Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
Neurocomputing 56 (2004) 101 – 121

NEUROCOMPUTING
www.elsevier.com/locate/neucom

A pseudoinverse learning algorithm for feedforward neural networks with stacked generalization applications to software reliability growth data

Ping Guo[a], Michael R. Lyu[b,*]

[a]*Department of Computer Science, Beijing Normal University, Beijing 100875, People's Republic of China*

[b]*Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, NT, Hong Kong, People's Republic of China*

Abstract

A supervised learning algorithm, Pseudoinverse Learning Algorithm (PIL), for feedforward neural networks is developed. The algorithm is based on generalized linear algebraic methods, and it adopts matrix inner products and pseudoinverse operations. Incorporating with network architecture of which the number of hidden layer neuron is equal to the number of examples to be learned, the algorithm eliminates learning errors by adding hidden layers and will give an exact solution (perfect learning). Unlike the existing gradient descent algorithm, the PIL is a feedforward only, fully automated algorithm, including no critical user-dependent parameters such as learning rate or momentum constant. The algorithm is tested on case studies with stacked generalization applications to software reliability growth data. The results indicate that the proposed algorithm is very efficient for the investigation on the computation-intensive generalization techniques.
© 2003 Elsevier B.V. All rights reserved.

Keywords: Feedforward neural networks; Generalized linear algebra; Pseudoinverse learning algorithm; Fast learning; Generalization

* Corresponding author.
E-mail address: lyu@cse.cuhk.edu.hk (M.R. Lyu).

0925-2312/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S0925-2312(03)00385-0

1. Introduction

Multilayer feedforward neural networks have already been found to be successful for various supervised learning tasks. Both theoretical and empirical studies have shown that the networks are of powerful capabilities for pattern classification and universal approximation [3,8,12]. Several adaptive learning algorithms for multilayer feedforward neural networks have recently been proposed [1,9,14,17]. Most of these algorithms are based on variations of the gradient descent algorithm, for example, back propagation (BP) algorithm [17]. These algorithms usually have a poor convergence rate and sometimes fall into local minima instead of global minima [20]. Convergence to local minima can result from the insufficient number of hidden neurons as well as improper initial weight settings. However, slow convergence rate is a common problem of the gradient descent methods, including the BP algorithm. Various attempts have been made to speed up learning, such as proper initialization of weights to avoid local minima, and an adaptive least-square algorithm using the second-order terms of error for weight updating [11]. There is another drawback for most gradient descent algorithms, namely, "learning factors problems", such as learning rate and momentum constant. The values of these parameters are often crucial for the success of the algorithm. Most gradient descent methods depend on these parameters which have to be specified by the user, as no theoretical basis for choosing them exists. Furthermore, for applications which require high precision output, such as the prediction of chaotic time series, the known algorithms are often too slow and inefficient. In some cases, for example, like *stacked generalization* [19] which requires to train a lot of networks to get level-1 training samples, it is very computation-time consuming when adopting BP algorithm to perform the required task. Therefore, it is worthwhile to seek new algorithms which are suitable for the applications that require high precision output, whereas the network structure is less important.

In order to reduce training time and investigate the generalization properties of learned neural networks, this paper presents a *Pseudoinverse Learning algorithm* (PIL), which is a feedforward-only algorithm. Learning errors are transferred forward and the network architecture is established. The previously trained weights in the network are not changed. Hence, the learning errors are minimized separately on each layer instead of globally for the network as a whole. The learning accuracy is determined by the number of layer. By adding layers to eliminate errors, all examples of a training set can be perfectly learned. From a mathematical computational point of view, the algorithm is based on generalized linear algebraic method and employs matrix inner products and pseudoinverse operations.

The paper is organized as follows: Section 2 discusses the network structure and presents the proposed algorithm. Section 3 describes how the algorithm can add or delete a training sample efficiently. In Section 4, several numerical examples are presented. The investigation of stacked generalization is provided in Section 5. Section 6 discusses the characteristics of the proposed algorithm. Finally, summary of the paper is given in Section 7.

2. The network structure and learning algorithm

2.1. The network structure

Let us consider a multilayer feedforward neural network. The network has one input layer, one output layer and several hidden layers. The first layer with n neurons is the input layer including last neuron being a bias neuron of constant output. The last layer with m neurons is the output layer. The number of hidden layers depends on the desired learning accuracy and the training data set.

The weight matrix $\mathbf{W}^l$ connects layer l and layer $l+1$ with elements $w^l_{i,j}$. Element $w^l_{i,j}$ connects neurons i of layer l with neurons j of layer $l+1$. Note that the $\mathbf{W}^0$ matrix connects the input layer and the first hidden layer, whereas the $\mathbf{W}^{\mathrm{L}}$ matrix connects the last hidden layer and the output layer. We assume only the input layer has a bias neuron, while the hidden layer(s) and the output layer have no bias neuron. The nonlinear activation function is denoted as $\sigma(\cdot)$. For example, we can use the so-called sigmoidal function:

$$\sigma(x) = \frac{1}{1+\mathrm{e}^{-x}}, \tag{1}$$

whose output is in the range of (0,1), or a hyperbolic function

$$\sigma(x) = \tanh(x) = \frac{\mathrm{e}^{x}-\mathrm{e}^{-x}}{\mathrm{e}^{x}+\mathrm{e}^{-x}}, \tag{2}$$

whose output is in the range of $(-1,1)$ as an activation function.

Given a training data set $D=\{\mathbf{x}^i,\mathbf{o}^i\}_{i=1}^N$, let $(\mathbf{x}^i,\mathbf{o}^i)$ be the ith input–output training pair, where $\mathbf{x}^i=(x_1,x_2,\ldots,x_n)\in R^n$ is the input signal vector and $\mathbf{o}^i=(o_1,o_2,\ldots,o_m)\in R^m$ is the corresponding target output vector. For given N sets of input–output vector pairs as examples to be learned, we can summarize all given input vectors into a matrix $\mathbf{X}^0$ with N rows and $n+1$ columns. Here the last column of $\mathbf{X}^0$ is a bias neuron of constant value θ. Each row of $\mathbf{X}^0$ contains the signals of one input vector. Note $\mathbf{X}^0=[\mathbf{X}|\theta]$, where matrix $\mathbf{X}$ consists of all signal $\mathbf{x}^i$ as row vectors. All desired target output vectors are summarized into a matrix $\mathbf{O}$ with N rows and m columns. Each row of the matrix $\mathbf{O}$ contains the signals of one output vector $\mathbf{o}^i$.

The described networks are of multilayer perception type: They first compute an inner product of the incoming signals matrix with their respective weight matrix. Afterwards, an activation function is applied, producing the output of the neuron which is sent to all neurons of the following layer. In this designed network structure, the activation function is not applied to the output layer, so the last layer is linear.

Basically, the task of training the network means trying to find the weight matrix which minimizes the sum-square-error function:

$$E = \frac{1}{2N}\sum_{i=1}^{N}\sum_{j=1}^{m}\|g_j(\mathbf{x}^i,\Theta)-o^i_j\|^2, \tag{3}$$

where $g(\mathbf{x},\Theta)$ is a network mapping function and Θ is the network parameter set. Θ includes connection weight $\mathbf{W}$ and a bias parameter. In a three-layer structure case

$$g_j(\mathbf{x},\Theta)=\sum_{i=1}^{N} w^1_{i,j}\sigma_i\left(\sum_{l=1}^{n} w^0_{i,l}x_l+\theta_i\right), \tag{4}$$

where θ_i is a bias value for the network input.

