International Journal of Neural Systems, Vol. 13, No. 1 (2003) 13–24

World Scientific
www.worldscientific.com

A HIERARCHICAL METHOD FOR FINDING OPTIMAL ARCHITECTURE AND WEIGHTS USING EVOLUTIONARY LEAST SQUARE BASED LEARNING

RANADHIR GHOSH* and BRIJESH VERMA†
School of Information Technology, Griffith University,
PMB 50, Gold Coast Mail Center, QLD 9726, Australia
**r.ghosh@gu.edu.au*
†*b.verma@gu.edu.au*

Received 5 August 2002
Revised 2 December 2002
Accepted 2 December 2002

In this paper, we present a novel approach of implementing a combination methodology to find appropriate neural network architecture and weights using an evolutionary least square based algorithm (GALS).[1] This paper focuses on aspects such as the heuristics of updating weights using an evolutionary least square based algorithm, finding the number of hidden neurons for a two layer feed forward neural network, the stopping criterion for the algorithm and finally some comparisons of the results with other existing methods for searching optimal or near optimal solution in the multidimensional complex search space comprising the architecture and the weight variables. We explain how the weight updating algorithm using evolutionary least square based approach can be combined with the growing architecture model to find the optimum number of hidden neurons. We also discuss the issues of finding a probabilistic solution space as a starting point for the least square method and address the problems involving fitness breaking. We apply the proposed approach to XOR problem, 10 bit odd parity problem and many real-world benchmark data sets such as handwriting data set from CEDAR, breast cancer and heart disease data sets from UCI ML repository. The comparative results based on classification accuracy and the time complexity are discussed.

Keywords: Learning algorithms; neural network architecture; evolutionary algorithms; least square methods; optimisation.

1. Introduction

Simultaneous search of weights and architecture has always been a challenging issue in the ANN research area due to the complex nature of the problem.[2–7] The topology of the architecture and weights are in a different level of hierarchy and the problems comprise of two major problems: one is that both the search spaces are multi-dimensional complex spaces and the other is that the weight search space is a sub set of the architecture space. Hence simultaneous search remains a major challenge. Some earlier research showed some prospects of simultaneous search.[8–11] But implementing those algorithms with a sequential machine increases the time complexity to many folds. A working but naïve solution could be to apply the two searching modules for weights and architecture separately. In that case the problem comprises of the combination methodology of the two searching algorithms. The question that remains to be answered is: Can there be a good but simple rule base system that can combine these two searching modules, yet provide a satisfactory result? It can be very interesting to note that a naïve search for the architecture for faster learning combined with the evolutionary least square based search[11,12] for weights updating

with a good combination dynamics yield a good result within a reasonably good time. Our combining methodology is based on certain simple rules considering the stopping criteria of the individual modules that can give a faster convergence and good generalized output.

Earlier work by Verma and Ghosh,[1] suggested an alternative learning methodology, which uses a hybrid technique by using evolutionary learning for the hidden layer weights and least square based solution method for the output layer weights. The proposed algorithm solved the problems of time complexity of the evolutionary algorithm by combining a fast searching methodology using least square technique. Also, by using a matrix based solution method, it could avoid further the inclusion of other iterative local search methods such as error back propagation (EBP), which has its own limitations. However the suggested algorithm could only modify the weights, hence a topology of the ANN architecture is obviously an area of concern. The other problem reported for GALS was its high memory demands. It was shown in some preliminary study that with an input matrix of order 1500×100 (row and column respectively) on a PC with 128 MB RAM and CPU speed of 512 MHz , the main problem was the memory usage of the least square solution routine.

The main aim of the research presented in this paper is to formulate a combination methodology for optimizing weights and architecture and some related issues regarding the new algorithm such as stopping criteria, issues in the breaking of fitness for the population pool. We first discuss the dynamics of the combining rules for the two different tasks of searching of weight and architecture, followed by the description of two different techniques in the consecutive sections.

2. Research Methodology

In Fig. 1, the flowchart for the dynamics of the combination methodology for searching the architecture and weights is given. Henceforth, the two separate modules for weight and architecture will be referred to as find Architecture and GALS module respectively. A simple rule base can describe the working of the stopping criterion for the algorithm.

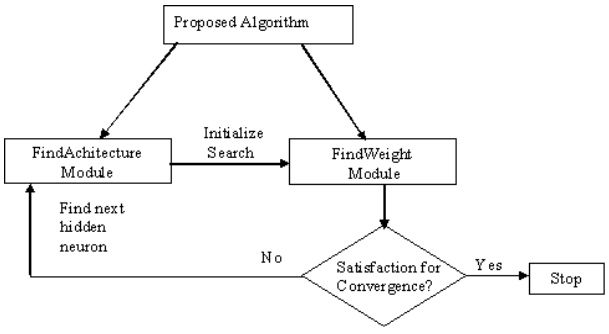

Fig. 1. Dynamics of combination methodology for weights and architecture search.

Rule 1: If the current output is satisfactory, then stop the algorithm, else check rule 2.

