Pseudoinverse Learning Algorithm for Feedforward Neural Networks*

PING GUO and MICHAEL R. LYU
Department of Computer Science and Engineering,
The Chinese University of Hong Kong,
Shatin, NT, Hong Kong.
SAR of P.R. CHINA
pguo@cse.cuhk.edu.hk, lyu@cse.cuhk.edu.hk http://www.cse.cuhk.edu.hk/~lyu

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. The algorithm eliminates learning errors by adding hidden layers and will give a perfect learning. Unlike the 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.

Key-Words: - Feedforward neural networks, Supervised learning, Generalized linear algebra, Pseudoinverse learning algorithm, Fast perfect learning.

1 Introduction

Several adaptive learning algorithms for multilayer feedforward neural networks have been proposed[1][2]. Most of these algorithms are based on variations of the gradient descent algorithm, for example, Back Propagation (BP) algorithm[1]. They usually have a poor convergence rate and sometimes fall into local minima[3]. Convergence to local minima can be caused by 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 weight to avoiding local minima, an adaptive least square algorithm using the second order terms of error for weight updating[4]. There is another drawback for most gradient descent algorithms, namely, "learning factors problem", such as learning rate, 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 still too slow and inefficient. For example, like *stacked generalization*[5], which needs to train a lot of networks to get level 1 training samples, it is very computation-time consuming when using BP algorithm to perform the required task.

In order to reduce training time and investigate the generalization properties of learned neural network, in this paper a *Pseudoinverse Learning algorithm* (PIL) is proposed, which is a feedforward-only algorithm. Learning errors are transferred forward and the network architecture established. The trained weights previously in the network are not changed. Hence, the learning error is minimized on each layer separately and not globally for the network as a whole. By adding layers to eliminate errors, all examples of a training set can be perfect learned.

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. While the number of hidden layer depend on the desired learning accuracy and the examples of a training set to be learned in this paper.

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, the $\mathbf{W}^L$ matrix connects

*This research work was fully supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region (Project No. CUHK4193/00E).

the last hidden layer and the output layer. We assume only the input layer has bias neuron, while the hidden layer(s) and the output layer have no bias neuron. The nonlinear activate function is $\sigma(\cdot)$, for example, we can use so called sigmoidal function,

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

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

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

which output is in a range of (-1,1) as an activate 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 $i-$th 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 correspond 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. $\mathbf{X}^0 = [\mathbf{X}|\theta]$, where matrix $\mathbf{X}$ consist 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 $\mathbf{o}^i$. In the designed network structure, the activate 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,

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

Where $g(\mathbf{x},\Theta)$ is a network mapping function and Θ is the network parameter set. In three layer structure case,

$$g_j(\mathbf{x},\Theta) = \sum_{i=1}^{N} w_{i,j}\sigma_i(\sum_{l=1}^{n} w_{i,l}x_l + \theta_i). \tag{4}$$

where θ_i is a bias value for network input.

For simplifying, we can write the system cost function in matrix form,

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

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 activate 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

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

where we use superscript L to donate the last hidden layer output and the last weight matrix.

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

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

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

Without loss generalization, in the following discussion we drop superscript index L in equation (7).

2.2 Pesudoinverse Solution

Now let us discuss the equation

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

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

If $\mathbf{Y} \in R^{N\times N}$ is invertible and has been learned in $L-1$ layer, the system of equation (9) is, in principle, easy to solve. The unique solution for 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 equation (9). There may be none, one or an infinite number of solutions depending on where $\mathbf{O} \in R(\mathbf{Y})$ and whether $N-$ rank$(\mathbf{Y}) > 0$.

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

From linear algebra theorem, it has:

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

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

Proof: See reference [6].

We intend to use pseudoinverse solution for finding weight matrix, the reason is that the theorem from linear algebra states that pseudoinverse solution is the best approximation.

Theorem 2 Suppose that $\mathbf{X} \in R^{p\times m}$, $\mathbf{A} \in R^{N\times p}, \mathbf{B} \in R^{N\times m}$, The best approximate solution of the equation $\mathbf{AX} = \mathbf{B}$ is $\mathbf{X_0} = \mathbf{A^+B}$ (The superscript $+$ denotes the pseudoinverse matrix)

Proof: From reference [6], it is easy proved.

