first order and second order training algorithms; at the same time, it can handle not only traditional MLP networks, but also ACN networks well. Furthermore, with several improvements in training algorithms, and strategies for “flat spot” and local minima problems, the revised NNT shows its excellent training ability. With these properties, the revised NNT did well jobs in the practical applications provided in the paper. With sufficient testing results presented in the paper and other tests, it could be safe to say that the revised NNT is a good tool for neural network training. The revised NNT software is available at <http://www.eng.auburn.edu/users/wilambm/nnt/>, and it will be updated from time to time.

## REFERENCE

- [1] Rumelhart D. E., G. E. Hinton, R. J. Williams, “Learning representations by back-propagating errors”. *Nature*, vol. 323, pp. 533-536, 1986.
- [2] J. A. Farrell, M. M. Polycarpou, “Adaptive Approximation Based Control: Unifying Neural, Fuzzy and Traditional Adaptive Approximation Approaches”, *IEEE Trans. on Neural Networks*, vol. 19, no. 4, pp. 731-732, April 2008.
- [3] A.Y. Alanis, E.N. Sanchez, A.G. Loukianov, “Discrete-Time Adaptive Backstepping Nonlinear Control via High-Order Neural Networks,” *IEEE Trans. on Neural Networks*, vol. 18, no. 4, pp. 1185-1195, April 2007.
- [4] G. Colin, Y. Chamaillard, G. Bloch, G. Corde, “Neural Control of Fast Nonlinear Systems—Application to a Turbocharged SI Engine With VCT,” *IEEE Trans. on Neural Networks*, vol. 18, no. 4, pp. 1101-1114, April 2007.
- [5] K. Cameron, A. Murray, “Minimizing the Effect of Process Mismatch in a Neuromorphic System Using Spike-Timing-Dependent Adaptation,” *IEEE Trans. on Neural Networks*, vol. 19, no. 5, pp. 899-913, May 2008.
- [6] G. Indiveri, E. Chicca, R. Douglas, “A VLSI array of low-power spiking neurons and bistable synapses with spike-timing dependent plasticity,” *IEEE Trans. on Neural Networks*, vol. 17, no. 1, pp. 211-221, Jan 2006.
- [7] A. Gopalan, A.H. Titus, “A new wide range Euclidean distance circuit for neural network hardware implementations,” *IEEE Trans. on Neural Networks*, vol. 14, no. 5, pp. 1176- 1186, May 2003.
- [8] B. Vigdor, B. Lerner, “Accurate and Fast Off and Online Fuzzy ARTMAP-Based Image Classification With Application to Genetic Abnormality Diagnosis,” *IEEE Trans. on Neural Networks*, vol. 17, no. 5, pp. 1288-1300, May 2006.
- [9] M. Kyperountas, A. Tefas, I. Pitas, “Weighted Piecewise LDA for Solving the Small Sample Size Problem in Face Verification,” *IEEE Trans. on Neural Networks*, vol. 18, no. 2, pp. 506-519, Feb 2007.
- [10] Y. Yamamoto, P.N. Nikiforuk, “A new supervised learning algorithm for multilayered and interconnected neural networks,” *IEEE Trans. on Neural Networks*, vol. 11, no. 1, pp. 36-46, Jan 2000.
- [11] J.-P. Martens, N. Weymaere, “An equalized error backpropagation algorithm for the on-line training of multilayer perceptrons,” *IEEE Trans. on Neural Networks*, vol. 13, no. 3, pp. 532-541, March 2002.
- [12] K.C. Tan, H.J. Tang, “New dynamical optimal learning for linear multilayer FNN,” *IEEE Trans. on Neural Networks*, vol. 15, no. 6, pp. 1562-1570, June 2004.
- [13] Fok Hing Chi Tivive, A. Bouzerdoum, “Efficient training algorithms for a class of shunting inhibitory convolutional neural networks,” *IEEE Trans. on Neural Networks*, vol. 16, no. 3, pp. 541-556, March 2005.
- [14] M. Riedmiller, H. Braun, “A direct adaptive method for faster backpropagation learning: The RPROP algorithm”. *Proc. International Conference on Neural Networks*, San Francisco, CA, 1993, pp. 586-591.
- [15] K. Levenberg, “A method for the solution of certain problems in least squares”. *Quarterly of Applied Mathematics*, 5, pp. 164-168, 1944.
- [16] Joel D. Hewlett, Bogdan M. Wilamowski, Gunhan Dunder, “Optimization using a modified second-order approach with evolutionary enhancement”. *IEEE Trans. Ind. Electron.*, vol. 55, no. 9, pp. 3374-3380, Sep. 2008.
- [17] Bogdan M. Wilamowski, N. Cotton, O. Kaynak, G. Dunder, “Method of computing gradient vector and Jacobian matrix in arbitrarily connected neural networks”. *Proc. IEEE ISIE*, Vigo, Spain, June, 4-7, pp. 3298-3303, 2007.
- [18] B. M. Wilamowski, N. Cotton, J. Hewlett, O. Kaynak, “Neural network trainer with second order learning algorithms”. *Proc. International Conference on Intelligent Engineering Systems*, June 29 2007-July 1 2007, pp. 127-132.
- [19] J. R. Alvarez-Sanchez, “Injecting knowledge into the solution of the two-spiral problem”. *Neural Compute and Applications*, Vol. 8, pp. 265-272, 1999.
- [20] Bogdan M. Wilamowski, “Midified EBP algorithm with instant training of the hidden layer”. *Proc. in 23rd International Conference on Industrial Electronics, Control, and Instrumentation*, vol. 3, pp.1098-1101, 1997.