Rule 2: If the stopping criterion for the weight search is met and the search is completely exhausted (in terms of number of iterations) then stop else check rule 3.

Rule 3: If the stopping criterion for the weight search is met then go to rule 4 else go to the next generation for the GALS.

Rule 4: If the stopping criterion for GALS is met then go to rule 1, else initialize for the next number of hidden neurons for GALS.

2.1. *Issues for combining the GA and least square method*

How we can combine the evolutionary algorithm with the least square method is again a very important issue as there are many possibilities joining the two independent modules. Depending on the connectivity for combining the GA and LS method, three different connection topologies are devised. The variations of the methods are mainly due to the connection time for calling the least square method from the GA. The three topologies are shown in Fig. 2.

On the basis of the time complexities for the three connections the three variations are known as T1, T2 and T3 on a descending order.[a] The two independent modules of GA and LS are connected together at some point for generating the solution for the weight matrix. The Table 1 shows a brief description of the three connection strategies before giving their individual details.

[a]The order of the connection topology is based on experimental results.

Table 1. Three different connection strategies for calling LS from GA.

Connection strategy	Description
T1	The GA and the LS method are called for every generation and every population to find the corresponding fitness values of the population.
T2	The LS method is called after one generation run for GA method. The best fitness population is halved and the upper half is saved as the weights for output layers. Afterwards, the GA/EA is run for the remaining generation.
T3	The LS method is called after the convergence of the GA is over. After certain number of generations for the GA, the best fitness population is halved and the lower half is used as the weights for the hidden layer and those weights are used for the LS method.

Experimental results show that the time/memory complexity highly depends on this factor of combination. The result shows that T3 is better than T1 and T2 in terms of both the time and memory complexity. So, unless specified otherwise the connection strategy will refer to the T3 connection. However one problem of T3 connection is that, there can be a risk for potential problem of fitness breaking of the chromosome. It can be because while modifying the upper half genes with the weight values from the least square methods, the updating is being done on the fitness of the full length gene, while using only the characteristic from the lower half gene. We show the results and analyze them based on the rank of the population pool to test whether breaking the chromosome into two halves for calling the least square method causes any major setback to the fitness of the gene. The result of fitness breaking analysis has been given in Table 2.

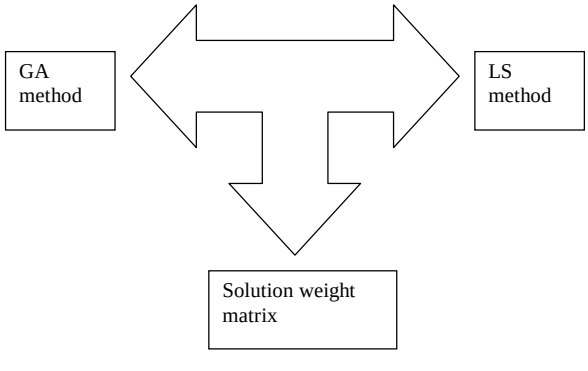

Fig. 2. A general T (1/2/3) connection for GA and LS method.

The flowchart in Fig. 3 shows the input/output relationship between the two modules **findArchitecture** and **GALS**. The decision box shows the combination of all the rules described in the beginning of the sections.

2.2. *Description of GALS*

GALS is described as follows:

Step 1.
Initialize the input range: All the inputs are mapped into a range of the open interval $(0, 1)$. The method of normalization is based on calculating the mean and standard deviation for each element column and uses these to perform additional scaling if required.

Step 2.
Start with some number of hidden neurons: We start the training process using a number of hidden neurons obtained from the **findArchitecture** module. The detail of the module is given in Sec. 2.4.

Step 3.
Initialize all the weights for the hidden layer: We initialize all the hidden layer weights using a uniform distribution of a closed interval range of $[-1, +1]$. We also encode for the genotype, which represents the weights for the hidden layer with those values for all the population strings.

A sample genotype for the lower half gene from the population pool for an n input, h hidden and m output neuron can be written as

$$w_{11}\mu_{11}w_{12}\mu_{12}\cdots w_{1n}\mu_{1n}w_{21}\mu_{21}w_{22}\mu_{22}\cdots \\ \cdots w_{2n}\mu_{2n}\cdots w_{h1}\mu_{h1}w_{h2}\mu_{h2}\cdots w_{hn}\mu_{hn} \quad (1)$$

where, range($\mathbf{w}$) initially is set between the closed interval $[-1\ +1]$.

μ is the variance vector, each value of μ is initialized by a Gaussian distribution of mean 0 and standard deviation 1.