From the theorem 2, it has,

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

From above analysis, we try to find the output layer weight in this way:

Let $\mathbf{W} = \mathbf{Y}^{+}\mathbf{O}$, the learning problem becomes $||\mathbf{Y}\mathbf{Y}^{+}\mathbf{O} - \mathbf{O}||^2 = 0$, where $\mathbf{Y}^{+}$ is the pseudoinverse of $\mathbf{Y}$. This is equal to find the matrix $\mathbf{Y}$ so that $\mathbf{Y}\mathbf{Y}^{+} - \mathbf{I} = \mathbf{0}$, where $\mathbf{I}$ is the identity matrix. So the task of training the network becomes that of managing to raise the rank of matrix $\mathbf{Y}$ up to full rank. As soon as $\mathbf{Y}$ becomes a full rank matrix, the $\mathbf{Y}\mathbf{Y}^{+}$ will become the identity matrix $\mathbf{I}$. Note that since we multiply $\mathbf{Y}$ on the right by $\mathbf{Y}^{+}$, it needs only requiring the right inverse of $\mathbf{Y}$ to exist, not necessarily for $\mathbf{Y}^{+}$ to be a two-sided inverse of $\mathbf{Y}$. This means that $\mathbf{Y}$ needs not be a square matrix, but its number of column should not be less than its number of row. This condition requires that hidden neuron numbers should be greater than or equal to N. If the condition is satisfied, we can find an exact solution for weight matrix. When we chose hidden neuron number to be equal to N, with such a network structure, we can find the weight matrix which can exactly mapping the training set.

2.3 Psudoinverse Learning Algorithm

Based on the above discussion, we first let weight matrix $\mathbf{W}^0$ be equal to $\mathbf{Y}_p^0$ which is an $n \times N$ matrix. Then we apply nonlinear activate function, that is to compute $\mathbf{Y}^1 = \sigma(\mathbf{Y}^0\mathbf{W}^0)$, then compute the $(\mathbf{Y}^1)^{+}$, the pseudoinverse of $\mathbf{Y}^1$, and so on. Because the algorithm is feedforward only, no error will propagate back to preceding layer of neural network. We cannot use standard error form $E = \frac{1}{2N}\text{Trace}[(\mathbf{G}-\mathbf{O})^T(\mathbf{G}-\mathbf{O})]$ to judge whether the trained network has reached the desired accuracy during 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 $\mathbf{W}^L = (\mathbf{Y}^L)^{+}\mathbf{O}$ and stop the training procedure. Otherwise, let $\mathbf{W}^l = (\mathbf{Y}^l)^{+}$, add another layer, feed forward this layer output to next layer again, until we reach the required learning accuracy.

To use any nonlinear activate function in the hidden nodes is to utilize the nonlinearity of the function and to increase the linear independency among the column vectors or, equivalently, the rank of the matrix. It is proved that sigmoidal functions can raise the dimension of the input space up to the number of the hidden neurons[7]. So through nonlinear activate action, the rank of the weight matrix will be raised layer by layer.

With above discussion, we propose a feedforward-only algorithm which reduce 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 remained error. If the third layer added still cannot satisfy the desired accuracy, then another hidden layer is added again to reduce the learning errors until the required accuracy is achieved. From a mathematical point of view, we can summarize the algorithm into the following steps:

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

Step 2. Compute $(\mathbf{Y}^0)^{+}$ =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, compute $\mathbf{Y}^{l+1} = \sigma(\mathbf{W}^l \cdot \mathbf{Y}^l)$.

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

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

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

Deep neural network structure

3 Add and Delete Sample

Equal to increase and decrease hidden neuron number

The proposed algorithm is a batch way learning algorithm, in which we assume that all the input data are available at the time of training. However, in real-time application, 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 pseudoinvese of a new weight matrix with PIL algorithm. When we assign the hidden neuron number is equal to the number of training samples, add or delete the sample is equivalent with add or delete hidden neuron number. Here we use add or delete neuron algorithm to efficiently compute the pseudoinverse matrix.

According to Griville's theorem[8], the first k columns of the $\mathbf{Y}$ matrix consist of a submatrix, the pseudoinverse of this submatrix can be calculated from the previous $(k-1)$-th pseudoinverse submatrix.

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

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

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

where $\mathbf{CD} = ||\mathbf{I} - \mathbf{Y}_{k-1}\mathbf{Y}_{k-1}^{+}\mathbf{y}_k||$.

It needs at most N times iterative cycle to obtain the pseudoinverse of a matrix if there are N columns in this matrix.

With this theorem, we can add the hidden neurons relative easy to calculate the pseudoinverse matrix.

When a hidden neuron is deleted, the matrix need to be update. It is not efficient at all if we compute the pseudoinverse matrix from beginning. Here we consider using bordering algorithm[9] to compute the inverse of the matrix. Given the inverse to 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 old $k \times k$ matrix with an additional row and column attached to its borders.

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

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

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

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

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

When delete 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 the four partitions:

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

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

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

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

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

4 Numerical Examples

The algorithm is tested with the following function mapping examples.

