Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
Neurocomputing 56 (2004) 1–38

NEUROCOMPUTING

www.elsevier.com/locate/neucom

Meta learning evolutionary artificial neural networks

Ajith Abraham*

Department of Computer Science, Oklahoma State University, 700 N Greenwood Avenue, Tulsa, OK 74106-0700, USA

Received 7 March 2003; accepted 7 March 2003

Abstract

In this paper, we present meta-learning evolutionary artificial neural network (MLEANN), an automatic computational framework for the adaptive optimization of artificial neural networks (ANNs) wherein the neural network architecture, activation function, connection weights; learning algorithm and its parameters are adapted according to the problem. We explored the performance of MLEANN and conventionally designed ANNs for function approximation problems. To evaluate the comparative performance, we used three different well-known chaotic time series. We also present the state-of-the-art popular neural network learning algorithms and some experimentation results related to convergence speed and generalization performance. We explored the performance of backpropagation algorithm; conjugate gradient algorithm, quasi-Newton algorithm and Levenberg–Marquardt algorithm for the three chaotic time series. Performances of the different learning algorithms were evaluated when the activation functions and architecture were changed. We further present the theoretical background, algorithm, design strategy and further demonstrate how effective and inevitable is the proposed MLEANN framework to design a neural network, which is smaller, faster and with a better generalization performance.
© 2003 Elsevier B.V. All rights reserved.

Keywords: Global optimization; Local search; Evolutionary algorithm and meta-learning

1. Introduction

The strong interest in neural networks in the scientific community is fueled by the many successful and promising applications especially to tasks of optimization [26],

* Tel.: +1-91-85241001; fax: +1-91-85948281.
E-mail address: ajith.abraham@ieee.org (A. Abraham).
URL: http://ajith.softcomputing.net

0925-2312/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S0925-2312(03)00369-2

speech recognition [16], pattern recognition [12], signal processing [57], function approximation [79], control problems [3,5], financial modeling [67], etc. Even though artificial neural networks (ANNs) are capable of performing a wide variety of tasks, yet in practice sometimes they deliver only marginal performance. Inappropriate topology selection and learning algorithm are frequently blamed. There is little reason to expect that one can find a uniformly best algorithm for selecting the weights in a feedforward ANN. This is in accordance with the no free lunch theorem, which explains that for any algorithm, any elevated performance over one class of problems is exactly paid for in performance over another class [54]. In sum, one should be skeptical of claims in the literature on training algorithms that one being proposed is substantially better than most others. Such claims are often defended through some simulations based on applications in which the proposed algorithm performed better than some familiar alternative [41,44,45,78].

At present, neural network design relies heavily on human experts who have sufficient knowledge about the different aspects of the network and the problem domain. As the complexity of the problem domain increases, manual design becomes more difficult and unmanageable. Evolutionary design of ANNs has been widely explored. Evolutionary algorithms (EA) are used to adapt the connection weights, network architecture and learning rules according to the problem environment. A distinct feature of evolutionary neural networks is their adaptability to a dynamic environment. In other words, such neural networks can adapt to an environment as well as changes in the environment. The two forms of adaptation: evolution and learning in evolutionary artificial neural networks (EANNs) make their adaptation to a dynamic environment much more effective and efficient than the conventional learning approach [2]. In Section 2, we present the different neural network learning paradigms followed by some experimentation results to demonstrate the difficulties to design neural networks, which are smaller, faster and with a better generalization performance. In Section 3, we introduce EA and state-of-the-art design of EANNs followed by the proposed meta-learning evolutionary artificial neural network (MLEANN) framework [2]. In the MLEANN framework, in addition to the evolutionary search of connection weights and architectures (connectivity and activation functions), local search techniques are used to fine-tune the weights (meta-learning). Experimentation results are provided in Section 3.2.1 and some discussions and conclusions are provided towards the end.

2. Artificial neural network learning algorithms

The ANN methodology enables us to design useful nonlinear systems accepting large numbers of inputs, with the design based solely on instances of input–output relationships. For a training set T consisting of n argument value pairs and given a d-dimensional argument x and an associated target value t will be approximated by the neural network output. The function approximation could be represented as

$$T = \{(x_i, t_i) : i = 1 : n\}.$$

In most applications the training set T is considered to be noisy and our goal is not to reproduce it exactly but rather to construct a network function that generalizes well to new function values. We will try to address the problem of selecting the weights to learn the training set. The notion of closeness on the training set T is typically formalized through an error function of the form

$$\psi_T = \sum_{i=1}^{n} \|y_i - t_i\|^2, \tag{1}$$

where y_i is the network output. Our target is to find a neural network η such that the output $y_i = \eta(x_i, w)$ is close to the desired output t_i for the input x_i (w = strengths of synaptic connections). The error $\psi_T = \psi_T(w)$ is a function of w because $y = \eta$ depends upon the parameters w defining the selected network η. The objective function $\psi_T(w)$ for a neural network with many parameters defines a highly irregular surface with many local minima, large regions of little slope and symmetries. The common node functions (tanh, sigmoidal, logistic, etc.) are differentiable to arbitrary order through the chain rule of differentiation, which implies that the error is also differentiable to arbitrary order. Hence we are able to make a Taylor's series expansion in w for ψ_T [30]. We shall first discuss the algorithms for minimizing ψ_T by assuming that we can truncate a Taylor's series expansion about a point w^0 that is possibly a local minimum. The gradient (first partial derivative) vector is represented by

$$g(w) = \nabla\psi_T|_w = \left[\frac{\partial\psi_T}{\partial w_i}\right]\Bigg|_w. \tag{2}$$

The gradient vector points in the direction of steepest increase of ψ_T and its negative points in the direction of steepest decrease. The second partial derivative also known as Hessian matrix is represented by H:

$$H(w) = H_{ij}(w) = \nabla^2\psi_T(w) = \frac{\partial^2\psi_T(w)}{\partial w_i \partial w_j}. \tag{3}$$

The Taylor's series for ψ_T, assumed twice continuously differentiable about w^0, can now be given as

$$\psi_T(w) = \psi_T(w^0) + g(w^0)^T(w - w^0)^T + \tfrac{1}{2}(w - w^0)^T H(w^0)(w - w^0) + O(\|w - w^0\|^2), \tag{4}$$

where $O(\delta)$ denotes a term that is of zero-order in small δ such that $\lim_{\delta\to 0}(O(\delta)/\delta) = 0$.

If for example there is continuous derivative at w^0, then the remainder term is of order $\|w - w^0\|^3$ and we can reduce (4) to the following quadratic model:

$$m(w) = \psi_T(w^0) + g(w^0)^T(w - w^0) + \tfrac{1}{2}(w - w^0)^T H(w^0)(w - w^0). \tag{5}$$

Taking the gradient in the quadratic model of (5) yields

$$\nabla m = g(w^0) + H(w - w^0). \tag{6}$$

If we set the gradient $g = 0$ and solving for the minimizing w^* yields

$$w^* = w^0 - H^{-1}g. \tag{7}$$

The model m can now be expressed in terms of minimum value of w^* as

$$m(w^*) = m(w^0) + \tfrac{1}{2}\, g(w^0)^T H^{-1} g(w^0),$$

$$m(w) = m(w^*) + \tfrac{1}{2}\,(w - w^*)^T H(w^*)(w - w^*), \tag{8}$$

a result that follows from (5) by completing the square or recognizing that $g(w^*) = 0$. Hence starting from any initial value of the weight vector, we can in the quadratic case move one step to the minimizing value when it exists. This is known as Newton's approach and can be used in the non-quadratic case where H is the Hessian and is positive definite.

2.1. Multiple minima problem in neural networks

A long recognized bane of analysis of the error surface and the performance of training algorithms is the presence of multiple stationary points, including multiple minima. Analysis of the behavior of training algorithms generally use the Taylor's series expansions discussed earlier, typically with the expansion about a local minimum w^0. However, the multiplicity of minima confuse the analysis because we need to be assured that we are converging to the same local minimum as used in the expansion. How likely are we to encounter a sizable number of local minima? Empirical experience with training algorithm shows that different initialization yield different resulting networks. Hence the issue of many minima is a real one. According to Auer et al. [8], a single node network with n training pairs and R^d inputs could end up having $(n/d)^d$ local minima. Hence not only multiple minima exist, but there may be huge numbers of them.

Different learning algorithms have their staunch proponents, who can always construct instances in which their algorithm perform better than most others. In practice, there are four types of optimization algorithms that are used to minimize $\Psi_T(w)$. The first three methods gradient descent, conjugate gradients and quasi-Newton are general optimization methods whose operation can be understood in the context of minimization of a quadratic error function. Although the error surface is surely not quadratic, for differentiable node functions it will be so in a sufficiently small neighborhood of a local minimum, and such an analysis provides information about the behavior of the training algorithm over the span of a few iterations and also as it approaches its goal. The fourth method of Levenberg and Marquardt is specifically adapted to minimization of an error function that arises from a squared error criterion of the form we are assuming. Backpropagation (BP) calculation of gradient can be adapted easily to provide the information about the Jacobian matrix J needed for this method. A common feature of these training algorithms is the requirement of repeated efficient calculation of gradients.

2.1.1. Backpropagation algorithm

BP provides an effective method for evaluating the gradient vector needed to implement the steepest descent, conjugate gradient, and quasi-Newton algorithms (QNA). BP differs from straightforward gradient calculations using the chain rule for differentiation in the way it organizes efficiently the gradient calculation for networks having more than one hidden layer [68]. BP iteratively selects a sequence of parameter vectors $\{w_k,\ k = 1:T\}$ for a moderate value of running time T, with the goal of having $\{\Psi_T(w_k) = \Psi(k)\}$ converge to a small neighborhood of a good local minimum rather than the usually inaccessible global minimum [30].

$$\psi_T^* = \min_{w \in W} \psi_T(w). \tag{9}$$

The simplest steepest descent algorithm uses the following weight update in the direction of $d_k = -g_k$ with a learning rate or step size α_k.

$$w_{k+1} = w_k - \alpha_k g_k. \tag{10}$$

A good choice α_k^* for the learning rate α_k for a given choice of descent direction d_k is the one that minimizes $\psi_{(k+1)}$.

$$\alpha_k^* = \arg\min_{\alpha} \psi(w_k + \alpha d_k). \tag{11}$$

To carry out the minimization we use

$$\left.\frac{\partial \psi(w_{k+1})}{\partial \alpha}\right|_{\alpha=\alpha_k^*} = \left.\frac{\partial \psi(w_k + \alpha d_k)}{\partial \alpha}\right|_{\alpha=\alpha_k^*} = 0. \tag{12}$$