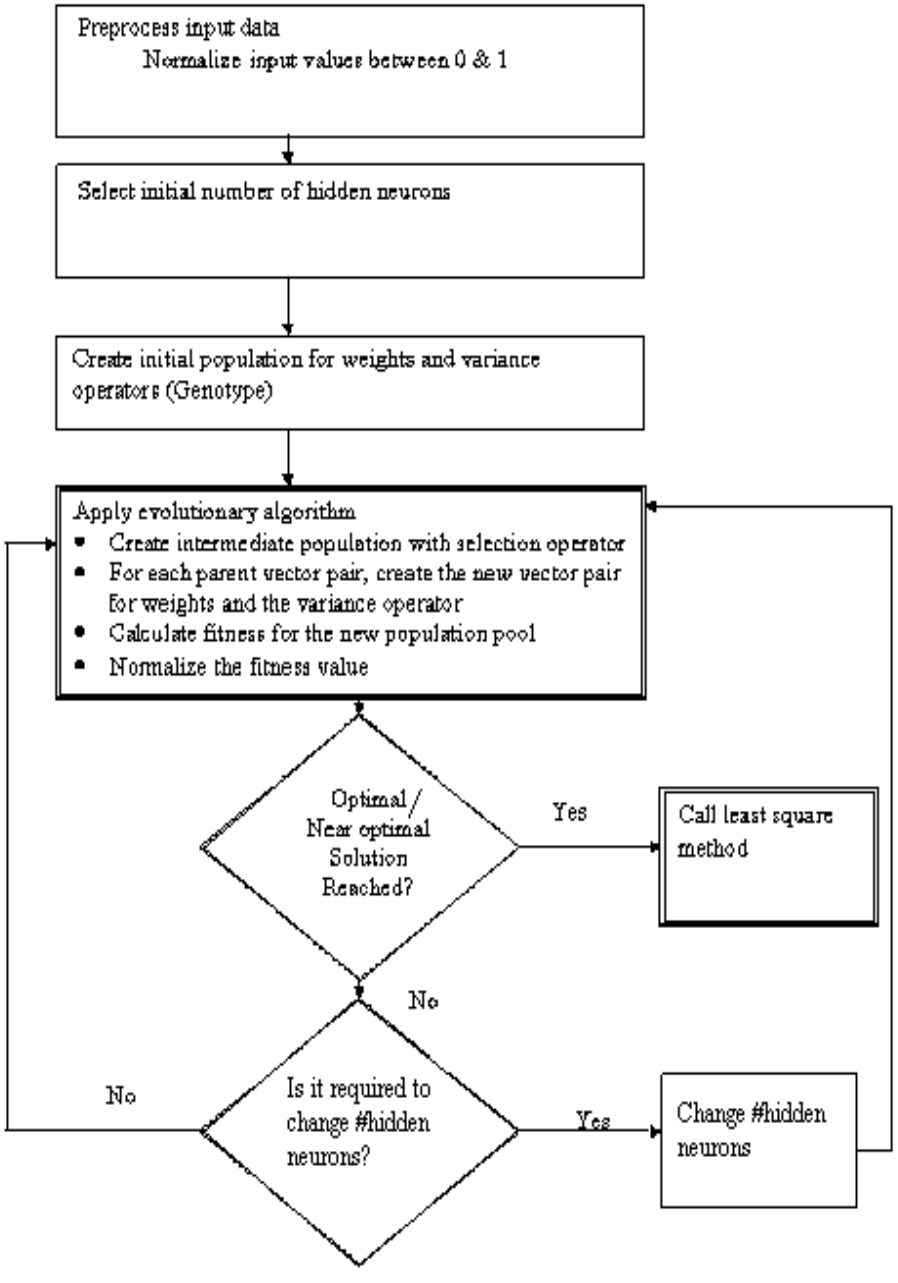

Fig. 3. Flowchart details for GALS and its Input/Output combination with the weight search.

Step 4.

Compute the weights for the output layer using a least square based method: Linear transformations map spaces into spaces. It is important to understand exactly what is being mapped into what in order to determine whether a linear system has solutions, and if so how many. The notion of orthogonality between spaces defines the null space and the range of a matrix and its rank. Two vector spaces $\mathbf{A}$ and $\mathbf{B}$ are said to be *orthogonal* to one another when every vector in $\mathbf{A}$ is orthogonal to every vector in $\mathbf{B}$. If vector space $\mathbf{A}$ is a subspace of $\mathbf{R}^m$ for some m, then the *orthogonal complement* of $\mathbf{A}$ is the set of all vectors that are orthogonal to all the vectors in

A. Any basis $a_1, a_2, \ldots, a_n$ for a subspace **A** of $\mathbf{R}^m$ can be extended into a basis for $\mathbf{R}^m$ by adding $m-n$ vectors $a_{n+1}, \ldots, a_m$. Given n vectors the following construction

$r = 0$
for $j = 1$ to n
$\quad a'_j = a_j - \sum_{t=1}^{r} (q_t^T a_j) q_t$
$\quad$ if $\|a'_j\| \neq 0$
$\quad r = r + 1$
$\quad q_r = \dfrac{a'_j}{\|a'_j\|}$
$\quad$ end
end

yields a set of *orthonormal* vectors $q_1, \ldots, q_r$ that span the same space as $a_a, \ldots, a_n$.

We compute the weights for the output layer using the least square method where the output of the hidden layer is computed as $f(\cdot)$ for the weighted sum of its input, where f is the sigmoid function.