For simplification, we can write the system error function in the matrix form:

$$E=\frac{1}{2N}\,\mathrm{Trace}[(\mathbf{G}-\mathbf{O})^{\mathrm{T}}(\mathbf{G}-\mathbf{O})]. \tag{5}$$

Propagating the given examples through the network, multiplying the output of layer l with the weights between layers l and $l+1$, and applying the nonlinear activation function to all matrix elements, we get:

$$\mathbf{Y}^{l+1}=\sigma(\mathbf{Y}^l\mathbf{W}^l) \tag{6}$$

and the network output should be

$$\mathbf{G}=\mathbf{Y}^{\mathrm{L}}\mathbf{W}^{\mathrm{L}}, \tag{7}$$

where we use superscript L to denote the last layer.

By examining the above equations and reformulating the task of training, the problem becomes

$$\mathrm{minimize}\|\mathbf{Y}^{\mathrm{L}}\mathbf{W}^{\mathrm{L}}-\mathbf{O}\|^2. \tag{8}$$

This becomes a linear least-square problem. If we can find the network weight parameter such that $\|\mathbf{Y}^{\mathrm{L}}\mathbf{W}^{\mathrm{L}}-\mathbf{O}\|^2=0$, we will have trained the neural network to learn all given examples exactly, that is, a perfect learning.

We focus our discussion on the last hidden layer. For the sake of convenience, in the following discussion we drop superscript index L in Eq. (7).

2.2. Existence of the solution

Now let us discuss the equation:

$$\mathbf{YW}=\mathbf{O},\quad \mathbf{W}\in R^{p\times m},\quad \mathbf{Y}\in R^{N\times p},\quad \mathbf{O}\in R^{N\times m}, \tag{9}$$

when $p<N$, the system is an *underdetermined* system. Notice that such a system either has no solution or has an infinitive number of solutions.

If $\mathbf{Y}\in R^{N\times N}$ is invertible and has been learned in $L-1$ layer, then the system of Eq. (9) is, in principle, easy to solve. The unique solution for the last layer weight matrix is $\mathbf{W}=\mathbf{Y}^{-1}\mathbf{O}$. If $\mathbf{Y}$ is an arbitrary matrix in $R^{N\times p}$, then it becomes more difficult to solve Eq. (9). There may be none, one or an infinite number of solutions depending on where $\mathbf{O}\in R(\mathbf{Y})$ space and whether $N-\mathrm{rank}(\mathbf{Y})>0$, where $R(\mathbf{Y})$ denotes the space spanned by the column vectors of $\mathbf{Y}$.

One would like to be able to find a matrix (or some matrices) $\mathbf{C}$, such that solution of Eq. (9) are of the form $\mathbf{CO}$. But if $\mathbf{O}\notin R(\mathbf{Y})$, then Eq. (9) has no solution.

In order to make our approach self-contained, we rewrite the relative linear algebra theorem in the following. The corresponding proof is from the Ref. [4].

Theorem 1. *The system* $\mathbf{YW} = \mathbf{O}$ *has a solution if and only if*

$$\mathrm{rank}([\mathbf{Y}, \mathbf{O}]) = \mathrm{rank}(\mathbf{Y}). \tag{10}$$

Proof. Let $\mathbf{S}$ denote the column space of $\mathbf{Y}$, and let $\mathbf{S}^*$ denote the column space of $[\mathbf{Y}, \mathbf{O}]$, then $\mathbf{YW} = \mathbf{O}$ has a solution if and only if $\mathbf{O}$ is in $\mathbf{S}$. But $\mathbf{O}$ is in $\mathbf{S}$ if and only if $\mathbf{S}$ and $\mathbf{S}^*$ have the same dimension, i.e., $\mathbf{Y}$ and $[\mathbf{Y}, \mathbf{O}]$ have the same rank. □

2.3. Pseudoinverse solution is the best approximation

We intend to use the pseudoinverse solution for finding weight matrices, as the theorem from linear algebra states that pseudoinverse solution is the best approximation solution for Eq. (9). It achieves a global minimum in the weight parameter space if the exact solution is reached.

Theorem 2. *Suppose that* $\mathbf{X} \in R^{p\times m}$, $\mathbf{A} \in R^{N\times p}$, $\mathbf{B} \in R^{N\times m}$, *then the best approximate solution of the equation* $\mathbf{AX} = \mathbf{B}$ *is* $\mathbf{X_0} = \mathbf{A}^+\mathbf{B}$ (*we use superscript* $+$ *to denote the pseudoinverse form of a matrix*).

Theorem 2 can be similarly derived from [4]. From the Theorem 2 we get:

Corollary 1. *The best approximate solution of* $\mathbf{AX} = \mathbf{I}$ *is* $\mathbf{X} = \mathbf{A}^+$.

Based on the above analysis, we try to find the output layer weight in the following way. Let $\mathbf{W} = \mathbf{Y}^+\mathbf{O}$, the learning problem becomes $\|\mathbf{YY}^+\mathbf{O} - \mathbf{O}\|^2 = 0$, where $\mathbf{Y}^+$ is the pseudoinverse of $\mathbf{Y}$. This is equal to finding the matrix $\mathbf{Y}$ so that $\mathbf{YY}^+ - \mathbf{I} = \mathbf{0}$, where $\mathbf{I}$ is the identity matrix. Now the task of training the network becomes that of managing to raise the rank of matrix $\mathbf{Y}$ up to a full rank. As soon as $\mathbf{Y}$ becomes a full rank matrix, $\mathbf{YY}^+$ will be equal to the identity matrix $\mathbf{I}$. Note that since we multiply $\mathbf{Y}$ on the right side by $\mathbf{Y}^+$, it only requires the right inverse of $\mathbf{Y}$ to exist, and $\mathbf{Y}^+$ is not necessary to be a two-sided inverse of $\mathbf{Y}$. This means that $\mathbf{Y}$ need not be a square matrix, but its number of columns should not be less than its number of rows. This condition requires that hidden neuron numbers be greater than or equal to N. If the condition is satisfied, we can find an exact solution for the weight matrix. In our network architecture design, we set the hidden neuron number to be equal to N. With this network structure, we can find the weight matrix which can exactly map to the training set.

2.4. The PIL algorithm

According to the above discussion, we first let the weight matrix $\mathbf{W}^0$ be equal to $(\mathbf{Y}^0)^+$ which is an $(n+1)\times N$ matrix. Then we apply a nonlinear activation function, that is, to compute $\mathbf{Y}^1 = \sigma(\mathbf{Y}^0\mathbf{W}^0)$, then compute $(\mathbf{Y}^1)^+$, the pseudoinverse of $\mathbf{Y}^1$, and so on. Because the algorithm is feedforward only, no error will propagate back to the preceding layer of the neural network, and we cannot use a standard error form $E = (1/2N)\,\mathrm{Trace}[(\mathbf{G}-\mathbf{O})^{\mathrm{T}}(\mathbf{G}-\mathbf{O})]$ to judge whether the trained network has reached the desired accuracy during the training procedure. Instead, we use the criterion $\|\mathbf{Y}^l \cdot (\mathbf{Y}^l)^+ - \mathbf{I}\|^2 < \mathbf{E}$. At each layer, we compute $\|\mathbf{Y}^l\mathbf{Y}^{l+} - \mathbf{I}\|^2$. If it is less than the desired error, we set $\mathbf{W}^{\mathrm{L}} = (\mathbf{Y}^{\mathrm{L}})^+\mathbf{O}$ and stop the training procedure. Otherwise, let $\mathbf{W}^l = (\mathbf{Y}^l)^+$, add another layer, and feed forward previous layer output to the next layer again, until we reach the required learning accuracy.