To evaluate this equation, note that

$$\frac{\partial \psi(w_k + \alpha d_k)}{\partial \alpha} = g_{k+1}^T d_k \tag{13}$$

and conclude that for optimal learning rate we must satisfy the orthogonality condition

$$g_{k+1}^T d_k = 0. \tag{14}$$

When the error function is not specified analytically, then its minimization along d_k can be accomplished through a numerical line search for α_k or through numerical differentiation as noted herein. The line search avoids the problem of setting a fixed step size. Analysis of such algorithms often examine their behavior when the error function is truly a quadratic as given in (5) and (6). In the current notation,

$$g_{k+1} - g_k = \alpha_k H d_k. \tag{15}$$

Hence the optimality condition for the learning rate α_k derived from the orthogonality condition (14) becomes

$$\alpha_k^* = \frac{-d_k^T g_k}{d_k^T H d_k}. \tag{16}$$

When search directions are chosen via $d_k = -M_k g_k$, with M_k symmetric, then the optimal learning rate is

$$\alpha_k^* = \frac{-g_k^T M_k g_k}{g_k^T M_k H M_k g_k}. \tag{17}$$

In the case of steepest descent for a quadratic error function, M_k is the identity and

$$\alpha_k^* = \frac{-g_k^T g_k}{g_k^T H g_k}. \tag{18}$$

One can think of α_k^* as the reciprocal of an expected value of the eigenvalues $\{\lambda_i\}$ of the Hessian with probabilities determined by the squares of the coefficients of the gradient vector g_k expanded in terms of the eigenvectors $\{e_i\}$ of the Hessian:

$$\frac{1}{\alpha_k^*} = \sum_{i=1}^{p} q_i \lambda_i, \qquad q_i = \frac{(g_k^T e_i)^2}{g_k^T g_k}. \tag{19}$$

The algorithm, even in the context of a truly quadratic error surface and with line search, suffers from greed. The successive directions do not generally support each other in that after two steps; say, the gradient is usually no longer orthogonal to the direction taken in the first step. In the quadratic case there exists a choice of learning rates that will drive the error to its absolute minimum in no more than $p+1$ steps where p is the number of parameters [30]. To see this, note that

$$\psi(w) = \psi(w^*) + \tfrac{1}{2}(w-w^*)^T H (w-w^*) = \psi(w^*) + \tfrac{1}{2} g^T H^{-1} g. \tag{20}$$

It is easily verified that if $g_k = g(w_k)$ then

$$g_k = \left[\prod_{1}^{k} (I - \alpha_j H)\right] g_0. \tag{21}$$

Hence for $k \geqslant p$, we can achieve $g_k = 0$ simply by choosing $\alpha_1, \ldots, \alpha_p$ any permutation of $1/\lambda_1, \ldots, 1/\lambda_p$, the reciprocals of the eigenvalues of the Hessian H; the resulting product of matrices is a matrix that annihilates each of the p eigenvectors and therefore any other vector that can be represented as their weighted sum. Of course, in practice, we do not know the eigenvalues and cannot implement this algorithm. However, this observation points out the distinction between optimality when one looks ahead only one step and optimality when one adopts a more distant horizon. Traditionally the step size is held at a constant value $\alpha_k = \alpha$. The simplicity of this approach is belied by the need to carefully select the learning rate. If the fixed step size is too large, then we leave ourselves open to overshooting the line search minimum, we may engage in oscillatory or divergent behavior, and we loose guarantees of monotone reduction of the error function ψ_T. If the step size is too small, then we may need a very large number of iterations T before we achieve a sufficiently small value of the error function. A variation on the constant learning rate is to adopt a deterministic learning rate schedule that varies the learning rate dependant on the iteration number.

An ad hoc departure from steepest descent is to add memory to the recursion through momentum term. Now the change in parameter vector w depends not only on the current gradient but also on the most recent change in parameter vector:

$$\Delta_{k+1} = w_{k+1} - w_k = \beta \Delta_k - \alpha_k g_k \quad \text{for } k \geqslant 0. \tag{22}$$

and what we gain is a high frequency smoothing effect through the momentum term. The change in parameter vector depends not only on the current gradient g_{k-1} but also, in an exponentially decaying fashion (provided that $0 \leqslant \beta < 1$), on all previous gradients. If the succession of recent gradients has tended to alternate directions, then the sum will be relatively small and we will make only small changes in the parameter vector. This could occur if we are in the vicinity of a local minimum, successive changes would just serve to bounce us back and forth past the minimum. If, however, recent gradients tends to align, then we will make an even larger change in the parameter vector and thereby move more rapidly across a large region of descent and possibly across over a small region of ascent that screened off a deeper local minimum. Of course, if the learning rate α is well chosen, then successive gradients will tend to be orthogonal and a weighted sum will not cancel itself out.

2.1.2. *Conjugate gradient algorithm*

The motivation behind the conjugate gradient algorithm is that we wish to iteratively select search directions (d_k) that are non-interfering in the sense that successive minimizations along these directions do not undo the progress made by previous minimizations. The search direction is selected in such a way that at each iteratively selected parameter value w_k, the current gradient g_k is orthogonal to all previous search directions $d_1,\dots,d_{k-1}$. Hence, at any given step in the iteration, the error surface has a direction of steepest descent that is orthogonal to the linear subspace of parameters spanned by the prior search directions. Steepest descent merely assured us that the current gradient is orthogonal to the last search direction. If the error function $\{\Psi_T(w_k)\}$ is quadratic with positive definite Hessian H, choosing the search directions (d_i) to be H-conjugate and the α_i to satisfy (16) is equivalent to the orthogonality between the current gradient and the past search directions given by

$$(\forall i < k < p) d_i^T g_k = 0, \tag{23}$$

it is easily verified that conjugate directions (d_i) also form a linearly independent set of directions in weight space [30]. If weight space has dimension p then of course there can be only p linearly independent directions of vectors. Hence, it is possible to represent any point as a linear combination of no more than p of the conjugate directions, and in particular if w^* is the sought location of the minimum of the error function, then there exist coefficients such that

$$w^* - w_0 = \sum_{i=0}^{p-1} \alpha_i^* d_i. \tag{24}$$

Thus if the error surface is quadratic with a positive definite Hessian, then selecting H-conjugate search directions and learning rates according to (16) guarantees a minimum in no more than p iterations. To be able to apply the method of conjugate gradients we must be able to determine such a set of directions and then solve for the correct coefficients. Conventional conjugate gradient algorithms use a line search to find the minimizing step and are initialized as follows:

$$d_0 = g_0. \tag{25}$$

Introducing a scaling β_k to be determined, and iterate with the simple recursion:

$$d_{k+1} = -g_{k+1+\beta_k} d_k. \tag{26}$$

According to the conjugacy condition in (23) and the recursion of (26) yield

$$d_k^T H d_{k+1} = 0 = d_k^T H(-g_{k+1+\beta_k} d_k). \tag{27}$$

Solving yields the necessary condition that

$$\beta_k = \frac{d_k^T H g_{k+1}}{d_k^T H d_k}. \tag{28}$$

Induction can be established that this recursive definition of conjugate gradient search directions does indeed yield a fully conjugate set when the error function is quadratic, although the derivation of (28) only established that d_k and d_{k+1} are conjugate. A version of the conjugate gradient algorithm that does not require line searches was developed by Moller [61] and uses the finite difference method for estimating Hd_k. To monitor the sign of the product $d_k^T H d_k$, define δ by

$$\delta = d_k^T H d_k. \tag{29}$$

Moller introduces two new variables, λ and $\bar{\lambda}$, to define an altered value of δ, $\bar{\delta}$. These variables are charged with ensuring that $\bar{\delta} > 0$. Although this does not affect the error surface, and the Hessian with the quadratic approximation will still that suggest that there is a maximum along the search direction, the method produces a step size that shows good results in practice. $\bar{\delta}$ is defined as follows:

$$\bar{\delta} = \delta + (\bar{\lambda} - \lambda) d_k^T d_k. \tag{30}$$

The requirement for $\bar{\delta} > 0$ gives a condition for $\bar{\lambda}$:

$$\bar{\lambda} \succ \lambda - \frac{\delta}{d_k^T d_k}. \tag{31}$$

Moller then sets $\bar{\lambda} = 2(\lambda - (\delta / d_k^T d_k))$ to satisfy (2.31) and so ensures $\bar{\delta} > 0$. Substituting this in (2.30)

$$\bar{\delta} = -\delta + \lambda d_k^T d_k. \tag{32}$$

In order to get a good quadratic approximation of the error surface, a mechanism to raise or lower λ is needed when the Hessian is positive definite. Detailed step-by-step description can be found in [61].

2.1.3. Quasi-Newton algorithm

If the error surface is purely quadratic, as per (7) we can solve the minimizing weight vector in a single step through Newton's method. This solution requires knowledge of the Hessian and assumes it to be constant and positive definite. We need a solution method that can take into account the variation of $H(w)$ with w, knowing the fact that the error function is at best only approximately quadratic and removal from a local minimum the approximating quadratic surface is likely to have a Hessian that is not positive definite and the evaluation of true Hessian is computationally too expensive.

The quasi-Newton method addresses themselves to these tasks by first generalizing the iterative algorithm to the form

$$w_{k+1} = w_k - \alpha_k M_k g_k. \tag{33}$$

The choice of step size α_k to use with a search direction $d_k = M_k g_k$ is determined by an approximate line search, and use of line search is essential to the success of this method. The quasi-Newton method iteratively tracks the inverse of the Hessian without ever computing it directly. Let $q_k = g_{k+1} - g_k$, and consider the expansion for the gradient (quadratic case):

$$q_k = H_k(w_{k+1} - w_k) = H_k p_k. \tag{34}$$

If we can evaluate the difference of gradients for p linearly independent increments $p_0, \dots, p_{p-1}$ in the weight vectors, then we can solve for the Hessian (assumed constant). To do so, form the matrices P with ith column the vector p_{i-1} and Q with ith column the vector q_{i-1}. Then we have the matrix equation

$$Q = HP, \tag{35}$$

which can be solved for the Hessian, when the columns of P are linearly independent, through

$$H = QP^{-1}. \tag{36}$$

Thus, from the increments in the gradient induced by the increments in the weight vectors as training proceeds, we have some hope of being able to track the Hessian. An approximation to the inverse M of the Hessian is achieved by interchanging q_k and p_k in an approximation to the Hessian itself:

$$M = PQ^{-1}. \tag{37}$$

Hence the information is available in the sequence of gradients that determine the q_k, and the sequence of search directions and learning rates that determine the p_k, to infer to the inverse of the Hessian, particularly if it is only slowly varying.

The Broyden–Fletcher–Goldfarb–Shanno (BFGS) quasi-Newton algorithm [25] implements the update for the approximate inverse M of the Hessian by

$$M_{k+1} = M_k + \left(1 + \frac{q_k^T M_k q_k}{q_k^T p_k}\right) \frac{p_k p_k^T}{p_k p_k^T} - \frac{p_k q_k^T M_k + M_k q_k p_k^T}{q_k^T p_k}. \tag{38}$$

This recursion is initialized by starting with a positive definite matrix such as the identity, $M_0 = I$. The Determination of the learning rates is critical, as was the case for the method of conjugate directions. Quasi-Newton methods enjoy asymptotically more rapid convergence than that of steepest descent or conjugate gradient methods.