The output of each of the hidden neurons can be found using the equations:

$$h_i = f\left(\sum_{j=1}^{n} w_{ij} I_j\right). \tag{2}$$

Where $i = 1, 2, \ldots, h$, I is the mapped input and

$$f = \frac{1}{1 + \exp^{-x}}. \tag{3}$$

Where x is the output of the hidden neuron before the activation function.

After obtaining the corresponding weight gene from the genotype, as we use a sigmoid activation function for the output also, we need to do the linearization, using the formula

$$netb_i = -\log\left(\frac{1 - net_i}{net_i}\right). \tag{4}$$

Where $i = 1, 2, \ldots, m$, *netb* is the output of the output neuron before the activation, and *net* is the output of the output neuron after the activation.

We then require to solve the over determined system of equations as given below

$$hid^* weight = netb. \tag{5}$$

Where *hid* is the output matrix from the hidden layer neurons and *weight* is the weight matrix output neurons. We use the least square method, which is based on the QR factorization technique to solve the equation for the weight matrix using the *qr* function

$$[Q, R] = qr(hid). \tag{6}$$

The function *qr* returns the orthogonal triangular decomposition of the *hid* matrix. It produces an upper triangular matrix **R** of the same dimension as *hid* and a unitary matrix **Q** so that $\mathbf{hid} = \mathbf{Q}^*\mathbf{R}$. The detail of the algorithm is as follows:

Let $m = n$ and **A** be non singular. We can solve $\mathbf{Ax} = \mathbf{b}$ by calculating the QR factorization of **A** and solving first $\mathbf{Qy} = \mathbf{b}$ (hence $\mathbf{y} = \mathbf{Q}^T\mathbf{b}$) and then $\mathbf{Rx} = \mathbf{y}$ (a triangular system).

Let

$$u \in R^m \backslash \{0\}$$
$$H = I - 2\frac{uu^T}{\|u\|^2} \tag{7}$$

is called a Householder transformations or a Householder reflection. Since $\mathbf{Hu} = -\mathbf{u}$, $\mathbf{Hv} = \mathbf{v}$ if $\mathbf{uTv} = 0$, this transformation reflects any vector $\mathbf{x} \in \mathbf{Rn}$ in the $(n-1)$ dimensional hyper plane spanned by the vectors orthogonal to **u**. Each such matrix **H** is symmetric and orthogonal. The later follows because reflection leaves the Euclidian distance invariant, or by direct calculation

$$\begin{aligned}
&\left(I - 2\frac{uu^T}{\|u\|^2}\right)^T \left(I - 2\frac{uu^T}{\|u\|^2}\right) \\
&= \left(I - 2\frac{uu^T}{\|u\|^2}\right)^2 \\
&= I - 4\frac{uu^T}{\|u\|^2} + 4\frac{u(u^T u)u^T}{\|u\|^4} \\
&= I.
\end{aligned} \tag{8}$$

The Householder transformation offers an alternative to given rotation in the calculation of the QR factorization. The goal of QR factorization is to multiply an $m \times n$ matrix **A** by a sequence of Householder transformation so that each product induces zeros under the diagonal in an entire successive column.

Let $\mathbf{a} \in \mathbf{Rm}$ be the first nonzero column of **A**. We wish to choose $\mathbf{u} \in \mathbf{Rm}$ so that the bottom $m-1$ entries of

$$\left(I - 2\frac{uu^T}{\|u\|^2}\right)^T a = a - 2\frac{u^T a}{\|u\|^2} u. \tag{9}$$

vanish and, in addition, we normalize **u** so that

$$2u^T a = \|u\|^2 , \quad a \neq 0 . \tag{10}$$

Therefore $u_i = a_i$, $i = 2, \ldots, m$ and the normalization implies that

$$\begin{aligned} 2u_1 a_1 + 2\sum_{i=2}^{m} a_i^2 &= u_i^2 + \sum_{i=2}^{m} a_i^2 \\ \Rightarrow u_i^2 - 2u_1 a_1 + a_1^2 - \sum_{i=1}^{m} a_i^2 &= 0 \\ \Rightarrow u_1 &= a_1 \pm \|a\| . \end{aligned} \tag{11}$$

It is usual to let the sign be the same as the sign of a_1, since $\|u\| \ll 1$ might lead to a division by a tiny number, hence to numerical difficulties. For large m we do not execute explicit matrix multiplication. Instead, to calculate

$$\left(I - 2\frac{uu^T}{\|u\|^2}\right) A = A - 2\frac{u^T A}{\|u\|^2} u . \tag{12}$$

We first evaluate $\mathbf{w}^T = \mathbf{u}^T \mathbf{A}$, subsequently forming

$$A - \frac{2}{\|u\|^2} u w^T .$$

We assume that the Householder transformation has been applied $k-1$ times, so that the first $k-1$ columns of the resultant matrix **A** have an upper triangular form. Let a be the kth column of **A**. By the next reflection we want to annihilate the $k+1, \ldots, m$ components of a, however leaving the previous columns as they are. We define the next **u** as

$$\begin{aligned} u_i &= 0 , \quad i < k \\ u_k &= a_k \pm \left(\sum_{i=k}^{m} a_i^2\right)^{1/2} \\ u_i &= a_i , \quad i > k . \end{aligned} \tag{13}$$

Due to its zero components, such a **u** is orthogonal to the previous $k-1$ columns of **A**, hence they remain invariant under reflection (as well as the first $k-1$ rows of **A**). The bottom $m-k$ component **H** a will vanish due to the above computation.