To use any nonlinear activation function in the hidden nodes is to utilize the nonlinearity of the function, and to increase the linear independency among the column (row) vectors or, equivalently, the rank of the matrix. It is proven that sigmoid functions of a hidden layer of the network can raise the dimension of the input space up to the number of the hidden neurons [18]. So through a nonlinear activating action, the rank of the transformed matrix will be raised layer by layer.

In this way, we get a feedforward-only algorithm which reduces learning errors on every layer. First we establish a two-layer neural network. If the given precision cannot be reached, a third layer is added to eliminate the remaining error. If the third added layer still cannot satisfy the desired accuracy, then another hidden layer is added again to reduce the learning errors, so on and so forth until the required accuracy is achieved. Mathematically, we can summarize the algorithm into the following steps:

Step 1. Set hidden neuron number as N, and let $\mathbf{Y}^0 = \mathbf{X}^0$.

Step 2. Compute $(\mathbf{Y}^0)^+ = \mathrm{Pseudoinverse}(\mathbf{Y}^0)$.

Step 3. Compute $\|\mathbf{Y}^l \cdot (\mathbf{Y}^l)^+ - \mathbf{I}\|^2$. If it is less than the given error E, go to step 6. If not, go on to the next step.

Step 4. Let $\mathbf{W}^l = (\mathbf{Y}^l)^+$. Feed forward the result to the next layer, and compute $\mathbf{Y}^{l+1} = \sigma(\mathbf{Y}^l\mathbf{W}^l)$.

Step 5. Compute $(\mathbf{Y}^{l+1})^+ = \mathrm{Pseudoinverse}(\mathbf{Y}^{l+1})$, set $l \leftarrow l+1$, and go to step 3.

Step 6. Let $\mathbf{W}^{\mathrm{L}} = (\mathbf{Y}^{\mathrm{L}})^+\mathbf{O}$.

Step 7. Stop training. The network mapping function is $\mathrm{G} = \sigma(\ldots\sigma(\sigma(\mathbf{Y}^0\mathbf{W}^0)\,\mathbf{W}^1)\ldots)\mathbf{W}^{\mathrm{L}}$.

3. Adding and deleting samples

The proposed algorithm is a batch-way learning algorithm, in which we assume that all the input signals are available at the time of training. However, in real-time applications, as a new input vector is given to the network, the weight matrix must be updated. Or, we need to delete a sample from the learned weight matrix. It is not efficient at all if we recompute the pseudoinverse function of a new weight matrix in the PIL algorithm. When we assign the hidden neuron number to be equal to the number of training samples, adding or deleting a sample is equivalent to adding or deleting a hidden neuron. Here we use neuron addition or deletion algorithms to efficiently compute the pseudoinverse matrix.

According to Griville's theorem [16], the first k columns of $\mathbf{Y}$ matrix consist of a submatrix, and the pseudoinverse function of this submatrix can be calculated from the previous $(k-1)$th pseudoinverse submatrix. That is,

$$\mathbf{Y}_k^+ = \begin{bmatrix} \mathbf{Y}_{k-1}^+(\mathbf{I} - \mathbf{y}_k\mathbf{b}^{\mathrm{T}}) \\ \mathbf{b}^{\mathrm{T}} \end{bmatrix}, \tag{11}$$

where the vector $\mathbf{y}_k$ is the kth column vector of the matrix $\mathbf{Y}$, while

$$\mathbf{b} = \begin{cases} (\mathrm{I} - \mathbf{Y}_{k-1}\mathbf{Y}_{k-1}^+)\mathbf{y}_k & \text{if } \|\mathbf{I} - \mathbf{Y}_{k-1}\mathbf{Y}_{k-1}^+\mathbf{y}_k\| \neq 0, \\ \dfrac{(\mathbf{Y}_{k-1}^+)^{\mathrm{T}}\mathbf{Y}_{k-1}^+\mathbf{y}_k}{1 + \|\mathbf{Y}_{k-1}^+\mathbf{y}_k\|^2} & \text{otherwise.} \end{cases} \tag{12}$$

It needs at most N times iterative cycles to obtain the pseudoinverse function of a matrix if there are N columns in this matrix. With this theorem, we can add the hidden neurons relatively easy to calculate the pseudoinverse matrix.

When a hidden neuron is deleted, the matrix needs to be updated. It is not efficient at all if we compute the pseudoinverse matrix from the beginning. Here we consider using bordering algorithm [5] to compute the inverse of the matrix. The formula for the pseudoinverse matrix can be obtained also from the partitioned matrix multiplication. Given the inverse of a $k \times k$ matrix, the method shows how to find the inverse of a $(k+1) \times (k+1)$ matrix, which is the same $k \times k$ matrix with an additional row and an additional column at its borders.

If the column vectors $\mathbf{y}_i$ in $\mathbf{Y}$ is linearly independent to each other, then by definition

$$\mathbf{Y}^+ = (\mathbf{Y}^{\mathrm{T}}\mathbf{Y})^{-1}\mathbf{Y}^{\mathrm{T}}. \tag{13}$$

Let $\mathbf{V} = \mathbf{Y}^{\mathrm{T}}\mathbf{Y}$, and we can calculate $\mathbf{V}_{k+1}^{-1}$ from the prior $\mathbf{V}_k^{-1}$ without inverting a matrix. The algorithm is

$$\mathbf{V}_{k+1}^{-1} = \begin{pmatrix} \mathbf{V}_k^{-1} + \frac{1}{\alpha}\mathbf{v}\mathbf{v}^{\mathrm{T}} & -\frac{1}{\alpha}\mathbf{v} \\ -\frac{1}{\alpha}\mathbf{v}^{\mathrm{T}} & \frac{1}{\alpha} \end{pmatrix}, \tag{14}$$

where $\mathbf{v} = \mathbf{V}_k^{-1}\mathbf{Y}_k^{\mathrm{T}}\mathbf{y}_{k+1}$, and $\alpha = \mathbf{v}_k^{\mathrm{T}}\mathbf{Y}_k^{\mathrm{T}}\mathbf{y}_{k+1}$.