2.1.4. Levenberg–Marquardt algorithm

The Levenberg–Marquardt (LM) algorithm [25] exploits the fact that the error function is a sum of squares as given in (1). Introduce the following notation for the error vector and its Jacobian with respect to the network parameters w:

$$J = J_{ij} = \frac{\partial e_j}{\partial w_i}, \quad i = 1, \dots, p, \;\; j = 1, \dots, n. \tag{39}$$

The Jacobian matrix is a large $p \times n$ matrix, all of whose elements are calculated directly by BP technique as presented in Section 2.1.1. The p dimensional gradient g for the quadratic error function can be expressed as

$$g(w) = \sum_{i=1}^{n} e_i \nabla e_i(w) = Je$$

and the Hessian matrix by

$$H = H_{ij} = \frac{\partial^2 \psi_T}{\partial w_i \partial w_j} = \frac{1}{2} \sum_{k=1}^{n} \frac{\partial^2 e_k^2}{\partial w_i \partial w_j} = \sum_{k=1}^{n} \left(e_k \frac{\partial^2 e_k}{\partial w_i \partial w_j} + \frac{\partial e_k \partial e_k}{\partial w_i \partial w_j} \right)$$

$$= \sum_{k=1}^{n} \left(e_k \frac{\partial^2 e_k}{\partial w_i \partial w_j} + J_{ik} J_{jk} \right). \tag{40}$$

Hence defining $D = \sum_{i=1}^{n} e_i \nabla^2 e_i$ yields the expression

$$H(w) = JJ^T + D. \tag{41}$$

The key to the LM algorithm is to approximate this expression for the Hessian by replacing the matrix D involving second derivatives by the much simpler positively scaled unit matrix $\in I$. The LM is a descent algorithm using this approximation in the form

$$M_k = [JJ^T + \in I]^{-1}, \quad w_{k+1} = w_k - \alpha_k M_k g(w_k). \tag{42}$$

Successful use of LM requires approximate line search to determine the rate α_k. The matrix JJ^T is automatically symmetric and non-negative definite. The typically large size of J may necessitate careful memory management in evaluating the product JJ^T. Hence any positive $\in$ will ensure that M_k is positive definite, as required by the descent condition. The performance of the algorithm thus depends on the choice of $\in$.

When the scalar $\in$ is zero, this is just Newton's method, using the approximate Hessian matrix. When $\in$ is large, this becomes gradient descent with a small step size. As Newton's method is more accurate, $\in$ is decreased after each successful step (reduction in performance function) and is increased only when a tentative step would increase the performance function. By doing this, the performance function will always be reduced at each iteration of the algorithm [12].

2.2. *Designing artificial neural networks*

The error surface of very small networks has been characterized previously. However, practical networks often contain hundreds of weights and in general, theoretical and empirical results on small networks do not scale up to large networks. To investigate the empirical performance with the different learning algorithms on different architectures and node transfer functions, we have choosen three famous chaotic time series benchmarks so that (a) we know the best solution, (b) can carefully control various parameters and (c) know the effect of the different learning algorithms namely BP, scaled conjugate gradient (SCG), QNA and LM algorithm.

We also report some experimentation results related to convergence speed and generalization performance of the four different neural network-learning algorithms discussed in Section 2.1. Performances of the different learning algorithms were evaluated when the activation functions and architectures were changed.

We used a feedforward neural network with 1 hidden layer and the numbers of hidden neurons were varied $(14, 16, 18, 20, 24)$ and the speed of convergence and generalization error for each of the four learning algorithms was observed. The effect of node activation functions, log-sigmoidal activation function (LSAF) and tanh-sigmoidal activation function (TSAF), keeping 24 hidden neurons for the four learning algorithms was also studied. Computational complexities of the different learning algorithms were also noted during each event. The experiments were replicated 3 times each with a different starting condition (random weights) and the worst errors were reported. No stopping criterion, and no method of controlling generalization is used other than the maximum number of updates (epochs). All networks were trained for an identical number of stochastic updates (2500 epochs).We used the following three chaotic time series:

(a) *Waste water flow prediction*. The problem is to predict the wastewater flow into a sewage plant [46]. The water flow was measured every hour. It is important to be able to predict the volume of flow $f(t+1)$ as the collecting tank has a limited capacity and a sudden increase in flow will cause to overflow excess water. The water flow prediction is to assist an adaptive online controller. The data set is represented as $[f(t), f(t-1), a(t), b(t), f(t+1)]$ where $f(t)$, $f(t-1)$ and $f(t+1)$ are the water flows at time $t, t-1$, and $t+1$ (h), respectively. $a(t)$ and $b(t)$ are the moving averages for 12 and 24 h. The time series consists of 475 data points. The first 240 data sets were used for training and remaining data for testing.

(b) *Mackey-glass chaotic time series*. The Mackey-glass differential equation [53] is a chaotic time series for some values of the parameters $x(0)$ and τ:

$$\frac{\mathrm{d}x(t)}{\mathrm{d}t} = \frac{0.2x(t-\tau)}{1+x^{10}(t-\tau)} - 0.1x(t). \tag{43}$$

We used the value $x(t-18)$, $x(t-12)$, $x(t-6)$, $x(t)$ to predict $x(t+6)$. Fourth order Runge-Kutta method was used to generate 1000 data series. The time step used in the method is 0.1 and initial condition were $x(0)=1.2$, $\tau=17$, $x(t)=0$ for $t<0$. First 500 data sets were used for training and remaining data for testing.

(c) *Gas furnace time series data*. This time series was used to predict the CO_2 (carbon dioxide) concentration $y(t+1)$ [17]. In a gas furnace system, air and methane are combined to form a mixture of gases containing CO_2. Air fed into the gas furnace is kept constant, while the methane feed rate $u(t)$ can be varied in any desired manner. After that, the resulting CO_2 concentration $y(t)$ is measured in the exhaust gases at the outlet of the furnace. Data is represented as $[u(t), y(t), y(t+1)]$. The time series consists of 292 pairs of observation and 50% of data was used for training and remaining for testing.

Table 1
Training and test performance for Mackey-glass series for different architectures

Mackey-glass time series			
Learning algorithm	Hidden Neurons	Root mean squared error	
		Training data	Test data
BP	14	0.0890	0.0880
	16	0.0824	0.0860
	18	0.0764	0.0750
	20	0.0452	0.0442
	24	0.0439	0.0437
SCG	14	0.0040	0.0051
	16	0.0053	0.0052
	18	0.0066	0.0067
	20	0.0058	0.0058
	24	0.0045	0.0045
QNA	14	0.0041	0.0040
	16	0.0031	0.0030
	18	0.0035	0.0036
	20	0.0038	0.0038
	24	0.0034	0.0036
LM	14	0.0016	0.0016
	16	0.0015	0.0015
	18	0.0015	0.0015
	20	0.0010	0.0011
	24	0.0009	0.0009

2.2.1. Simulation results using ANNs

Results for four different learning algorithms for different architectures, node transfer functions for the three different time series are presented in the following sections.

2.2.1.1. Network architecture This section investigates the training and generalization behavior of the networks when the architecture of the neural network was changed. The same architecture was used for the three different time series for the four learning algorithms using same node transfer function. Tables 1–3 summarize the empirical results of training and generalization. Figs. 1–6 graphically depict the training and generalization performance for the different learning methods.

2.2.1.2. Node transfer functions This section investigates the effect of different node transfer functions on training and generalization performance for the four learning algorithms. To compare empirically we maintained the same architecture and only changing the node transfer functions and learning algorithms. All the networks were randomly initialized and trained for 2500 epochs. Tables 4–6 summarizes the empirical results of

Table 2
Training and test performance for gas furnace time series for different architectures

Gas furnace time series			
Learning algorithm	Hidden Neurons	Root mean squared error	
		Training data	Test data
BP	14	0.0670	0.1291
	16	0.0835	0.1056
	18	0.0716	0.0766
	20	0.0800	0.0950
	24	0.0663	0.0970
SCG	14	0.0160	0.0331
	16	0.0157	0.0330
	18	0.0165	0.0330
	20	0.0158	0.0361
	24	0.0153	0.0367
QNA	14	0.0137	0.0529
	16	0.0133	0.0465
	18	0.0133	0.0376
	20	0.0136	0.0410
	24	0.0128	0.0516
LM	14	0.0118	0.0450
	16	0.0140	0.0971
	18	0.0116	0.1080
	20	0.0100	0.1880
	24	0.0100	0.1856

training and generalization for the two node transfer functions, TSAF and LSAF, when the architecture was fixed with 24 hidden neurons. Figs. 7–12 graphically depict the convergence characteristics of the four training algorithms for different node transfer functions during 2500 epochs training.

2.2.1.3. Computational complexity of learning algorithms This section investigates the computational complexity of the different learning algorithms when the architecture of the hidden layer is varied using TSAF. The networks were randomly initialized and trained for 2500 epochs using the different learning algorithms. Table 7 summarizes the empirical values of the computational load for the different learning methods for the three different time series.

2.3. Discussion of results obtained

In this section we would like to evaluate and summarize the results of the various experimentations mentioned in Section 2.2.1. For Mackey-glass series (Table 1), all

Table 3
Training and test performance for wastewater flow series for different architectures

Wastewater time series			
Learning algorithm	Hidden Neurons	Root mean squared error	
		Training data	Test data
BP	14	0.1269	0.1340
	16	0.1184	0.1360
	18	0.1182	0.1350
	20	0.1221	0.1370
	24	0.1169	0.1412
SCG	14	0.0459	0.0900
	16	0.0428	0.1130
	18	0.0425	0.1130
	20	0.0423	0.1626
	24	0.0400	0.0920
QNA	14	0.0423	0.1271
	16	0.0367	0.1369
	18	0.0363	0.1360
	20	0.0339	0.1450
	24	0.0316	0.2620
LM	14	0.0364	0.0950
	16	0.0303	0.1631
	18	0.0314	0.1800
	20	0.0259	0.1314
	24	0.0244	0.1560

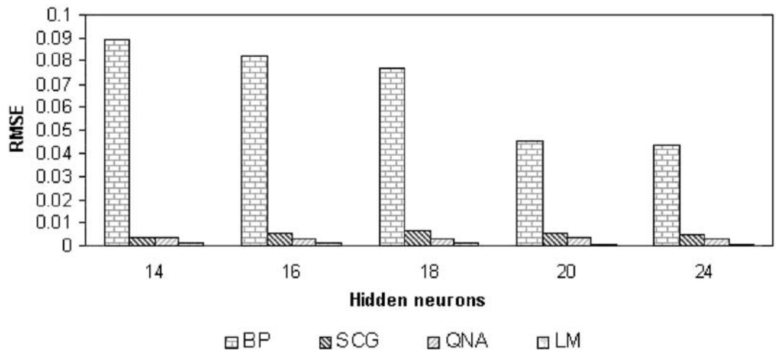


Fig. 1. Architecture variation: Mackey-glass time series training performance for different training algorithms.

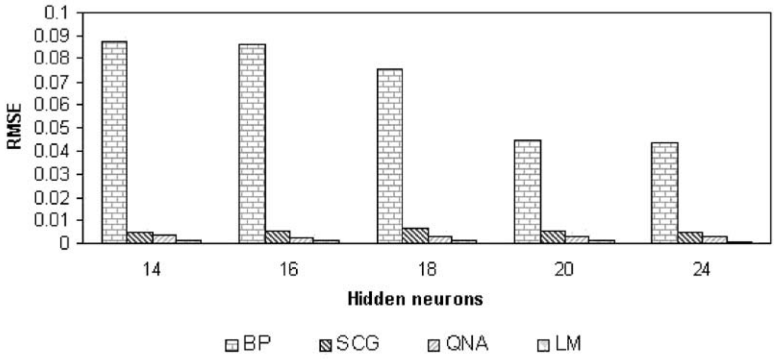


Fig. 2. Architecture variation: Mackey-glass time series generalization performance for different learning algorithms.