Example 1. Consider a nonlinear mapping problem of *Sine* function by neural network. For the training set, 50 input-output signals (x_i, y_i) pairs were generated with $x_i = 2\pi * i/49$, for $i = 0, 1, 2, \cdots, 49$, and correspond y_i were computed using $y_i = sin(x_i)$. The given learning error is $E = 10^{-7}$. If learning error $E < 10^{-7}$, we regard that perfect learning has been reached. For this problem, input neuron number is $n + 1 = 2$ including 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 4 layers including input and output layer. The actual learning error is $E = 7.533 \times 10^{-18}$.

Example 2. The nonlinear mapping of 8 input quantities x_i into three output quantities y_i problem, defined by Biegler-König and Bärmann in [10]:

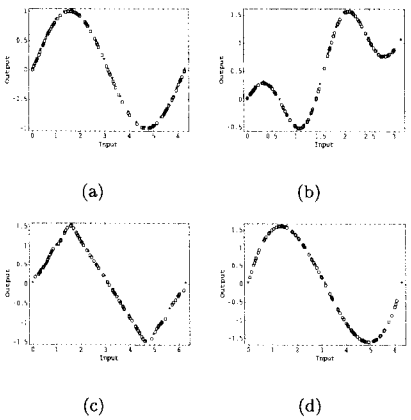
$$\begin{aligned} y_1 &= (x_1 * x_2 + x_3 * x_4 + x_5 * x_6 + x_7 * x_8)/4.0 \\ y_2 &= (x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8)/8.0 \\ y_3 &= (1 - y_1)^{0.5} \end{aligned} \tag{17}$$

All three functions are defined for values between 0 and 1 and produce values in this range. For the training set, 50 sets of input signals x_i were randomly generated in the range of 0 to 1, the corresponding y_i were computed using the above equation. The desired learning error we give is $E = 1.0 \times 10^{-7}$. 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.

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

4.1 Generalization

What is the generalization abilities of trained networks? We also tested the 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 $i = 0, 1, 2, \cdots, 19$, and the corresponding y_i were computed using $y_i = sin(x_i)$. After the network is trained, $N_1 = 100$ input signals x_i randomly generated within the range of 0 to 2π are used to test the network, the corresponding y_i were computed using trained network. Figure 1a and 1b show the results for example 1



(a) (b)

(c) (d)

Fig.1. The trained network output for (a) $y = sin(x)$ function mapping, (b) $y = sin(x)cos(x) + x/3$ function mapping, (c) for function defined in equation (18) with 20 learning examples, and (d) for function defined in equation (18) with only 5 learning examples. "$*$" is training data, "o" is test data.

and 3. It is reasonable good. We have also tested examples 2 and 3 with 20 examples training network and using 100 randomly generated input signals for testing.

For further investigating the proposed network architecture and learning algorithm's response to unlearned data, let us see another example.

Example 4. A $sin(x)$ like nonlinear function is defined by:

$$y = \begin{cases} x, & \text{if } 0 \le x < \pi/2, \\ \pi - x, & \text{if } \pi/2 \le x < 3\pi/2, \\ x - 2\pi, & \text{if } 3\pi/2 \le x \le 2\pi. \end{cases} \quad (18)$$

First, 20 examples with $x_i = 2\pi * i/19$, for $i = 0, 1, 2, \cdots, 19$, and the corresponding y_i were computed using above equation are used to train the network, then 100 random input signals generated in the range of 0 to 2π are used to test the trained network. The result is shown in Figure 1c.

When using 5 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 5 hidden neurons. The learning error is $E = 3.314 \times 10^{-26}$. Afterward, 100 sets of input signals x_i which were randomly generated within the range of 0 to 2π are used to test the network. The result is shown in Figure 1d. From the Figure 1d, it can be seen that the network acts like a *Sine* function. It should be reminded that the architecture and weight matrices is the same as in example 1 and example 4 when using the above 5 set examples. From this result, it can be known that the network forecast unlearned data ability is better for smooth function when the data in the range of training input space. When 50 set examples with $x_i = 2\pi * i/49$, for $i = 0, 1, 2, \cdots, 49$, and the corresponding y_i were computed using corresponding equation are used to train the network, then 100 randomly generated input signals in the range of 0 to 2π are used to test the trained network. In example 1 and 4, only $\mathbf{W}^L$ matrix is different. The other matrices are the same while 50 set examples are used to train the network. But the network's response to the same input matrix is totally different.