When deleting a vector from the matrix, consider the original matrix containing $k+1$ vector pairs. The key step is to compute $\mathbf{V}_k^{-1}$ from $\mathbf{V}_{k+1}^{-1}$. When the $(k+1)$th pair is deleted from the matrix, we rewrite $\mathbf{V}_{k+1}^{-1}$ as four partitions:

$$\mathbf{V}_{k+1}^{-1} = \begin{pmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{b}^{\mathrm{T}} & c \end{pmatrix}, \tag{15}$$

where $\mathbf{A}$ is $k \times k$, $\mathbf{b}$ is $k \times 1$, and c is a scalar. By comparing with Eq. (14), it is apparent that $\mathbf{A} = \mathbf{V}_k^{-1} + \frac{1}{\alpha}\mathbf{v}\mathbf{v}^{\mathrm{T}}$, $\mathbf{b} = (1/\alpha)\mathbf{v}$, and $c = 1/\boldsymbol{\alpha}$. From these expressions, we find that the desired result is

$$\mathbf{V}_k^{-1} = \mathbf{A} - \frac{1}{c}\mathbf{b}\mathbf{b}^{\mathrm{T}}. \tag{16}$$

The inverse of the $k \times k$ matrix can now be calculated from the $(k+1) \times (k+1)$ matrix. This is equivalent to deleting the last hidden neuron and updating the weight matrix.

This algorithm is very effective in the case of leave-one-out cross-validation partition training samples (CVPS).[1] Because in each CVPS data set only one sample is different from the total sample set. We can first compute the inverse of the matrix which is obtained based on the full sample set, then at each time, move only one sample to the last column position, and use the above algorithm to delete this sample. In this way we can obtain the desired weight matrices on CVPS data sets efficiently.

4. Numerical examples

4.1. Function mapping examples

The algorithm is tested with the following function mapping examples. The total learning error is defined as in Eq. (3), while average learning error is

$$RMSE = \frac{1}{N}\frac{1}{m}\sqrt{\sum_{j}^{N}\sum_{i}^{m}(g_i^j - o_i^j)^2}, \tag{17}$$

where o_i^j and g_i^j are the desired network output and the actual output, respectively.

Example 1. Consider a nonlinear mapping problem of Sine function by neural networks. For the training set, 50 input–output signals (x_i, y_i) pairs are generated with $x_i = 2\pi * i/49$, for $i = 0, 1, 2, \ldots, 49$, and the corresponding y_i are computed using $y_i = \sin(x_i)$. The given learning error is $E = 10^{-7}$. If the learning error is $E < 10^{-7}$, we regard that perfect learning has been reached. For this problem, input neuron number is $n+1=2$ including the bias one, output neuron is $m=1$, and hidden layer neuron number is $N=50$. After using the PIL algorithm proposed above, we reach the perfect learning when two hidden layers are added. The trained network altogether has four layers including input and output layers. The actual learning error is $E=7.533\times10^{-18}$.

Example 2. This is the nonlinear mapping of eight input quantities x_i into three output quantities y_i problem, defined by Biegler-König and Bärmann in [2]:

$$\begin{aligned} y_1 &= (x_1 * x_2 + x_3 * x_4 + x_5 * x_6 + x_7 * x_8)/4.0,\\ y_2 &= (x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8)/8.0,\\ y_3 &= (1 - y_1)^{0.5}. \end{aligned} \tag{18}$$

All three functions are defined for values between 0 and 1 and they produce values in this range. For the training set, 50 sets of input signals x_i are randomly generated in the range of 0–1, and the corresponding y_i's are computed using the above equation. The desired learning error we require is $E=1.0\times10^{-7}$. When training is finished, only one hidden layer is added, and the actual learning error is $E = 3.573 \times 10^{-25}$ for this problem.

[1] The formal mathematical expression of CVPS will be shown in Section 5.

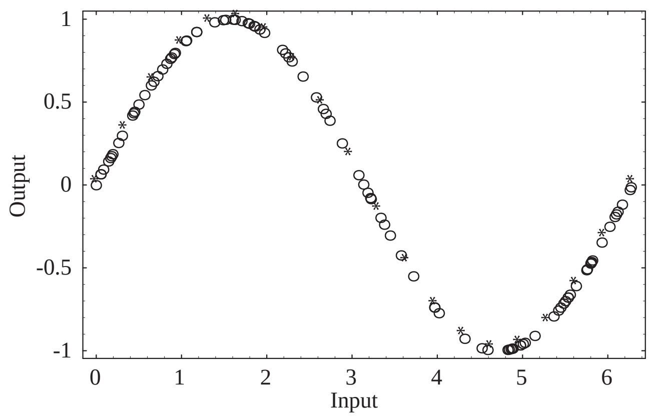

Fig. 1. The trained network output for $y = \sin(x)$ function mapping problem. "$*$" stands for training data output, while "o" stands for test data.

Table 1
Generalization ability test results

	Input range	N	Test set N_1	Generalized E	Generalized $RMSE$	Max deviation
Example 1	0–2π	20	100	0.00049	0.00031	0.0121
Example 2	0–1	20	100	0.23481	0.00228	0.1838
Example 3	$0-\pi$	20	100	0.00452	0.00095	0.0899

Given training error is 10^{-7}.

Example 3. Another functional mapping problem is $y = \sin(x)\cos(3x) + x/3$. Similar to Example 1, we use 50 examples with x_i in the region of 0–π to train the network. Perfect learning is reached after two hidden layers are added. Actual learning error is $E = 4.734 \times 10^{-17}$.

4.2. Generalization

We also tested the generalization ability of trained networks to forecast function values of examples not belonging to the training set. For Sine functional mapping, we train the network using 20 examples with $x_i = 2\pi * i/19$, for $i = 0, 1, 2, \ldots, 19$, and the corresponding y_i are computed using $y_i = \sin(x_i)$. After the network is trained, $N_1 = 100$ input signals x_i's are randomly generated within the range of 0–2π for testing the network, and the corresponding y_i's are computed using trained network. Fig. 1 shows the result, which is reasonably good. We have also tested Examples 2 and 3 with 20 examples training network and using 100 randomly generated input signals for testing. The results are shown in the Table 1 and Fig. 2. In the tables of this paper, Max deviation is defined as the maximum value of the difference between real network

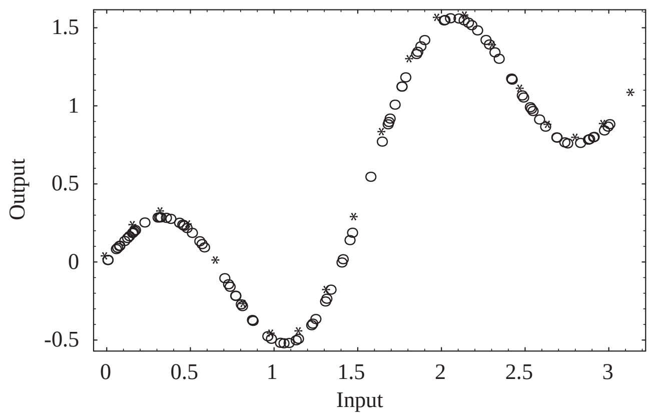

Fig. 2. The trained network output for $y = \sin(x)\cos(x) + x/3$ function mapping problem. "$*$" stands for training data, while "o" stands for test data.

output values and the desired values. Namely, Max deviation = maximum$|g_i^j - o_i^j|$, for $i = 1, 2, \dots m$ and $j = 1, 2, \dots, N_1$.

From Table 1, we see that Sine function mapping problem has the least generalized errors. For further investigating the proposed network architecture and learning algorithm's response to unlearned data, we present the following example.