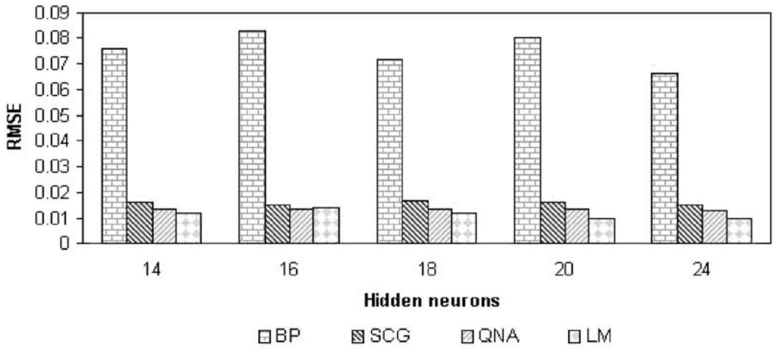


Fig. 3. Architecture variation: gas furnace time series training performance for different training algorithms.

the 4 learning algorithms tend to generalize well as the hidden neurons were increased. However, the generalization was better when the hidden neurons were using TSAF. LM showed the fastest convergence regardless of architecture and node activation function. However, the computational complexity of LM algorithm is very amazing as depicted in Table 7. For Mackey-glass series (with 14 hidden neurons), when BP was using 0.625 billion flops, LM technique required 29.4 billion flops. When the hidden neurons were increased to 24, BP used 1.064 billion flops and LM's share jumped to 203.10 billion flops. LM gave the lowest generalization RMSE of 0.0009 with 24 hidden neurons.

As shown in Table 2, for gas furnace series the generalization performance were entirely different for the different learning algorithms. BP gave the best generalization RMSE of 0.0766 with 18 hidden neurons. RMSE for SCG, QNA and LM were 0.0330 (16 neurons), 0.0376 (18 neurons) and 0.045 (14 neurons), respectively. As depicted in

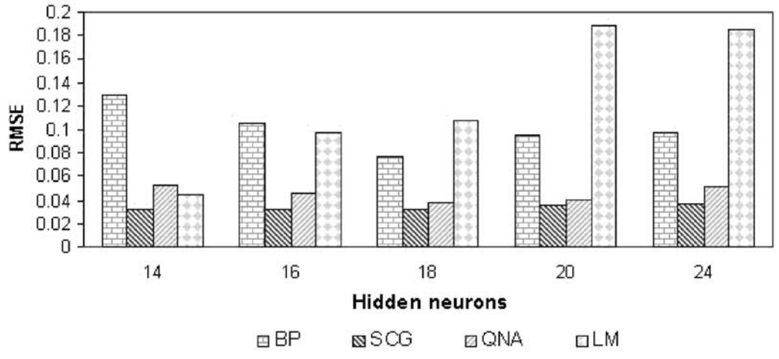


Fig. 4. Architecture variation: gas furnace time series generalization performance for different learning algorithms.

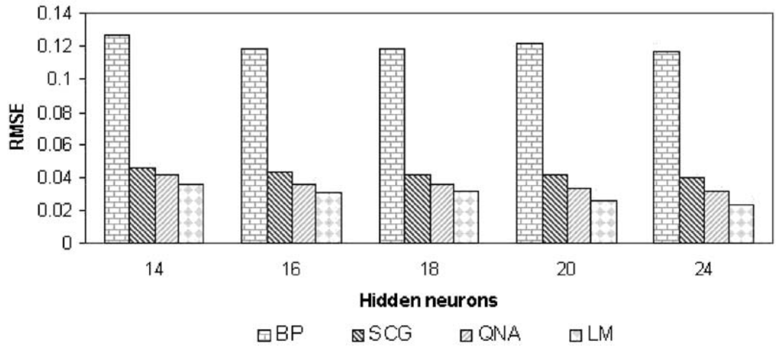


Fig. 5. Architecture variation: waste water time series training performance for different training algorithms.

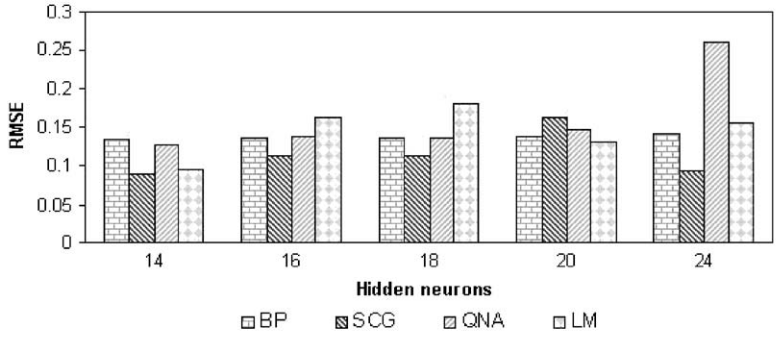


Fig. 6. Architecture variation: waste water time series generalization performance for different learning algorithms.

Table 4
Mackey-glass time series: training and generalization performance for different activation functions

Time series	Learning algorithm	Activation function	Root mean squared error	
			Training	Test
Mackey glass	BP	TSAF	0.0439	0.0437
		LSAF	0.0970	0.0950
	SCG	TSAF	0.0045	0.0045
		LSAF	0.0076	0.0074
	QNA	TSAF	0.0033	0.0034
		LSAF	0.0029	0.0029
	LM	TSAF	0.0009	0.0009
		LSAF	0.0009	0.0010

Table 5
Gas furnace series: training and generalization performance for different activation functions

Time series	Learning algorithm	Activation function	Root mean squared error	
			Training	Test
Gas furnace	BP	TSAF	0.0663	0.0970
		LSAF	0.0940	0.1025
	SCG	TSAF	0.0153	0.0367
		LSAF	0.0162	0.0367
	QNA	TSAF	0.0128	0.0516
		LSAF	0.0137	0.0420
	LM	TSAF	0.0100	0.1856
		LSAF	0.0089	0.1009

Figs. 9 and 10 the node transfer function also has an effect on the training speed and generalization performance. LM algorithm converged much faster and gave a better generalization performance when the node transfer function was changed to LSAF (Refer to Fig. 10(b)).

Waste water prediction series also showed a different generalization performance when the architecture was changed for the different learning algorithms (Refer to Table 3). BP's best generalization RMSE was 0.135 with 18 hidden neurons using TSAF and that of SCG, QNA and LM were 0.0900, 0.1271 and 0.095 with 14 neurons each, respectively. LM algorithm converged much faster and gave a better generalization performance when the node transfer function was changed to LSAF (Refer to Fig. 12(b)).

Table 6
Waste water time series: training and generalization performance for different activation functions

Time series	Learning algorithm	Activation function	Root mean squared error	
			Training	Test
Waste water	BP	TSAF	0.1169	0.1412
		LSAF	0.0156	0.1600
	SCG	TSAF	0.0400	0.0920
		LSAF	0.0420	0.0820
	QNA	TSAF	0.0316	0.4600
		LSAF	0.0256	0.2110
	LM	TSAF	0.0244	0.1560
		LSAF	0.2160	0.1770

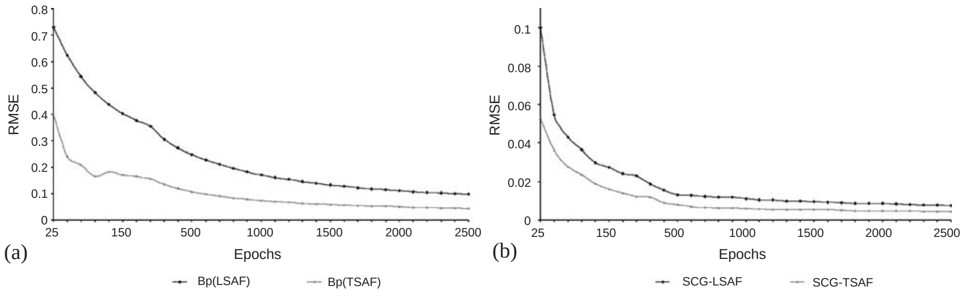


Fig. 7. Mackey-glass time series: convergence of training when node transfer function is changed: (a) BP training and (b) SCG algorithm.

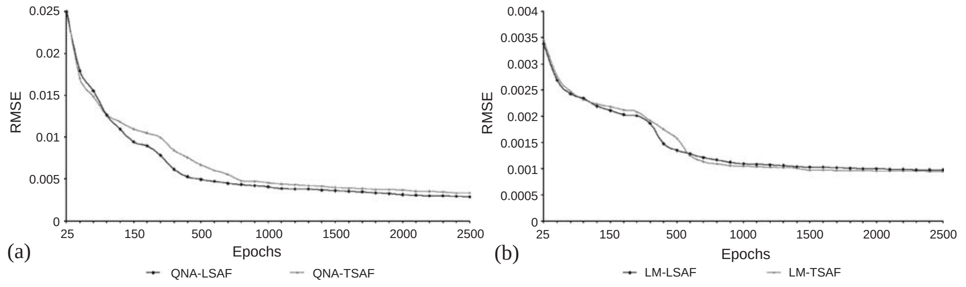


Fig. 8. Mackey-glass time series: convergence of training when node transfer function is changed: (a) QNA and (b) LM algorithm.

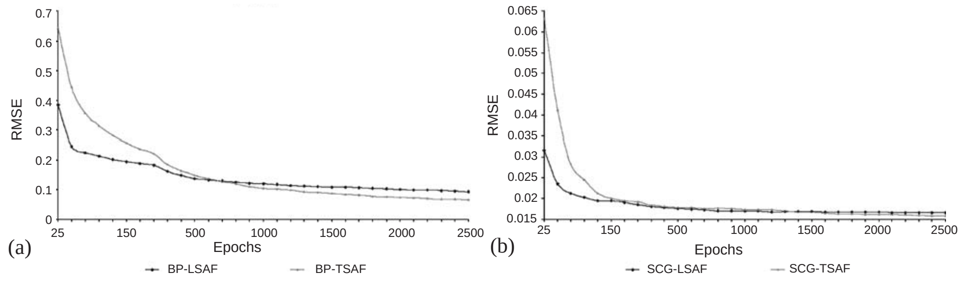


Fig. 9. Gas furnace time series: convergence of training when node transfer function is changed: (a) BP training and (b) SCG algorithm.

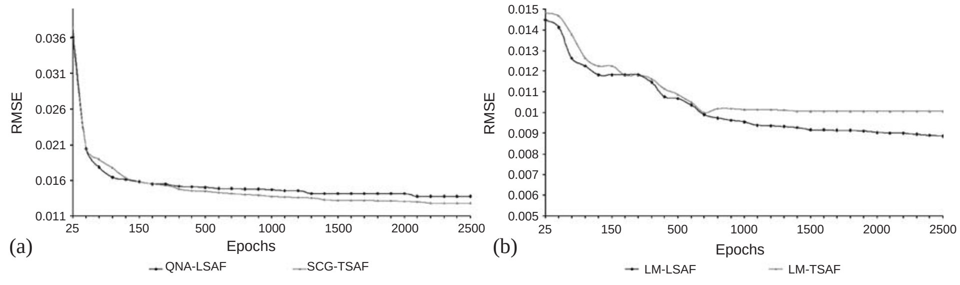


Fig. 10. Gas furnace series: convergence of training when node transfer function is changed: (a) QNA and (b) LM algorithm.