We process columns **A** in sequence, in each stage pre-multiplying a current **A** by the requisite Householder transformation. The end result is an upper triangular matrix **R**. To determine **Q**, we set $\Omega = I$ initially, and for each successive reflection, we replace Ω by

$$\left(I - 2\frac{uu^T}{\|u\|^2}\right) \Omega = \Omega - \frac{2}{\|u\|^2} u(u^T \Omega) . \tag{14}$$

As in the case of given rotation, by the end of the computation, $\mathbf{Q} = \Omega\mathbf{T}$. However if we require just vector $\mathbf{c} = \mathbf{Q}^T\mathbf{b}$, rather than matrix **Q**, then we can set initially $\mathbf{c} = \mathbf{b}$ and in each stage we replace **c** by

$$\left(I - 2\frac{uu^T}{\|u\|^2}\right) c = c - \frac{2u^T c}{\|u\|^2} u . \tag{15}$$

If **A** is dense, it is in general more convenient to use Householder reflections. Given rotations come into their own, however when **A** has many leading zeros in its rows. In an extreme case, if an $n \times n$ matrix **A** consists of zeros underneath the first sub diagonal, they can be rotated away in just $n-1$ rotations at the cost of $0(n^2)$ operations.

The solution matrix can be found from the **R** matrix using one step iterative process as

$$x = \frac{R}{R^T / (hid^T * netb)} . \tag{16}$$

The error **e** can be calculated as

$$r - netb - hid^* x . \tag{17}$$

$$e = \frac{R}{R^T / (hid^T * r)} . \tag{18}$$

The final value of solution for weight matrix can be then found as

$$weight = x + e . \tag{19}$$

Step 5.

Apply Evolutionary algorithm: We create an intermediate population from the current population using a selection operator. We use roulette wheel selection. The method creates a region of a wheel based on the fitness of a population string. All the population strings occupy the space in the wheel based on their rank in fitness. A uniform random number is then generated within a closed interval of $[0, 1]$. The value of the random number is used as a pointer and the particular population string that occupies the number is selected for the offspring generation. Once the intermediate population strings are generated, we randomly choose two parents from the pool of the intermediate population, and apply the genetic operators (mutation only) to generate the offspring with some predefined probability distribution.

We continue this process until the number of offspring population becomes the same as the parent population.

Once the new population has been created, we find the fitness for the whole population based on the weights of the hidden weights (obtained from the population string) and the output layer weights (obtained from the least square method). We also normalize the fitness value to force the population string to maintain a pre-selected range of fitness.

Intermediate population generation:

$$netOutput = f(hid^* weight)\,. \tag{20}$$

Where f is the sigmoid function

$$RMSError = \sqrt{\frac{\sum_{i=1}^{n}(netOutput_i - net_i)^2}{n^*m}}\,. \tag{21}$$

$$popRMSError_i = norm(RMSError_i)\,. \tag{22}$$

norm function normalized the fitness of the individual, so the fitness of each individual population is forced to be within certain range.

Offspring generation:
Each individual population gene (w_i, η_i), $I = 1, 2, \ldots, \mu$ creates a single offspring (w'_i, η'_i) for $j = 1, 2, \ldots, n$

$$\begin{aligned} \eta'_i(j) &= \eta'_i(j)\exp(\tau N(0,1) + \tau' Nj(0,1)) \\ w'_i(j) &= w'_i(j) + \eta'_i(j) Nj(0,1) \end{aligned} \tag{23}$$

Where $w_i(j)$, $w'_i(j)$, $\eta_i(j)$ and $\eta'_i(j)$ denote the jth component of the vectors w_i, w'_i, η_i and η'_i, respectively. $N(0, 1)$ denotes a normally distributed one-dimensional random number with mean and variance of 0 and 1 respectively. $N_j(0, 1)$ denotes that the random number is generated a new for each value of j. The parameter τ and τ' are set to

$$(\sqrt{2\sqrt{n}})^{-1} \quad \text{and} \quad (\sqrt{2n})^{-1}\,.$$

Step 6.
Check the error goal: If any population string from the population pool meets the error goal criterion, we stop the algorithm. Otherwise, go to step 7.

Step 7.
Go back to the ***findArchitecture*** *module.*

2.3. *Stopping criteria for GALS*

The Stopping criterion for GALS is also based on a few simple rules. All the rules are based on the current train and test output and the maximum number of generations for the evolution algorithm. In the following, we describe the stopping criterion for the convergence of the evolutionary algorithm.