Example 4. A Sine-like piecewise linear function is defined by

$$y = \begin{cases} x & \text{if } 0 \leqslant x < \pi/2, \\ \pi - x & \text{if } \pi/2 \leqslant x < 3\pi/2, \\ x - 2\pi & \text{if } 3\pi/2 \leqslant x \leqslant 2\pi. \end{cases} \tag{19}$$

First, 20 examples with $x_i = 2\pi * i/19$, for $i = 0, 1, 2, \dots, 19$, and the corresponding y_i's computed based on the above equation are used to train the networks. Then 100 input signals randomly generated in the range of 0–2π are applied to test the trained network. The result is shown in Fig. 3.

When using five set examples $\{(0,0), (\pi/2, 1), (\pi, 0), (3\pi/2, -1), (2\pi, 0)\}$ to train the network, we get a network structure which has one hidden layer with five hidden neurons. The learning error is $E = 3.314 \times 10^{-26}$. Afterward, 100 sets of input signals x_i randomly generated within the range of 0–2π are applied to test the network. The result is shown in Fig. 4. From Fig. 4, it can be seen that the network acts like a Sine function. It should be reminded that the architecture and weight matrices are the same for Examples 1 and 4 when using the above five examples. This result shows that the network forecast ability is better for smooth function when the data are in the range of training input space. When 50 examples with $x_i = 2\pi * i/49$, for $i = 0, 1, 2, \dots, 49$, and the corresponding y_i computed based on the corresponding equation are used to train the network, 100 randomly generated input signals in the range of 0–2π are applied to test the trained network, and the results are shown in Table 2. In Examples 1 and

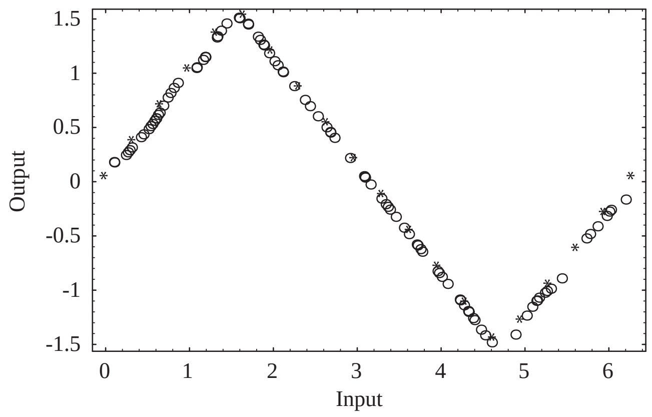

Fig. 3. The trained network output for function defined in Eq. (19) with 20 learning examples. "∗" stands for training data, while "o" stands for test data.

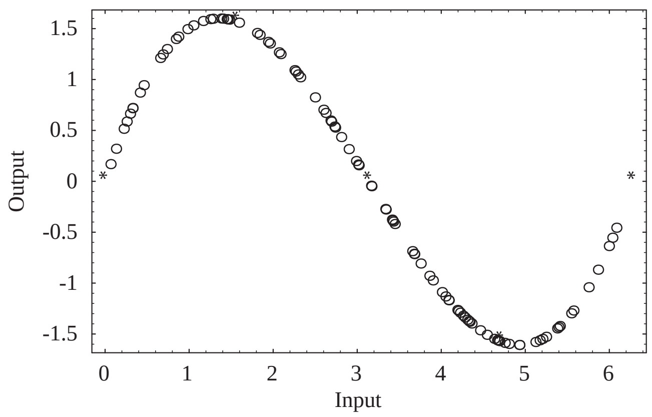

Fig. 4. The trained network output for function defined in Eq. (19) with only 5 learning examples. "∗" stands for training data, while "o" stands for test data.

Table 2
Generalization ability comparison of two examples

	Input range	N	Test set N_1	Generalized E	Generalized $RMSE$	Max deviation
Example 1	0–2π	5	100	0.47846	0.00971	0.16118
		50	100	2.439×10^{-9}	0.9843×10^{-7}	2.5148×10^{-5}
Example 4	0–2π	5	100	5.00536	0.03164	0.56247
		50	100	0.41345	0.00455	0.21708

4, only W^{L} matrix is different, and the other matrices are the same, whereas 50 set examples are used to train the network. But the network's response to the same input matrix is totally different.

One point that needs to be mentioned is that the computation accuracy is machine-dependent due to precision of internal data representation. Furthermore, randomly generated test data may vary when re-computed. Therefore, only the order of magnitude of the learning error is of practical interest.

5. Stacked generalization

As we know, one of the important purpose to train a neural network is for generalization. When training samples set is small and deteriorates by random noise, the network is sometimes overtrained and becomes fitted to the noise, while overfitting the noisy data will degrade the prediction accuracy of the network.

The method of *stacked generalization* [19] provides a way of combining trained networks together, engaging partitioning of the data set to find an overall system with improved generalization performance. The idea is to train the level-0 networks first and then examine their behavior when generalizing. This provides a new training set for training the level-1 network.

The specific procedure for setting up the stacked generalization system is as follows. Let the complete set of available data be denoted by D. We first leave aside a single data point from D as a validation point, and treat the remainder of D as a training set. All level-0 networks are then trained by the training partition and their outputs are measured using the validation data point. This generates a single pattern for a new data set which will be used to train the level-1 network. The inputs of this pattern consist of the outputs of all the level-0 networks, and the target value is the corresponding target value from the original full data set. This process is repeated with a different choice for the data point which is kept aside. After cycling through the full data set of N points we have N patterns in the new data set, which is now used to train the level-1 network. Finally, all of the level-0 networks are re-trained using the full data set D. Predictions on new data can now be made by presenting new input vector to the level-0 networks and taking their outputs as the inputs to the level-1 network, whose output constitutes the predicted output.

Mathematical expression is as the following for CVPS of stacked generalization. Given a training data set $D = \{\mathbf{x}^i, \mathbf{o}^i\}_{i=1}^N$, we randomly partition the data into K almost-equal subset $Ds_1, Ds_2, \ldots, Ds_K$. Define Ds_j and $Ds_{(-j)} = D - Ds_j$ to be the validation and training sets for the jth fold of a K-fold cross-validation. These are called level-0 models. Especially, if $K = N$, the validation set only has one sample, while training set contains $N - 1$ samples. This is called leave-one-out cross-validation.

Let $\mathbf{z}_i$ denote the validation output of the model M_j on $\mathbf{x}_i$. At the end of the entire cross-validation process, the data set assembled from the outputs of the models is

$$D_{\mathrm{cv}} = \{\mathbf{z}_i, \mathbf{o}_i\}_{i=1}^N. \tag{20}$$

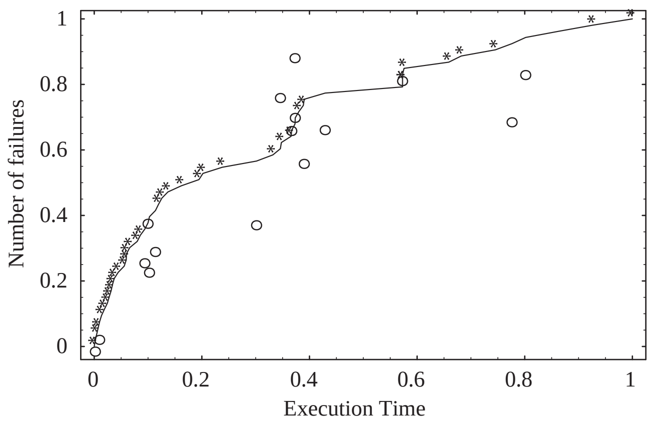

Fig. 5. The neural network model trained with software reliability Sys1 data set(normalized). Solid line is the original data, "∗" stands for training data samples, while "o" stands for test data samples. Because of overfitting the training samples, the network generalization is poor.