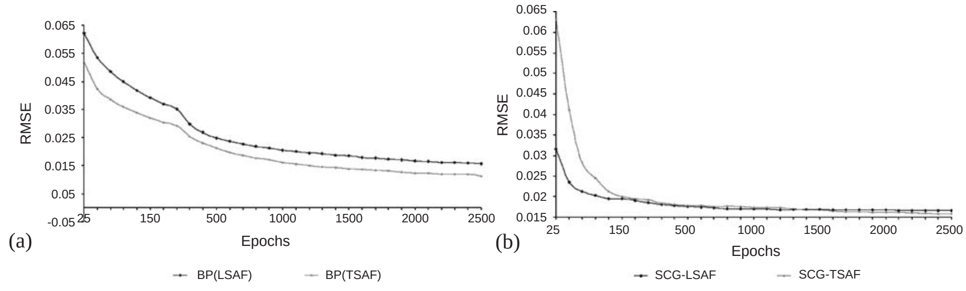


Fig. 11. Wastewater time series: convergence of training when node transfer function is changed: (a) BP training and (b) SCG algorithm.

In spite of computational complexity, LM performed well for Mackey-glass series. For gas furnace and waste water prediction SCG algorithm performed better. However, the speed of convergence of LM in all the three cases is worth noting. This leads us

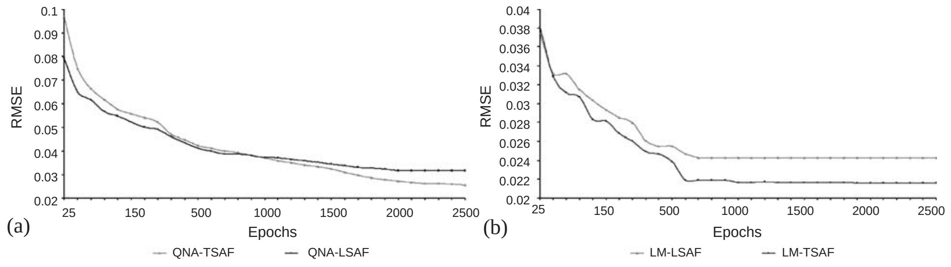


Fig. 12. Wastewater time series: convergence of training when node transfer function is changed: (a) QNA and (b) LM algorithm.

Table 7
Approximate computational load for the different time series using the different training algorithms

Learning algorithm	Hidden neurons	Computational load (billion flops)		
		Mackey glass	Gas furnace	Waste water
BP	14	0.625	0.142	0.301
	16	0.713	0.305	0.645
	18	0.800	0.488	0.880
	20	0.888	0.690	1.460
	24	1.064	0.932	1.970
SCG	14	1.256	0.286	0.604
	16	1.429	0.326	0.689
	18	1.605	0.366	0.774
	20	1.781	0.406	0.859
	24	2.133	0.486	1.029
QNA	14	2.570	0.679	1.910
	16	3.319	0.8899	2.582
	18	4.221	0.9000	3.388
	20	5.313	1.131	4.384
	24	7.989	2.193	6.925
LM	14	29.40	3.930	12.46
	16	57.51	8.355	27.72
	18	93.29	14.03	93.79
	20	137.83	21.10	118.53
	24	203.10	31.83	175.22

to the following questions:

- What is the optimal architecture (number of neurons and hidden layers) for a given problem?
- What node transfer function(s) should one choose?
- What is the optimal learning algorithm and its parameters?

From the above discussion it is clear that the selection of the topology of a network and the best learning algorithm and its parameters is a tedious task for designing an optimal ANN, which is smaller, faster and with a better generalization performance. EA is an adaptive search technique based on the principles and mechanisms of natural selection and survival of the fittest from natural evolution [32]. The interest in evolutionary search procedures for designing neural network topology has been growing in recent years as they can evolve towards the optimal architecture without outside interference, thus eliminating the tedious trial and error work of manually finding an optimal network.

3. Evolutionary algorithms

EAs are population based adaptive methods, which may be used to solve optimization problems, based on the genetic processes of biological organisms [31,32]. Over many generations, natural populations evolve according to the principles of natural selection and “Survival of the Fittest”, first clearly stated by Charles Darwin in “On the Origin of Species”. By mimicking this process, EAs are able to “evolve” solutions to real world problems, if they have been suitably encoded. The procedure may be written as the difference equation [32]:

$$x[t+1] = s(v(x[t])), \tag{44}$$

where $x(t)$ is the population at time t, v is a random operator, and s is the selection operator (Fig. 13).

3.1. Evolutionary artificial neural networks

Many of the conventional ANNs now being designed are statistically quite accurate but they still leave a bad taste with users who expect computers to solve their problems accurately. The important drawback is that the designer has to specify the number of neurons, their distribution over several layers and interconnection between them. Several methods have been proposed to automatically construct ANNs for reduction in network complexity that is to determine the appropriate number of hidden units, layers, etc. Topological optimization algorithms such as Extentron [9], Upstart [35], Pruning [63,75] and Cascade Correlation [29], etc. got its own limitations.

The interest in evolutionary search procedures for designing ANN architecture has been growing in recent years as they can evolve towards the optimal architecture without outside interference, thus eliminating the tedious trial and error work of manually finding an optimal network [2,6,7,14,18–21,37,50,60,69,70,77,81,83–85]. The advantage of the automatic design over the manual design becomes clearer as the complexity of ANN increases. EANNs provide a general framework for investigating various aspects of simulated evolution and learning [10,14,15,50,52].

3.1.1. General framework for EANNs

In EANNs evolution can be introduced at various levels. At the lowest level, evolution can be introduced into weight training, where ANN weights are evolved. At the

```
1. Generate the initial population P(0) at random and set i=0;
2. Repeat until the number of iterations or time has reached or the
   population has converged.
     • Evaluate the fitness of each individual in P(i)
     • Select parents from P(i) based on their fitness in P (i)
     • Apply reproduction operators to the parents and produce
       offspring, the next generation, P(i+1) is obtained from the
       offspring and possibly parents.
```

Fig. 13. Pseudo-code of an EA.

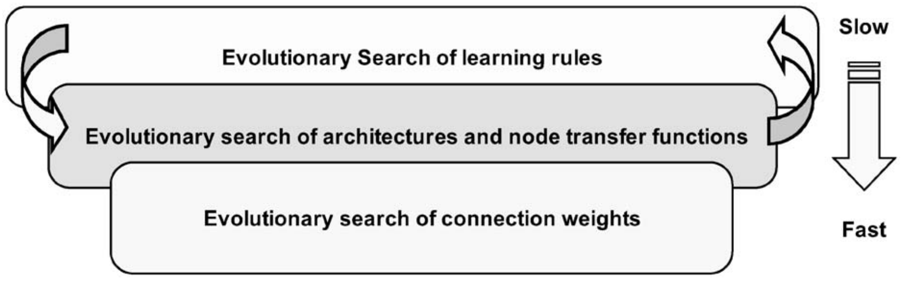

Evolutionary Search of learning rules
Evolutionary search of architectures and node transfer functions
Evolutionary search of connection weights
Slow
Fast

Fig. 14. A General Framework for EANNs.

next higher level, evolution can be introduced into neural network architecture adaptation, where the architecture (number of hidden layers, no of hidden neurons and node transfer functions) is evolved. At the highest level, evolution can be introduced into the learning mechanism. A general framework of EANNs which includes the above three levels of evolution is given in Fig. 14 [2,6].

From the point of view of engineering, the decision on the level of evolution depends on what kind of prior knowledge is available. If there is more prior knowledge about EANN's architectures than that about their learning rules or a particular class of architectures is pursued, it is better to implement the evolution of architectures at the highest level because such knowledge can be used to reduce the search space and the lower level evolution of learning rules can be more biased towards this kind of architectures. On the other hand, the evolution of learning rules should be at the highest level if there is more prior knowledge about them available or there is a special interest in certain type of learning rules.

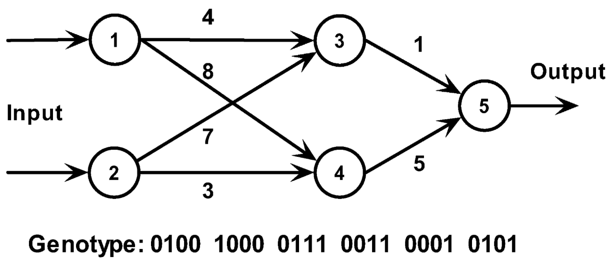


Fig. 15. Connection weight chromosome encoding using binary representation.

3.1.1.1. Evolutionary search of connection weights The shortcomings of the BP algorithm mentioned in Section 2.1 could be overcome if the training process is formulated as a global search of connection weights towards an optimal set defined by the EA. Optimal connection weights can be formulated as a global search problem wherein the architecture of the neural network is pre-defined and fixed during the evolution. Connection weights may be represented as binary strings represented by a certain length. The whole network is encoded by concatenation of all the connection weights of the network in the chromosome. A heuristic concerning the order of the concatenation is to put connection weights to the same node together. Fig. 15 illustrates the binary representation of connection weights wherein each weight is represented by 4 bits.

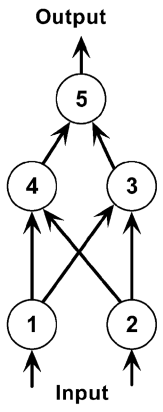
Real numbers have been proposed to represent connection weights directly [66]. A representation of the ANN could be (2.0, 6.0, 5.0, 1.0, 4.0, 10.0). However, proper genetic operators are to be chosen depending upon the representation used.

Evolutionary Search of connection weights can be formulated as follows:

(1) Generate an initial population of N weight chromosomes. Evaluate the fitness of each EANN depending on the problem.
(2) Depending on the fitness and using suitable selection methods reproduce a number of children for each individual in the current generation.
(3) Apply genetic operators to each child individual generated above and obtain the next generation.
(4) Check whether the network has achieved the required error rate or the specified number of generations has been reached. Go to Step 2.
(5) End.

While gradient based techniques are very much dependant on the initial setting of weights, the proposed algorithm can be considered generally much less sensitive to initial conditions. When compared to any gradient descent or second-order optimization technique that can only find local optimum in a neighborhood of the initial solution, EA always try to search for a global optimal solution. Performance by using the above approach will directly depend on the problem.

3.1.1.2. Evolutionary search of architectures evolutionary Evolutionary architecture adaptation can be achieved by constructive and destructive algorithms. Constructive



From / To	1	2	3	4	5	Bias	Gene
1	0	0	0	0	0	0	000000
2	0	0	0	0	0	0	000000
3	1	1	0	0	0	1	110001
4	1	1	0	0	0	1	110001
5	0	0	1	1	0	1	001101

Genotype: 000000 000000 110001 110001 001101

Fig. 16. Architecture chromosome using binary coding.

algorithms, which add complexity to the network starting from a very simple architecture until the entire network is able to learn the task [35,56,59]. Destructive algorithms start with large architectures and remove nodes and interconnections until the ANN is no longer able to perform its task [63,75]. Then the last removal is undone. Fig. 16 demonstrates how typical neural network architecture could be directly encoded and how the genotype is represented. For an optimal network, the required node transfer function (Gaussian, sigmoidal, etc.) can be formulated as a global search problem, which is evolved simultaneously with the search for architectures [51].