If (best_RMS_error[b] < goal_RMS_error) then
Stop

Else if (number_of_generation = total_number_of_generation[c]) then
Stop

Else if (train_classification_error is increased in #m[d] consecutive generation) then
Stop Else continue

2.4. *Finding optimal number of hidden neurons (findArchitecture)*

We used two different types of experiments – linear incrementing for GALS (LIGALS) and binary search type for GALS (BTGALS) to find the number of hidden neurons:-

1. Starting with a small number, and then incrementing by 1 (LIGALS)
2. Using a binary search type (BTGALS)

2.4.1. *Experiment A (LIGALS)*

In experiment A, we start with a small number of hidden neurons and then increment by one. The stopping criterion for this experiment is as follows:

If (train_classification_error = 0) then
Stop

Else If (the test classification error is high in #n[e] consecutive generation) then
Stop

[b]The best_RMS_error is the best of the RMS error from the population pool.
[c]Total number of generations is considered as 30.
[d]m is considered as 3.
[e]We use $n = 3$ for our experiments, which was determined by trial and error method.

2.4.2. *Experiment B (BTGALS)*

In experiment B, we use a binary search type to find the optimal number. A pseudo-code of the algorithm is given below:

Step 1: Find the % test classification error & train_classification_error (error_min) for #min number of hidden neurons
error_min = (train_classification_error (%)+ test_classification_error (%))/2

Step 2: find the % test classification error & train classification error (error_max) for #max number of hidden neurons
error_max = (train_classification_error (%) + test_classification_error (%))/2

Step 3: Find the % test classification error & train classification error (error_mid) for #mid (mid = (min + max)/2) number of hidden neurons
error_mid = (train_classification_error (%) + test_classification_error (%))/2

Step 4: if (error_mid < error_min) and (error_mid > error_max) then

$$\text{min} = \text{mid}$$
$$\text{mid} = (\text{min} + \text{max}/2)$$

else

$$\text{max} = \text{mid}$$
$$\text{mid} = (\text{min} + \text{max}/2)$$

end if

Step 5: Go to Step1, if (mid > min) and (mid < max) Else go to Step 6

Step 6: #number of hidden neurons = mid

3. Data Set

We tested our proposed approach on XOR, 10 bit Odd Parity, and some other real-world benchmark data sets such as handwriting character dataset from CEDAR, Breast cancer and Heart Disease from UCI Machine Learning repository. Table 3 shows the details of the individual data sets considered for training and testing for this new proposed methodology.

Table 2. Abbreviation.

Name	Abbreviation
10 bit odd parity	OP
Handwriting character	HC
Breast cancer (Wisconsin)	BC
Heart disease (Cleveland)	HD
#Hidden neurons	HN
Time	T
RMS	R
Class	C

Table 3. Input data specification details.

Data Set Name	Input Details	Attribute Information
XOR	#Training = 4 #Testing = X	#I/P = 2 #O/P = 1
OP	#Training = 1000 #Testing = 24	#I/P = 10 #O/P = 1
HC[f]	#Training = 300 #Testing = 50	#I/P = 100 #O/P = 29 (26 characters and 3 rejected characters)
BC	#Training = 400 #Testing = 299	#I/P = 10 #O/P = 1
HD	#Training = 250 #Testing = 53	#I/P = 14 #O/P = 1

The following abbreviations (Table 2) are used for the different data set and the terms.

4. Experimental Results

Table 4 shows the results of the fitness breaking for the chromosome based on their ranks. We consider it for handwriting data set of pattern length 100. After each generation of the evolutionary algorithm, we calculate the fitness for the population pool, as calculated by our evolutionary algorithm. Then we calculate the fitness of the gene by breaking it into two halves, and taking the first halves and then combining it with the output layer weights (by calling the least square function). The comparisons of the fitness values are reported[g]. We only consider the

[f]For handwriting character data set features were first extracted using the chain code feature extractor from the image file, and then the feature vectors were used as an input for the ANN.

[g]Results for first 10 best fitness chromosomes are given only for two generations of the GALS algorithm due to the simplicity. The further experiment also shows that the order of the rank for the chromosomes remains almost the same.

Table 4. Analysis of fitness breaking for calling least square method.

Generation	Fitness for Full Length Gene		Fitness for Half Length Gene with Least Square	
	Population Number	Ranking	Population Number	Ranking
1	10	1	10	2
	11	2	11	1
	9	3	9	3
	33	4	33	5
	37	5	37	7
	1	6	1	4
	41	7	41	6
	36	8	36	8
	17	9	17	9
	19	10	19	14
2	17	1	17	1
	3	2	3	2
	8	3	8	4
	1	4	1	3
	46	5	46	6
	47	6	47	10
	11	7	11	8
	19	8	19	7
	32	9	32	13
	17	10	17	9

first 10 ranking of the population pool from the set of 50. The results in Table 4 show that the fitness order does not change much, when the upper half gene was considered. The detail analysis for this result is given in Sec. 5.