This is the level-1 data set used to train level-1 model. To complete the training process, the final level-0 models are derived using all the data in D.

The experiments show that with smooth function or piecewise smooth function, the trained network generalization performance is good with stacked generalization. The examples also illustrate that generalization can be expected when the underlying function is sufficiently smooth. However, for noisy data, if the network is overtrained (overfit to noise), the generalization will be poor. Using stacked generalization cannot improve the network performance when overtrained networks are engaged. The reason is that the overtrained network is biased to particular training samples, therefore, forecasting the values which are not in the training set will be far away from the expected values.

In order to investigate the properties of the stacked generalization technique in noisy data case, we adopt real world data sets in further experiments. The data sets are Sys1 and Sys3 software failure data applied for software reliability growth modelling in [13].

Sys1 data set contains 54 data pairs. In the experiment, we partition the data into two parts: training set and test set. The training set consists of 37 samples which are randomly drawn from the original data set. The remaining 17 samples consist the test set. The data set are normalized to the range of [0,1]. Normalizing is a standard procedure for data preprocessing. In this problem, the network input is normalized successive failure occurrence times, and the network output is the accumulated failure number. During training, each input sample x_t at time t is associated with the corresponding output value o_t at the same time t. This kind of training is called generalization training [10].

Fig. 5 shows the experimental result for software reliability growth modelling trained by using data set Sys1, which is one of the level-0 network output. Fig. 6 shows the stacked generalization output for Sys1 data set. Because of overfitting the training samples, the level-0 output strays away. These samples are not in level-1 training data

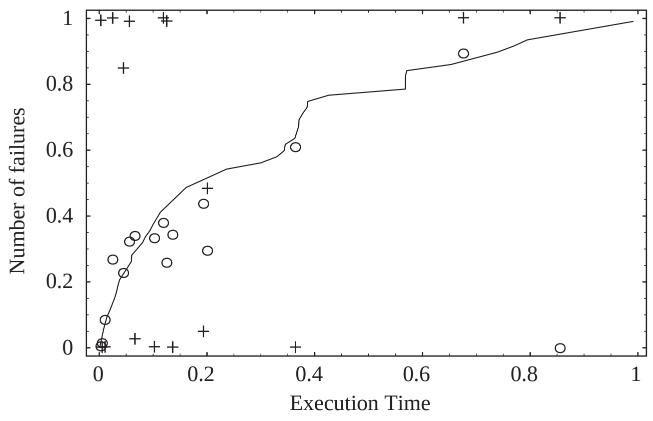

Fig. 6. The stacked generalization output for Sys1 data set (normalized). Solid line is the original data, "o" stands for test data output of level-0 neural network, while "+" stands for test data output of level-1 network output. The results are also poor.

either, and the level-1 network outputs are further away from the desired values. The generalization ability is not improved by stacked generalization because of overfitting to the noise. Here we can see that when overfitting to the noise occurs, stacked generalization is not a suitable technique for improving network generalization performance. Poor generalization ability is not what we expected, so we should seek for the methods that can avoid overfitting in noisy data cases.

As we have mentioned early in this paper, PIL algorithm can eliminate learning errors layer by layer. For the generalization problem, we do not expect to realize the perfect learning. Therefore, we may adopt the strategy like early stopping to employ a three-layer neural network structure.

Fig. 7 shows the experimental result by a three-layer structure trained with data set Sys1, which is one of the level-0 network output. To avoid overfitting, the training error is not small, and the network outputs for training samples are not completely fitting the target values. Compared with perfect leaning error which is 2.3×10^{-9}, the training error is now 0.0034. This introduces the bias to the training samples, and the output tend to be a smooth curve.

Fig. 8 shows the stacked generalization output for Sys1 data set. In this case, with stacked generalization, the total sum-of-square test error is 0.0152. While without stacked generalization, the total sum-of-square test error is 0.0434. Therefore, generalization ability is improved by stacked generalization.

Another data set is Sys3. In this data set, altogether there are 278 data pairs. In the experiment, we partition the data into a training set and a test set. The number of training data is about $\frac{2}{3}$ of the total data number, consisting of randomly drawn 186 samples from the original data set. The remaining 92 samples form the test set.

If we assign the training error as 10^{-7}, after two hidden layers are added, the final training error reaches the order of 10^{-14}. But with this trained network, the test error (20.329) is large. Fig. 9 shows the results.

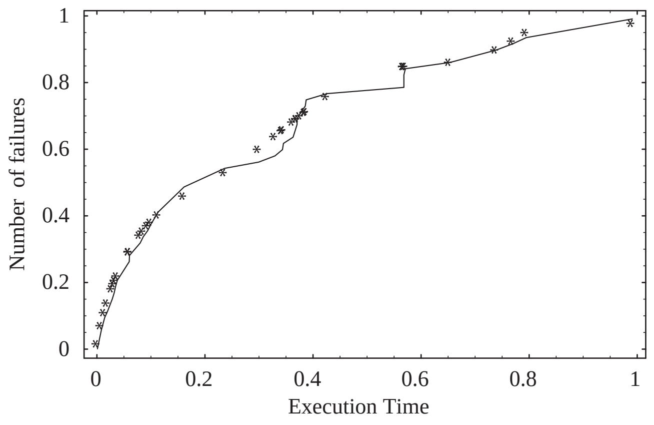

Fig. 7. The three-layer network trained with software reliability Sys1 data set (normalized). Solid line is the original data, and "$*$" stands for training data samples. Training accuracy is not very high and overfitting is avoided.

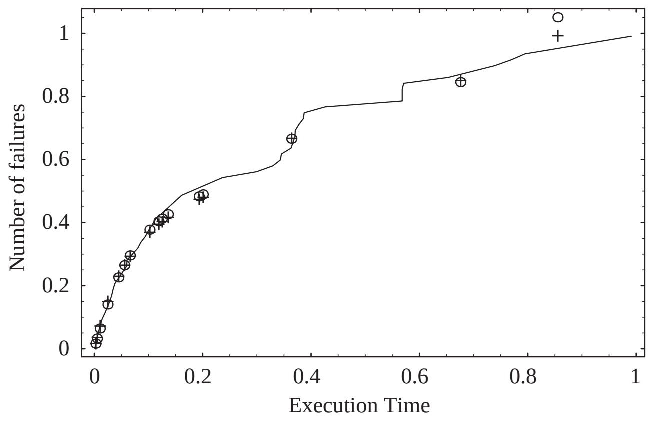

Fig. 8. The stacked generalization output for Sys1 data set (normalized). Solid line is the original data, "o" stands for test data output of level-0 neural network, while "+" stands for test data output of level-1 network output. Generalization is improved at the cost of introducing training bias.

Now we still use leave-one-out CVPS to train level-0 neural networks for stacked generalization. At this time, the three-layer network structure is adopted. For individual network, the training error is about 0.0442, while the test error is 0.0221. Fig. 10 shows the individual network training output, while Fig. 11 is for stacked generalization results.