The method of *stacked generalization*[5] provides a way of combining trained networks together which uses partitioning of the data set to find an overall system with usually improved generalization performance. The experiments show that with smoothed function or piece wise smoothed function, the trained network generalization performance is good with stacked generalization. However, for noise data set, if the network is overtrained, the generalization ability will be poor. Using stacked generalization can not improve the network performance when over trained networks are used. When overfitting to the noise occur, stacked generalization not a suitable technique for improving network generalization performance. We should seek other generalization techniques such as ensemble networks[11][12] to improve the network performance, but this topic is beyond the scope of this paper.

5 Discussion

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 feedforward examples to get a weight matrix and the solution will not converge to local minima. This is different from the BP algorithm. It can also be seen that training procedure is in fact the processing of raising the rank of weight matrix. When a matrix of some hidden layer output becomes full rank, the right inverse of the matrix can be obtained, and we end the training procedure. From the learning procedure, it is obvious that no differentiable activate function is needed. We only require that the activate function can perform nonlinear transform to raise the rank of the weight matrix. Because the PIL algorithm is based on the nonlinear function transformation to raise the matrix rank, it will fail if there two or more input vectors are identity in the input matrix. But this case can be eliminated through preprocessing input patterns.

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

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. For most problems, with two hidden layers, the network can reach the perfect learning. From the examples, we see that network layer number is not only dependent on learning accuracy, but also on the examples to be learned. The algorithm is suitable for some applications which require high precision output, in which case the network structure is less important than precision output.

One of the algorithm's important feature is that desired output matrix $\mathbf{T}$ is embedded in the weight matrix $\mathbf{W}^L$ which connects last hidden layer and output layer. This give us a very easy and fast way to get the weight matrix for different target output, as long as input matrix is the same. For example, after we have trained the network to learning *Sine* function mapping in the region from 0 to 2π, we only need recalculate the $\mathbf{W}^L$, in order to get *Cosine* function mapping problem in the same region with *Sine* function. For BP algorithm, it is necessary to train whole network again to get all weight matrices for *Cosine* function mapping, though input matrix is the same with *Sine* function.

We have not compared the overall performance of this algorithm with others. 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 high learning accuracy using the same machine, the PIL algorithm is obviously fast than other gradient descent algorithms in its learning speed.

6 Summary

The pseudoinverse learning algorithm was introduced in this paper. The algorithm is more effective than the standard BP and other gradient descent algorithm for most problems. The algorithm does not contain any user-dependent parameters whose values are crucial for the success of the algorithm. The mathematical operations are simple, it is only based on generalized linear algebra and adopt pseudoinverse and matrix inner product operations. On considering its learning speed and accuracy, the PIL algorithm is most competitive to other gradient descent algorithms in real or near real time practical use. With the PIL algorithm, it allows us to investigate the computation-intensive techniques such as stacked generation more efficiently.

References:

[1] D. E. Rumelhurt, G. E. Hinton, and R. J. Williams, "Learning Internal Representation by Error Propagation", in *Parallel Distributed Processing,* Vol. 1, D.E.Rumelhart and J.L.McClelland, Eds. (MIT Press, Cambridge, MA), Chapter 8, 1986.

[2] E. Barnard "Optimization for training neural nets". IEEE Transactions on Neural Networks, Vol.3, No.2, pp232-240, 1992.

[3] F. A. Zodewyk, Wesswls and B. Etienne, "Avoid False local minima by proper Initializations of connections", *IEEE Transaction on Neural networks*, Vol. 3, pp899-905, 1992.

[4] S. Kollias, and D. Anastassiou, "An adaptive least squares algorithm for the efficient training of artificial neural networks", *IEEE Transaction on Circuit and System*, CAS-36, pp1092-1101,1989.

[5] D. H. Wolpert, "Stacked Generalization", *Neural Networks*, Vol.5, pp241-259, 1992.

[6] Thomas L. Boullion and Patrick L. Odell, "Generalized Inverse Matrices", John Wiley and Sons, Inc. (New York), 1971.

[7] S. Tamura, "Capabilities of a Tree Layer Feedforward Neural Network", *Proceedings of International Joint Conference on Neural Networks*, pp2757-2762, (Seattle), 1991.

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

[9] Jon F. Claerbout, "Fundamentals of Geophysical Dada Processing with applications to petroleum prospecting", McGraw-Hill Inc. ,(USA), (TN 271, P4C6), 1976.

[10] F. Biegler-König, and F. Bärmann, , "A learning Algorithm for Multilayered Neural Networks Based on Linear Least Squares Problems", *Neural Networks*, vol. 6, pp127-131, 1993.

[11] M. P. Perrone and 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 and Hall, pp126-142, (London), 1993.

[12] Ping Guo, "Averaging ensemble neural networks in parameter space", In *Proceedings of fifth international conference on neural information processing*, pp486-489, (Japan), 1998.