Table 5 shows the effect on finding the inverse function for the QR decomposition method, which is used to find the weight matrix. The other simulation results suggested that the difference of order in the hidden matrix and the difference in range give a better result for calculating the inverse function for the **R** matrix, which in turns becomes the weight matrix. We use the handwriting data set as input. The length of the pattern is 100. After initializing the hidden layer weights with specific range as closed interval, the output of the hidden neurons were tested.

Table 5. Analysis of initial weight range for the hidden layers.

Range	Max	Min	Std. Dev
0–9	1	1	0
0.1–0.9	1	0.9979	0.000156
0.01–0.09	0.83648	0.64958	0.0337
0.001–0.009	0.541417	0.515473	0.00455
0.0001–0.0009	0.706357	0.5706	0.0237

Tables 6, 7 and 8 show the results for training and testing RMS and classification errors using EBP[h,i] and the GALS (applying two different stopping criteria). The time complexity of all the algorithms is found by the total time taken by each algorithm to

[h]All the experiments were conducted on SP2 supercomputer at Griffith University, which consists of eight RS/6000 390 machines and 14 RS/6000 590 machines connected by a high speed switch.

[i]The training time for all the algorithms was considered within some specific range for comparison purpose. The training for EBP was forcibly stopped after it crossed the time range limit from the other two algorithms.

Table 6. RMS and classification error results for EBP.

Data Set	Train Error		Test Error		HN	T $(m)^j$
	R	C (%)	R	C (%)		
XOR	0.03	0	0.02	50	3	3
OP	0.368	9	0.042	19	23	80
HC	0.537	18	0.154	34	27	128
BC	0.342	19	0.083	32	19	87
HD	0.317	11	0.068	23	14	77

Table 7. RMS and classification error results for GALS (LIGALS).

Data Set	Train Error		Test Error		HN	T (m)
	R	C (%)	R	C (%)		
XOR	0.003	0	0.002	25	2	2.5
OP	0.068	9	0.009	21	56	67
HC	0.048	11	0.046	17	101	113
BC	0.058	14	0.037	21	46	82
HD	0.048	9	0.005	12	41	68

Table 8. RMS and classification error results for GALS (BTGALS).

Data Set	Train Error		Test Error		HN	T (m)
	R	C (%)	R	C (%)		
XOR	0.003	0	0.002	25	2	2
OP	0.059	10	0.009	22	48	51
HC	0.061	13	0.037	16	86	92
BC	0.047	11	0.036	19	38	77
HD	0.039	8	0.003	13	34	64

find the optimal number of hidden neurons and the learning within some specific time range.

5. Analysis and Discussion of the Results

This section presents the analysis and discussion of results listed in Tables 4–8. The results in Table 4 show that the fitness of the gene (specially for the chromosome with very high fitness) are not affected much when it is broken and combined with the least square method. So, even when the upper half characteristic was replaced by the least square weights, and only the lower half characteristic was considered, it did not break the fitness and affected their rank. The fitness based on the lower half characteristic of the gene had almost the same rank in the population pool, with the fitness based on the full length gene.

The results in Table 6 show that an initial close range of (0.01–0.09) was the best choice for initialization of the hidden layer weights, because this range produced the highest diversities in the hidden layer weights. When combined with the least square method the highest diversified hidden layer output produced the best solution when we compared the result with the least diversified hidden layer weight range.

The results in Tables 6–8, show that the EBP takes comparatively smaller number of hidden neurons than LTGALS and BTGALS, but in terms of training and testing classification error and time complexity the proposed algorithm outperforms the traditional EBP. The graphs in Figs. 4–6 show the comparison results in terms of training and testing classification accuracy as well as the time complexity for all the algorithms for clear explanation.

Figure 4 shows the comparison of classification accuracy for the training data set. The graph shows that, in terms of training classification accuracy for LTGALS and BTGALS outperforms the traditional EBP.

Figure 5 shows the comparison of classification accuracy for the testing data set. The graph shows that, in terms of testing complexity LTGALS and

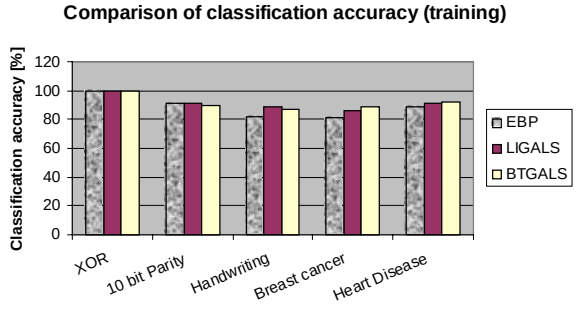

Fig. 4. Classification accuracy for training data set.

$^j m$ denotes minutes.

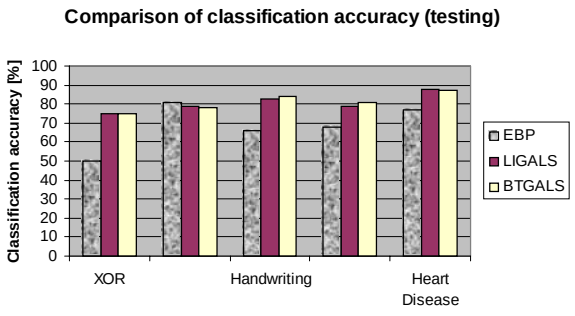

Fig. 5. Classification accuracy for testing data set.