From these real-world experimental results, we can see that it is at the cost of introducing the bias (training error) to reduce the variance (generalization error) [6]. For most generalization problems the stacked generalization can be expected to reduce the

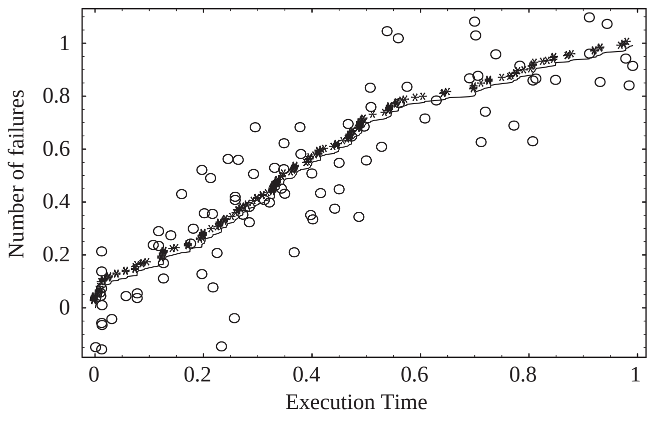

Fig. 9. The network output for Sys3 data set (normalized). Solid line is the original data, "o" stands for test data output. Because of overfitting the training samples, the network generalization is poor.

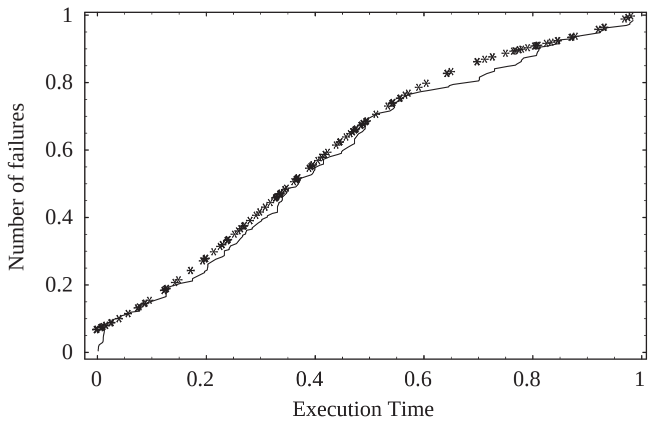

Fig. 10. The three layer network model trained with software reliability Sys3 data set(normalized). Solid line is the original data, and "$*$" stands for training data samples. Training accuracy is not very high and overfitting is avoided.

generalization error rate. For example, in the Sys1 experiment, the test error is 0.0434 without stacked generalization, while the test error reduces to 0.0152 with stacked generalization. However, for some particular data set such as Sys3, stacked generalization dose not show significant improvement (test error is reduced from 0.0221 to 0.0215), but the computation time is dramatically increased. The results are summarized in Table 3.

For a large-scale data set, one of the well-known method is *divide-and-conquer* method. That is, partition the data set into subsets, so as to reduce the individual network size. We can also use other methods such as ensemble networks [15] to im-

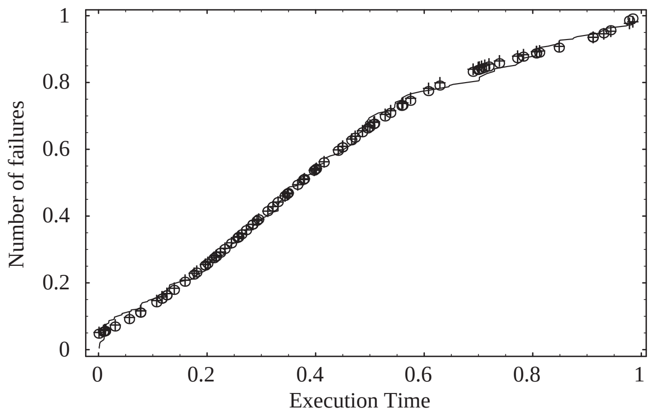

Fig. 11. The stacked generalization output for Sys3 data set (normalized). Solid line is the original data, "o" stands for test data output of level-0 neural network, while "+" stands for test data output of level-1 network output. Generalization is improved at the cost of introducing training bias.

Table 3
Training error and generalization error for software reliability growth model data set

Data set	Sys1	Sys3
Training number	37	186
Test number	17	92
Individual net (4-*layers*)		
Training error	3.49×10^{-4}	1.47×10^{-14}
Test error	58.003	20.329
Individual net (3-*layers*)		
Training error	0.0181	0.0442
Test error	0.0434	0.0221
Stacked (level-1)	0.0152	0.0215
Test error		

prove the network performance and then apply weight parameter average to reduce the network size [7]. For example, we can employ k-fold CVPS to train neural networks. Details on ensemble neural network generalization ability and its application to software reliability growth model is beyond the scope of this paper.

6. Discussion on PIL features

In this section, we discuss the characteristics of the proposed PIL algorithm.

On examining the algorithm, it can be seen that we do not need to consider the question of how the weight matrix should be initialized to avoid local minima. We just feed forward examples to get a weight matrix and its solution. The algorithm will not fall into a local minima because the pseudoinverse solution achieves a global minima. This is different from the BP algorithm.

Furthermore, in PIL, the output layer is a linear layer. We do not need to calculate the inverse of the activation function. It does not need to be a invertible activation function. In this aspect, the PIL is different from either the BP algorithm or other gradient descent algorithms.

As a comparison, the BP algorithm requires user-selected parameters, such as step size or momentum constant. These parameters have an effect on the learning speed. There is no theoretical basis which guides us how to select these parameters to speed up learning. In PIL, on the other hand, such a problem does not exist.

It can also be seen that the training procedure is in fact the processing of raising the rank of the weight matrix. When a matrix of some hidden layer output becomes full rank, the right inverse of the matrix can be obtained, thus completing the training procedure. From this learning procedure, it is obvious that no differentiable activation function is needed. We only require that the activation function can perform nonlinear transform to raise the rank of the weight matrix. Nevertheless, a sigmoidal-like nonlinear function is used in this paper since its transformation has been proven to be capable of raising the rank of a matrix [18].

Another characteristics is that if the input matrix has rank N then a right inverse exists, and we will get a linear network with only two layers. If we give learning error $E = 1.0 \times 10^{-7}$ to Example 1, and when N is greater than 2 and less than 10, it is necessary to add one hidden layer in order to reach the learning accuracy. If N is greater than 10, it is necessary to add two hidden layers. In Example 2, when N is less than 10, we will get a linear network with input and output layers only. The situation of Example 3 is the same as in Example 1 because the input matrices have the same rank. For most problems, with two hidden layers, the network can reach the required high learning accuracy. From the given examples, we see that the network layer number is not only dependent on learning accuracy, but also on the data to be learned. One thing we should address is that after the nonlinear transformation, the degree of rank change is data dependent. It is a very difficult problem to formulate a universal theory to determine how many layers are needed for the perfect learning. But if we intend to reduce the network complexity, we can add a same-dimension gaussian noise matrix to perturb the transformed matrix in step 4 of the PIL algorithm. The inverse function of the perturbed matrix will exist with probability one because the noise is an identical and independent distribution. In such a strategy, we can constrain the hidden layers to at most two to reach the perfect learning. In the perfect learning case, the trained network generalization will be degraded with noisy data set. For some real-world tasks, however, high learning accuracy is not needed.