To minimize the size of the genotype string and improve scalability, when priori knowledge of the architecture is known it will be efficient to use some indirect coding (high level) schemes. For example, if two neighboring layers are fully connected then the architecture can be coded by simply using the number of layers and nodes. The blueprint representation is a popular indirect coding scheme where it assumes architecture consists of various segments or areas. Each segment or area will define a set of neurons, their spatial arrangement and their efferent connectivity. Several high level coding schemes like graph generation system [49], symbiotic adaptive neuro-evolution (SANE) [62,65], marker based genetic coding [36], L-systems [13], cellular encoding [38], fractal representation [58], etc. are some of the rugged techniques.

Global search of transfer function and the connectivity of the ANN using EA can be formulated as follows:

(1) The evolution of architectures has to be implemented such that the evolution of weight chromosomes are evolved at a faster rate, i.e. for every architecture chromosome, there will be several weight chromosomes evolving at a faster time scale.
(2) Generate an initial population of N architecture chromosomes. Evaluate the fitness of each EANN depending on the problem.

(3) Depending on the fitness and using suitable selection methods reproduce a number of children for each individual in the current generation.
(4) Apply genetic operators to each child individual generated above and obtain the next generation.
(5) Check whether the network has achieved the required error rate or the specified number of generations has been reached. Go to Step 3.
(6) End.

3.1.1.3. Evolutionary search of learning rules For the neural network to be fully optimal the learning rules are to be adapted dynamically according to its architecture and the given problem. Deciding the learning rate and momentum can be considered as the first attempt of learning rules [48]. The basic learning rule can be generalized by the function

$$\Delta w(t) = \sum_{k=1}^{n} \sum_{i_1,i_2,\ldots,i_k=1}^{n} \left(\theta_{i_1,i_2,\ldots,i_k} \prod_{j=1}^{k} x_{ij}(t-1) \right), \tag{45}$$

where t is the time, Δw is the weight change, $x_1, x_2, \ldots, x_n$ are local variables and the θ's are the real values coefficients which will be determined by the global search algorithm. In the above equation different values of θ's determine different learning rules. The above equation is arrived based on the assumption that the same rule is applicable at every node of the network and the weight updating is only dependent on the input/output activations and the connection weights on a particular node. Genotypes (θ's) can be encoded as real-valued coefficients and the global search for learning rules using the hybrid algorithm can be formulated as follows:

(1) The evolution of learning rules has to be implemented such that the evolution of architecture chromosomes are evolved at a faster rate i.e. for every learning rule chromosome, there will be several architecture chromosomes evolving at a faster time scale.
(2) Generate an initial population of N learning rules. Evaluate the fitness of each EANN depending on the problem.
(3) Depending on the fitness and using suitable selection methods reproduce a number of children for each individual in the current generation.
(4) Apply genetic operators to each child individual generated above and obtain the next generation.
(5) Check whether the network has achieved the required error rate or the specified number of generations has been reached. Go to Step 3.
(6) End.

Several researches have been going on about how to formulate different optimal learning rules [4,6,11,33,82]. The adaptive adjustment of BP algorithm's parameters, such as the learning rate and momentum, through evolution could be considered as the first

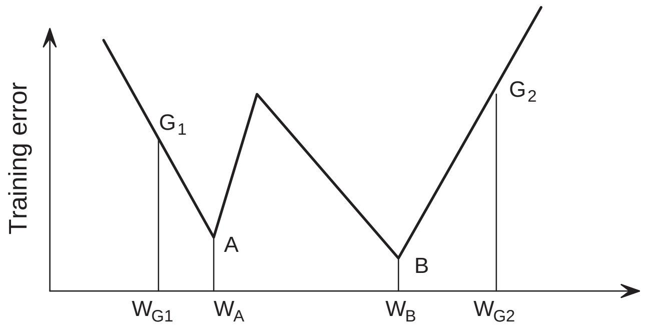


Fig. 17. Fine tuning of weights using meta-learning.

attempt of the evolution of learning rules [40]. Chalmers [23] defined the form of learning rules as a linear function of four local variables and their six pair wise products [11,33]. Global optimization of neural network has been widely addressed using several other techniques [22,28,34,64,71–74,86]. Sexton et al. [72] used simulated annealing algorithm for optimization of learning. For optimization of the neural network learning, in many cases a pre-defined architecture was used and in a few cases architectures were evolved together. No work has been reported to the best of our knowledge, where the network is fully automated (interaction of the different evolutionary search mechanisms) using the generic framework mentioned in Section 3.1. Many a times, the search space is narrowed down by pre-defined architecture, node transfer functions and learning rules.

3.2. Meta learning evolutionary artificial neural networks

Experimental evidence had indicated cases where EA are inefficient at fine tuning solutions, but better at finding global basins of attraction [2,82,43,80]. The efficiency of evolutionary training can be improved significantly by incorporating a local search procedure into the evolution. EA are used to first locate a good region in the space and then a local search procedure is used to find a near optimal solution in this region. It is interesting to consider finding good initial weights as locating a good region in the space. Defining that the basin of attraction of a local minimum is composed of all the points, sets of weights in this case, which can converge to the local minimum through a local search algorithm, then a global minimum can easily be found by the local search algorithm if the EA can locate any point, i.e., a set of initial weights, in the basin of attraction of the global minimum. Referring to Fig. 17, G_1 and G_2 could be considered as the initial weights as located by the evolutionary search and W_A and W_B the corresponding final weights fine-tuned by the meta-learning technique.

Fig. 18 illustrates the general interaction mechanism with the learning mechanism of the EANN evolving at the highest level on the slowest time scale. All the randomly

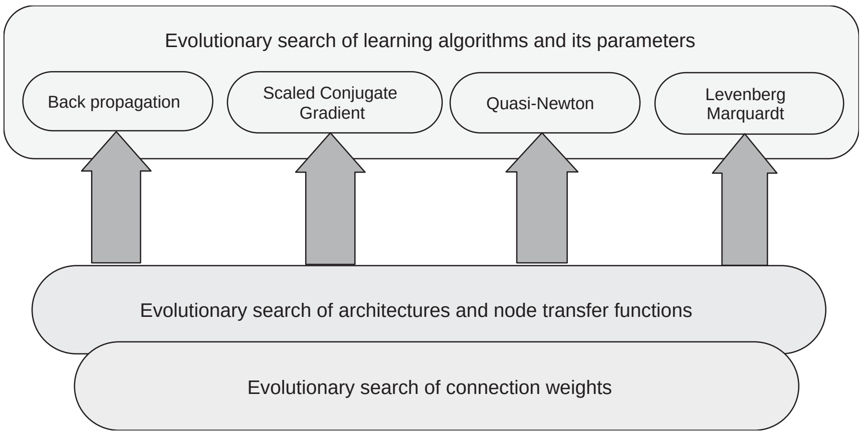

Evolutionary search of learning algorithms and its parameters
Back propagation
Scaled Conjugate Gradient
Quasi-Newton
Levenberg Marquardt
Evolutionary search of architectures and node transfer functions
Evolutionary search of connection weights

Fig. 18. Interaction of various evolutionary search mechanisms.

1. *Set t=0 and randomly generate an initial population of neural networks with architectures, node transfer functions and connection weights assigned at random.*
2. *In a parallel mode, evaluate fitness of each ANN using BP/SCG/QNA and LM*
3. *Based on fitness value, select parents for reproduction*
4. *Apply mutation to the parents and produce offspring (s) for next generation. Refill the population back to the defined size.*
5. *Repeat step 2*
6. *STOP when the required solution is found or number of iterations has reached the required limit.*

Fig. 19. Meta-learning algorithm for EANNs.

generated architecture of the initial population are trained by four different learning algorithms (BP, SCG, QNA and LM) and evolved in a parallel environment. Parameters controlling the performance of the learning algorithm will be adapted (example, learning rate and momentum for BP) according to the problem [2,6]. Fig. 19 depicts the basic algorithm of proposed meta-learning EANN. Architecture of the chromosome is depicted in Fig. 20.

3.2.1. MLEANN: experimentation setup

We have applied the proposed meta learning framework to the three-time series prediction problems discussed in Section 2.2. For performance comparison, we used the

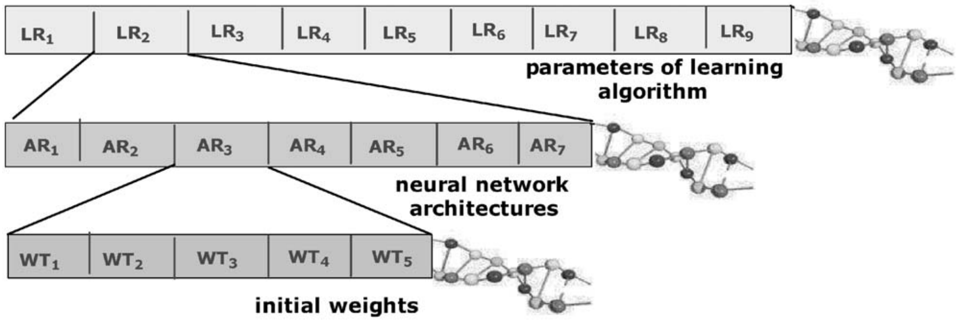


Fig. 20. Chromosome representation of the proposed MLEANN framework.

Table 8
Parameters used for evolutionary design of ANNs

Population size	40
Maximum no of generations	40
Number of hidden nodes	• Experiment 1: 5–16 hidden nodes • Experiment 2: maximum 4 neurons
Activation functions	tanh (T), logistic (L), sigmoidal (S), tanh-sigmoidal (T^*), log-sigmoidal (L^*)
Output neuron	linear
Training epochs	500
Initialization of weights	± 0.3
Ranked based selection	0.50
Elitism	5%
Mutation rate	0.40

same set of training and test data that were used for experimentations with conventional design of neural networks. For performance evaluation, the parameters used in our experiments were set to be the same for all the 3 problems. Fitness value is calculated based on the RMSE achieved on the test set. In this experiment, we have considered the best-evolved neural network as the best individual of the last generation. As the learning process is evolved separately, user has the option to pick the best neural network (e.g. less RMSE or less computational expensive, etc.) among the four learning algorithms. All the genotypes were represented using binary coding and the initial populations were randomly generated based on the following parameters shown in Table 8. The parameter settings, which were evolved for the different learning algorithms, are illustrated in Table 9. The parameter settings mentioned in Tables 8 and 9 were finalized after a few trail and error approaches. We also investigated the performance of the proposed method with a restriction of architecture (no of hidden neurons). We set a maximum number of four hidden neurons and evaluated the learning performance. The experiments were repeated three times and the worst RMSE values are reported.