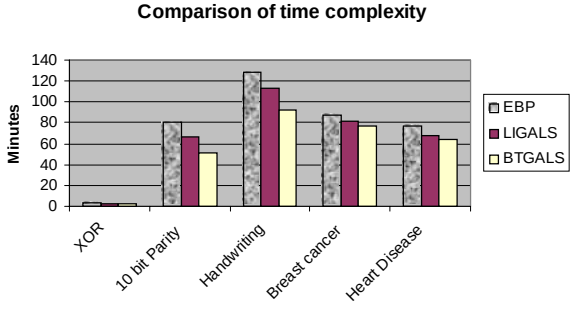

Fig. 6. Time complexity for all the data set.

BTGALS outperforms the traditional EBP in almost all the cases.

Figure 6 shows the comparison of time complexity for all the data set. The results show that in terms of time complexity also, LTGALS and BTGALS are slightly better than the traditional EBP.[k]

6. Conclusion and Further Research

In this paper, we proposed a novel approach for finding appropriate architecture and weights of an MLP. We have discussed with experimental results, how to achieve that with two different stopping criteria approaches. We also improved our earlier GALS algorithm, to reduce the memory complexity and the choice of parameters setting for the algorithm. From the experimental results, we show that the new approach outperforms some other traditional approaches in terms of time, classification accuracy, etc., and it provides us with an optimal/near optimal solution. It should be noted here that had we considered a fully such automated system for EBP, the training time would have even been much higher. When compared to the traditional EBP, in terms of required number of hidden neurons, it is shown that GALS requires more number of hidden neurons. In future, we would like to improve the memory complexity of the algorithm further by introducing a clustering technique for the feature vector of the input data set. So that the training can be performed for not the whole data set but for the data set equal to the number of classes, where each of the vector will be representing a single class.

References

1. B. Verma and R. Ghosh 2002, "A novel evolutionary neural learning algorithm," *IEEE World Congress on Computational Intelligence*, 1884–1889.
2. V. Petridis, S. Kazarlis, A. Papaikonomu and A. Filelis 1992, "A hybrid genetic algorithm for training neural network," *Artificial Neural Networks* **2**, 953–956.
3. A. Likartsis, I. Vlachavas and L. H. Tsoukalas 1997, "New hybrid neural genetic methodology for improving learning," *Proc. of 9th IEEE Intl. Conf. on Tools with Artificial Intelligence*, 32–36.
4. D. Whitley, T. Starkweather and C. Bogart 1990, "Genetic algorithms and neural networks – optimizing connections and connectivity," *Parallel Computing* **14**, 347–361.
5. M. Koeppen, M. Teunis and B. Nickolay 1994, "Neural network that uses evolutionary learning," *Proc. of IEEE Intl. Conf. on Neural Networks* **5**(7), 635–639.
6. R. Hush and B. G. Horne 1993, "Progress in supervised neural networks," *IEEE Signal Processing Magazine* **10**(1), 8–39.
7. M. F. Moller 1993, "A scaled conjugate gradient algorithm for fast supervised learning," *IEEE Trans. on Neural Networks* **6**, 525–523.
8. A. P. Topchy and O. A. Lebedko 1997, "Neural network training by means of cooperative evolutionary search," *Nuclear Instruments & Methods in Physics Research, Section A*: *Accelerators, Spectometers, Detectors and Associated Equipment* **389**(1–2), 240–241.
9. X. Yao and Y. Liu 1998, "Making use of population information in evolutionary artificial neural networks," *IEEE Trans. on Systems, Man and Cybernetics* **28**(3), 417–425.
10. G. Gutierrez, P. Isasi, J. M. Molina, A. Sanchis and I. M. Galvan 2001, "Evolutionary cellular configurations for designing feedforward neural network architectures, connectionist models of neurons, learning processes and artificial intelligence," eds. Jose Mira

[k]It should be noted here as discussed earlier that EBP was forcibly stopped after the time for EBP crossed the upper bound time limit for a certain range from the time taken by the other two algorithms for comparison purposes.

et al., (Springer-Verlag, Germany, LNCS) **2084**, 514–521.
11. A. D. Brown and H. C. Card 1997, "Evolutionary artificial neural networks," *IEEE Canadian Conf. on Voyage of Discovery* **1**, 313–317.
12. C. S. Leung, A. C. Tsoi and L. W. Chan 2001, "Two regularizers for recursive least squared algorithms in feedforward multilayered neural networks," *IEEE Trans. on Neural Networks* **12**(6), 1314–1332.
13. O. Stan and E. Kamen 2000, "A local linearized least squares algorithm for training feedforward neural networks," *IEEE Trans. on Neural Networks* **11**(2), 487–495.