We have not compared the overall performance of this algorithm with other gradient descent algorithms. Obviously, the number of iterations is not a valid metric considering the fact that the calculation complexity per iteration is not the same for any of the algorithms. However, if we consider the CPU time cost on training network to reach the same high learning accuracy using the same machine, the PIL algorithm is much faster than other gradient descent algorithms in its learning speed. For example, we use the same machine (Sun Ultra 5/270 workstation) and the same software environment

(Mathematica software) to train one neural network with the data set Sys3 to reach the learning accuracy as high as 10^{-14}, it takes less than 7.8 s (including display time) when using the PIL algorithm. As a contrast, it requires more than 10 h of computation time when using the BP algorithm to reach the same result.

7. Summary

The pseudoinverse learning (PIL) algorithm was introduced in this paper. The algorithm is more effective than the standard BP and other gradient descent algorithms for most problems. The algorithm does not contain any user-dependent parameters whose values are crucial for the success of the algorithm. This algorithm is especially suitable for functional mapping and pattern recognition problems. When considering its learning speed and accuracy, the PIL algorithm is most competitive to other gradient descent algorithms in real-time or near real-time applications for practical use. The algorithm is tested on case studies with the stacked generalization applications to software reliability growth modelling data. The fast learning property of the PIL algorithm makes it possible for us to investigate the computation-intensive generalization techniques more efficiently.

Acknowledgements

The authors wish to thank the anonymous reviewers for their useful suggestions and comments on the paper.

This research work was supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region (Project No. CUHK4193/00E) and a grant from the National Natural Science Foundation of China (Project No. 60275002).

References

[1] E. Barnard, Optimization for training neural nets, IEEE Trans. Neural Networks 3 (1992) 232–240.

[2] F. Biegler-König, F. Bärmann, A learning algorithm for multilayered neural networks based on linear least squares problems, Neural Networks 6 (1993) 127–131.

[3] C.M. Bishop, Neural Networks for Pattern Recognition, Oxford University Press, Oxford, 1995.

[4] T.L. Boullion, P.L. Odell, Generalized Inverse Matrices, Wiley, New York, 1971.

[5] J.F. Claerbout, Fundamentals of Geophysical Data Processing with Applications to Petroleum Prospecting, McGraw-Hill, USA, 1976.

[6] S. Geman, E. Bienenstock, R. Doursat, Neural networks and the bias/variance dilemma, Neural Comput. 4 (1992) 1–58.

[7] P. Guo, Averaging ensemble neural networks in parameter space, in: Proceedings of the Fifth International Conference on Neural Information Processing, IOS Press, Japan, 1998, pp. 486–489.

[8] S. Haykin, C. Deng, Classification of radar clutter using neural networks, IEEE Trans. Neural Networks 2 (1991) 589–600.

[9] R.A. Jacobs, Increased rates of convergence through learning rate adaptation, Neural Networks 1 (1988) 295–307.

[10] N. Karunanithi, D. Whitley, Y.K. Malaiya, Prediction of software reliability using connectionist models, IEEE Trans. Software Eng. 18 (1992) 563–574.

[11] S. Kollias, D. Anastassiou, An adaptive least squares algorithm for the efficient training of artificial neural networks, IEEE Trans. Circuit System 36 (1989) 1092–1101.

[12] Y. LeCun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, Handwritten digit recognition with a back-propagation network, in: D.S. Touretsky (Ed.), Advances in Neural Information Processing Systems, Morgan Kaufmann Publishers, San Mateo, CA, 1990, pp. 396–404.

[13] M.R. Lyu, Handbook of Software Reliability Engineering, IEEE Computer Society, McGraw-Hill, New York, 1996.

[14] P. Patrick, Van Der Smagt, Minimization methods for training feedforward neural networks, Neural Networks 7 (1994) 1–11.

[15] M.P. Perrone, L.N. Cooper, When networks disagree: ensemble methods for hybrid neural networks, in: R.J. Mammone (Ed.), Artificial Neural Networks for Speech and Vision, Chapman & Hall, London, 1993, pp. 126–142.

[16] C.R. Rao, S.K. Mitra, Generalized Inverse of Matrices and Its Applications, Wiley, New York, 1971.

[17] D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning internal representations by error propagating, in: D.E. Rumelhart, J.L. McClelland (Eds.), Parallel Distributed Processing, Vol. 1, MIT Press, Cambridge, MA, 1986, pp. 318–362.

[18] S. Tamura, Capabilities of a tree layer feedforward neural network, in: Proceedings of the International Joint Conference on Neural Networks, Seattle, USA, 1991, pp. 2757–2762.

[19] D.H. Wolpert, Stacked generalization, Neural Networks 5 (1992) 241–259.

[20] W.F.A. Zodewyk, B. Etienne, Avoid false local minima by proper initializations of connections, IEEE Trans. Neural Networks 3 (1992) 899–905.

Ping Guo is currently a Professor at the Computer Science Department of the Beijing Normal University. From 1993 to 1994 he was with the Department of Computer Science & Engineering at the Wright State University as a visiting faculty. From May 2000 to August 2000 he was with the National Laboratory of Pattern Recognition at Chinese Academy of Sciences as a guest researcher. He received his M.S. degree in physics from Peking University, his Ph.D. degree in Computer Science from the Chinese University of Hong Kong. His current research interests include neural network, image process, software reliability engineering, optical computing and spectra analysis.

Michael R. Lyu is currently a Professor at the Computer Science and Engineering department of the Chinese University of Hong Kong. He worked at the Jet Propulsion Laboratory as a Technical Staff Member from 1988 to 1990. From 1990 to 1992 he was with the Electrical and Computer Engineering Department at the University of Iowa as an Assistant Professor. From 1992 to 1995, he was a Member of the Technical Staff in the Applied Research Area of the Bell Communications Research (Bellcore). From 1995 to 1997 he was a research Member of the Technical Staff at Bell Labs., which was first part of AT&T and later became part of Lucent Technologies.

Dr. Lyu's research interests include software reliability engineering, distributed systems, fault-tolerant computing, wireless communication networks, Web technologies, digital library, and E-commerce systems. He has published over 120 refereed journal and conference papers in these areas. He has participated in more than 30 industrial projects, and helped to develop many commercial systems and software tools. He has been frequently invited as a keynote or tutorial speaker to conferences and workshops in U.S., Europe, and Asia. He initiated the first International Symposium on Software Reliability Engineering (ISSRE) in 1990. He was the program chair for ISSRE'96, and has served in program committees for many conferences, including ISSRE, SRDS, HASE, ICECCS, ISIT, FTCS, ICDSN, EUROMICRO, APSEC, PRDC, PSAM and ICCCN. He is the General Chair for ISSRE'2001, and the

WWW10 Program Co-Chair. He is the editor for two book volumes: Software Fault Tolerance, published by Wiley in 1995 and the Handbook of Software Reliability Engineering, published by IEEE and McGraw-Hill in 1996. He is an associated editor of IEEE Transactions on Reliability, IEEE Transactions on Knowledge and Data Engineering, and Journal of Information Science and Engineering.

Dr. Lyu received his B.S. in Electrical Engineering from National Taiwan University in 1981, his M.S. in Computer Engineering from University of California, Santa Barbara, in 1985, and his Ph.D. in Computer Science from University of California, Los Angeles, in 1988.