Table 9
Parameters settings of the learning algorithms

Learning algorithm	Parameter	Setting
BP	Learning rate	0.25–0.05
	Momentum	0.25–0.05
SCG algorithm	Change in weight for second derivative approximation	0–0.0001
	Regulating the indefiniteness of the Hessian	0–1.0E-06
QNA	Step lengths	1.0E-06–100
	Limits on step sizes	0.1–0.6
	Scale factor to determine performance	0.001–0.003
	Scale factor to determine step size	0.1–0.4
LM	Learning rate	0.001–0.02

Table 10
Performance comparison between MLEANN (without architecture restriction) and ANN

Time series	Learn algo.	EANN			ANN	
		RMSE		Architecture	RMSE	Architecture
		Training	Test			
Mackey glass	BP	0.0072	0.0077	7 T, 3 L	0.0437	24 T*
	SCG	0.0030	0.0031	11 T	0.0045	24 T*
	QNA	0.0024	0.0027	6 T, 4 T*	0.0034	24 T*
	LM	0.0004	0.0004[a]	8 T, 2 T* 1 L*	0.0009	24 T*
Gas furnace	BP	0.0159	0.0358	8 T	0.0766	18 T*
	SCG	0.0110	0.0210[a]	8 T, 2 T*	0.0330	16 T*
	QNA	0.0115	0.0256	7 T, 2 L*	0.0376	18 T*
	LM	0.0120	0.0223	6 T, 1 L, 1 T*	0.0451	14 T*
Waste water	BP	0.0441	0.0547	6 T, 5 T*, 1 L	0.1360	16 T*
	SCG	0.0457	0.0579	6 T, 4 L*	0.0820	14 T*
	QNA	0.0673	0.0823	5 T, 5 TS	0.1276	14 T*
	LM	0.0425	0.0521[a]	8 T, 1 LS	0.0951	14 T*

[a]Lowest RMSE error.

3.2.2. MLEANN: experimentation results

Table 10 displays empirical values of RMSE on test data for the three time series problems without architecture restriction. For comparison purposes, test set RMSE values using conventional design techniques are also presented in Table 10 (adapted from Tables 1–3). Convergence of test set RMSE for the three time series (without architecture restriction) is depicted in Figs. 24–26. Table 11 illustrates the RMSE

Table 11
Performance results and run time comparison of MLEANN

Time series	Learn algo.	EANN			Run time in minutes	
		RMSE		Architecture	A[a]	B[b]
		Training	Test			
Mackey glass	BP	0.0166	0.0168	4 T	1181	288
	SCG	0.0062	0.0067	3 T, 1 T*	2066	504
	QNA	0.0059	0.0058	3 T*, 1 L	2169	528
	LM	0.0056	0.0061[c]	2 L*, 2 T*	2463	602
Gas furnace	BP	0.0189	0.0371	3 L	305	62
	SCG	0.0179	0.0295	1 T*, 2 L	629	121
	QNA	0.0156	0.0295	2 T*, 1 L*, 1 L	661	128
	LM	0.0181	0.0290[c]	1 T, 1 L, 1 T*	696	132
Waste water	BP	0.0647	0.0639	2 T, 2 T*	702	146
	SCG	0.0580	0.0600	2 T*, 1 T, 1 L	1254	267
	QNA	0.0590	0.0596	3 T*, 1 L*	1291	279
	LM	0.0567	0.0591[c]	2 L, 1 T, 1 T*	1176	294

[a]Without architecture restriction.
[b]With architecture restriction.
[c]Lowest RMSE error.

Table 12
Performance comparison between MLEANN and neuro-fuzzy systems

Time series	RMSE					
	EANN		Mamdani–NF		Takagi Sugeno–NF	
	Training	Test	Training	Test	Training	Test
Mackey glass	0.0004	0.0004	0.0023	0.0042	0.0019	0.0018
Gas furnace	0.0110	0.0210	0.0140	0.0490	0.0137	0.0570
Waste water	0.0425	0.0521	0.0019	0.0750	0.0530	0.0810

values on training/test set data using the meta-learning technique when the architecture restriction was imposed. Run times for the two different experimentations are also presented.

3.2.3. Comparison with neuro-fuzzy systems

In this section we compare the performance of MLEANN (RMSE values on training and test sets) with two popular neuro-fuzzy models. The neuro-fuzzy models [1] considered were dynamic evolving fuzzy neural networks (dmEFuNN) [24,47] implementing a Mamdani fuzzy inference system [55] and an Adaptive Neuro-Fuzzy Inference System (ANFIS) [42] implementing a Takagi–Sugeno fuzzy inference system [76]. The same training and test sets of the three time series were used to compare the performance with the neuro-fuzzy systems. The empirical results are depicted in Table 12.

4. Discussions and conclusions

Table 10 shows comparative performance between MLEANN and a conventional ANN without any architecture restriction. For Mackey-glass series (Fig. 21), using 500 epochs of BP learning, RMSE on test set was reduced by 82% (BP), 31% (SCG), 29% (QNA) and 56% (LM). At the same time, number of hidden neurons got reduced by approximately 58% (BP), 54% (SCG), 58% (QNA) and 55% for LM. LM algorithm gave the best RMSE error on test set (0.0004) even though it is highly computational expensive as demonstrated in Table 7.

For the gas furnace time series (Fig. 22), RMSE on test set was reduced by 53% (BP), 36% (SCG), 69% (QNA) and 73% (LM). Savings in hidden neurons amounted to 55% (BP), 37% (SCG), 50% (QNA) and 55% (LM). SCG training gave the best RMSE value (0.0210) for gas furnace series.

For the wastewater time series (Fig. 23), RMSE on test set was reduced by 60% (BP), 29% (SCG), 35% (QNA) and 45% (LM). Savings in hidden neurons amounted to 25% (BP), 29% (SCG), 29% (QNA) and 36% (LM). LM learning gave the best RMSE value (0.0521) for wastewater series.

To have an empirical comparison, we deliberately terminated the local search after 500 epochs (regardless of early stopping in some cases) for all the training algorithms. In some cases the generalization performance could have been further improved. As depicted in Table 11, our experimentations with limited architecture also reveal the efficiency of MLEANN technique. The gas furnace time series and wastewater series could be learned just with 4 hidden neurons using LM algorithm. However, for Mackey-glass series the results were not that encouraging when compared with the conventional design using 24 hidden neurons. Perhaps Mackey-glass series requires more hidden neurons to learn the problem within the required accuracy. Table 12 depicts empirical comparison between two popular neuro-fuzzy systems. As evident, MLEANN has outperformed both neuro fuzzy models in terms of the lowest RMSE

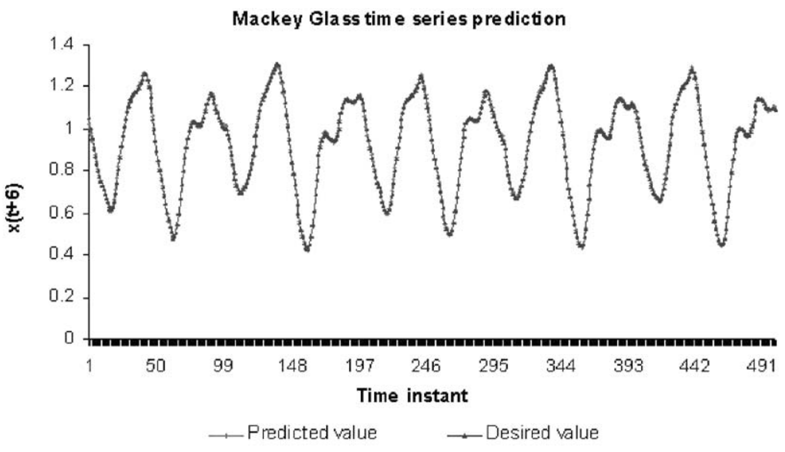


Fig. 21. Test results using 500 epochs BP meta learning for Mackey-glass series.

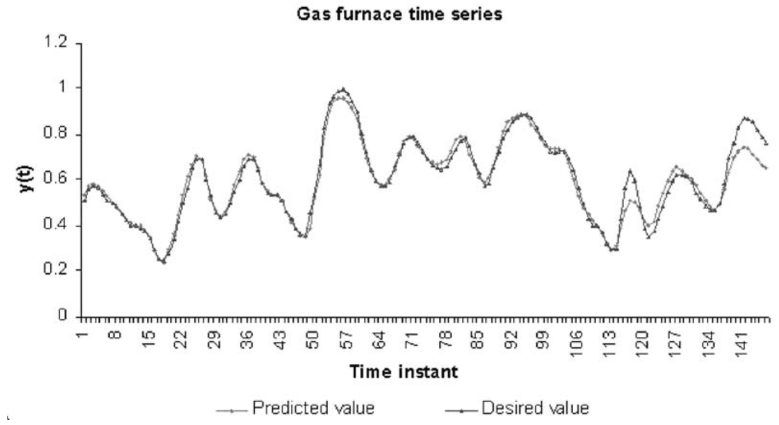


Fig. 22. Test results using 500 epochs BP meta-learning for gas furnace series.

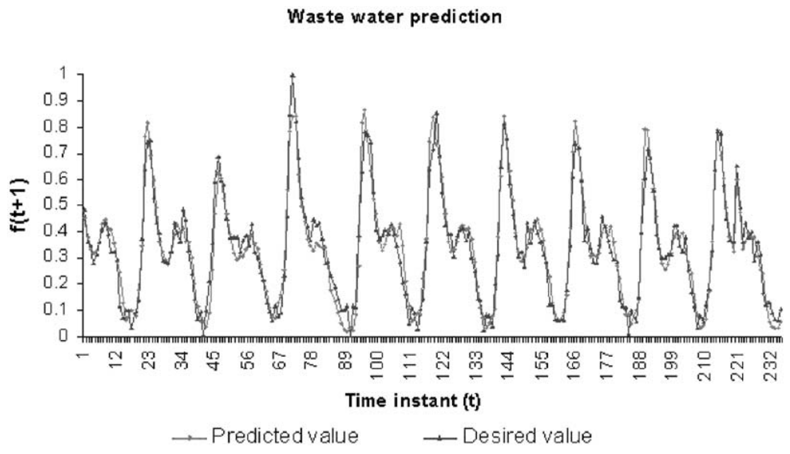


Fig. 23. Test results using 500 epochs BP meta-learning for waste water flow series.

values on test set for all the three time series. Selection of the architecture (number of layers, hidden neurons, activation functions and connection weights) of a network and correct learning algorithm is a tedious task for designing an optimal ANN. Moreover, for critical applications and hardware implementations optimal design often becomes a necessity. In this paper, we have formulated and explored; MLEANN: an adaptive computational framework based on evolutionary computation for automatic design of optimal ANNs. Empirical results are promising and show the importance and efficacy of the technique.

In MLEANN, our work was mostly concentrated on the evolutionary search of optimal learning algorithms for feedforward neural networks. Similar approach could be

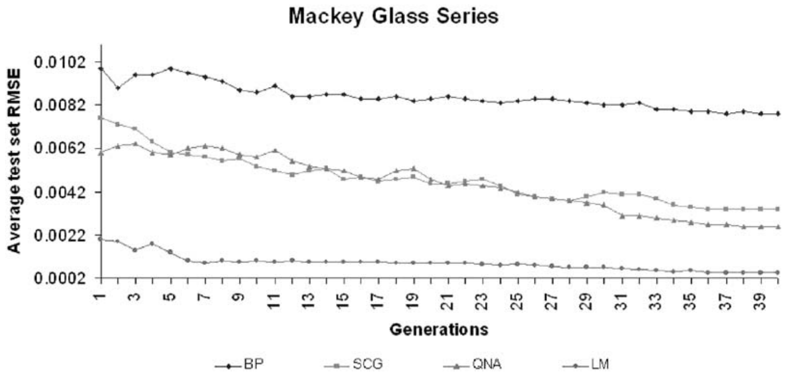


Fig. 24. Mackey-glass time series: average test set RMSE values during the 40 generations and meta-learning.

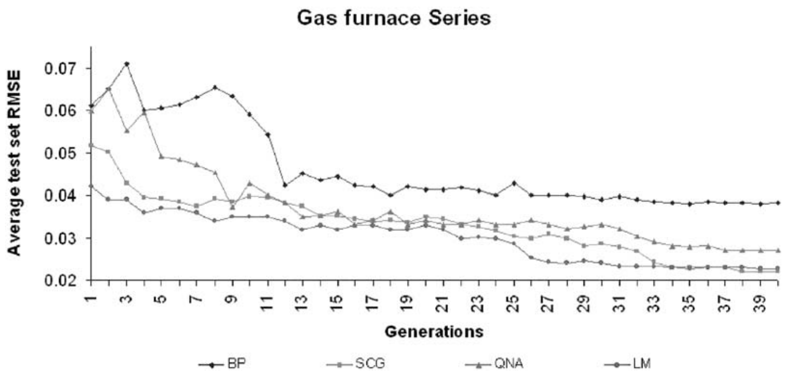


Fig. 25. Gas furnace time series: average test set RMSE values during the 40 generations and meta-learning.

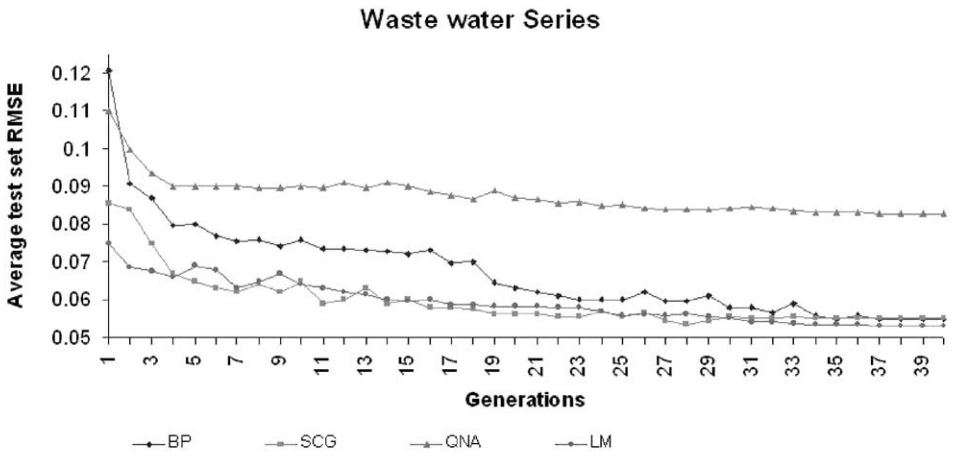


Fig. 26. Wastewater time series: average test set RMSE values during the 40 generations and meta-learning.

used for optimizing recurrent neural networks and other connectionist networks. For the evolutionary search of architectures, it will be interesting to model as co-evolving [27] sub-networks instead of evolving the whole network. Further, it will be worthwhile to explore the whole population information of the final generation for deciding the best solution. We used a fixed chromosome structure (direct encoding technique) to represent the connection weights, architecture, learning algorithms and its parameters. As size of the network increases, the chromosome size grows. Moreover, implementation of crossover is often difficult due to production of non-functional offspring's. Parameterized encoding overcomes the problems with direct encoding but the search of architectures is restricted to layers. In the grammatical encoding rewriting grammar is encoded. So the success will depend on the coding of grammar (rules). Cellular configuration might be helpful to explore the architecture of neural networks more efficiently. Gutierrez et al. [39] has shown that their cellular automata technique performed better than direct coding.

Acknowledgements

The author wishes to thank the anonymous reviewers for their valuable comments.

References

[1] A. Abraham, Neuro-fuzzy systems: state-of-the-art modeling techniques, in: M. Jose, P. Alberto (Eds.), Connectionist Models of Neurons, Learning Processes, and Artificial Intelligence, Springer, Germany, Granada, Spain, 2001, pp. 269–276.

[2] A. Abraham, Optimization of evolutionary neural networks using hybrid learning algorithms, IEEE 2002 Joint International Conference on Neural Networks, Vol. 3, IEEE Press, New York, 2002, pp. 2797–2802.

[3] A. Abraham, B. Nath, Failure prediction of critical electronic systems in power plants using artificial neural networks, in: M. Isreb (Ed.), Proceedings of First International Power & Energy Conference, Australia, December 1999, ISBN 0732 620 945.

[4] A. Abraham, B. Nath, Optimal design of neural nets using hybrid algorithms, in: Proceedings of Sixth Pacific Rim International Conference on Artificial Intelligence (PRICAI 2000), Melbourne, Australia, 2000, pp. 510–520.

[5] A. Abraham, B. Nath, Artificial neural networks for intelligent real time power quality monitoring systems, in: M. Isreb (Ed.), Proceedings of First International Power & Energy Conference, Australia, December 1999, 2000, ISBN 0732 620 945.

[6] A. Abraham, B. Nath, ALEC—an adaptive learning framework for optimizing artificial neural networks, in: N.A. Vassil, et al., (Eds.), Computational Science, Springer, Germany, San Francisco, USA, 2001, pp. 171–180.

[7] P.J. Angeline, G.B. Saunders, J.B. Pollack, An evolutionary algorithm that evolves recurrent neural networks, IEEE Trans. Neural Networks 5 (1) (1994) 54–65.

[8] P. Auer, M. Herbster, M. Warmuth, Exponentially many local minima for single neurons, in: D. Touretzky, et al., (Eds.), Advances in Neural Information Processing Systems, Vol. 8, MIT Press, Cambridge, MA, 1996, pp. 316–322.

[9] P.T. Baffles, J.M. Zelle, Growing layers of perceptrons: introducing the exentron algorithm, Proceedings on the International Joint Conference on Neural Networks, Vol. 2, Beijing, China, 1992, pp. 392–397.

[10] K. Balakrishnan, V. Honawar, Some experiments in evolutionary synthesis of robotic neurocontrollers, Proceedings of World Congress on Neural Networks, San Diego, USA, 1996, pp. 1035–1040.
[11] J. Baxter, The evolution of learning algorithms for artificial neural networks, in: D. Green, T. Bossomaier (Eds.), Complex Systems, IOS Press, Amsterdam, 1992, pp. 313–326.
[12] C.M. Bishop, Neural Networks for Pattern Recognition, Oxford Press, Oxford, 1995.
[13] E.J.W. Boers, M.V. Borst, I.G. Sprinkhuizen-Kuyper, Artificial neural nets and genetic algorithms, in: D.W. Pearson, et al., (Eds.), Proceedings of the International Conference in Ales, France, Springer, New York, 1995, pp. 333–336.
[14] E.J.W. Boers, M.V. Borst, I.G. Sprinkhuizen-Kuyper, Evolving artificial neural networks using the Baldwin effect, in: D.W. Pearson, et al., (Eds.), Artificial Neural Nets and Genetic Algorithms, Proceedings of the International Conference in Alès, France, Springer, New York, 1995, pp. 333–336.
[15] E.J.W. Boers, H. Kuiper, B.L.M. Happel, I.G. Sprinkhuizen-Kuyper, Designing modular artificial neural networks, in: H.A. Wijshoff (Ed.), Proceedings of Computing Science in The Netherlands, Amsterdam, The Netherlands, 1993, pp. 87–96.
[16] H.A. Bourlard, N. Morgan, Connectionist Speech Recognition: A Hybrid Approach, Kluwer Academic Publishers, Boston, Dordrecht, 1994.
[17] G.E.P. Box, G.M. Jenkins, Time Series Analysis, Forecasting and Control, Holden Day, San Francisco, 1970.
[18] J. Branke, U. Kohlmorgen, H. Schmeck, A distributed genetic algorithm improving the generalization behavior of neural networks, in: N. Lavrac et al. (Eds.), Proceedings of the European Conference on Machine Learning, 1995, pp. 107–112.
[19] H. Braun, On optimizing large neural networks (multilayer perceptrons) by learning and evolution, in: Proceedings of the Third International Congress on Industrial and Applied Mathematics, ICIAM, Hamburg, Germany, 1995.
[20] H. Braun, J. Weisbrod, Evolving neural networks for application oriented problems, in: D.B. Fogel (Ed.), Proceedings of the Second Conference on Evolutionary Programming, USA, 1993.
[21] H. Braun, P. Zagorski, ENZO-M—a hybrid approach for optimizing neural networks by evolution and learning, in: Y. Davidor et al. (Eds.), Proceedings of the Third International Conference on Parallel Problem Solving from Nature, Israel, 1994.
[22] P.A. Castillo, J.J. Merelo, A. Prieto, V. Rivas, G. Romero, G-Prop: global optimization of multilayer perceptrons using GAs, Neurocomputing 35 (2000) 149–163.
[23] D.J. Chalmers, The evolution of learning: an experiment in genetic connectionism, in: D.S. Touretzky, et al., (Eds.), Proceedings of the 1990 Connectionist Models Summer School, Morgan Kaufmann, Los Altos, CA, 1990, pp. 81–90.
[24] V. Cherkassky, Fuzzy inference systems: a critical review, in: O. Kayak, L.A. Zadeh, et al. (Eds.), Computational Intelligence: Soft Computing and Fuzzy-Neuro Integration with Applications, Springer, Berlin, 1998, pp. 177–197.
[25] E.K.P. Chong, S.H. Zak, An Introduction to Optimization, Wiley, New York, 1996.
[26] A. Cichocki, R. Unbehauen, Neural Networks for Optimization and Signal Processing, Wiley, New York, 1993.
[27] P.J. Darwen, Co-evolutionary learning by automatic modularization with speciation, Ph.D. Thesis, University of New South Wales, 1996.
[28] W. Duch, J. Korczak, Optimization and global minimization methods suitable for neural networks, Neural Computing Surveys, Lawrence Erlbaum Associates Inc, USA, Vol. 2, 1999, pp. 163–212.
[29] S.E. Fahlman, C. Lebiere, The cascade—correlation learning architecture, in: D. Tourretzky (Ed.), Advances in Neural Information Processing Systems, Morgan Kaufmann, Los Altos, CA, 1990, pp. 524–532.
[30] T.L. Fine, Feedforward Neural Network Methodology, Springer, New York, 1999.
[31] D.B. Fogel, The advantages of evolutionary computation, in: D. Lundh, B. Olsson, A. Narayanan (Eds.), Bio-Computing and Emergent Computation, World Scientific Press, Singapore, Sköve, Sweden, 1997, pp. 1–11.
[32] D. Fogel, Evolutionary Computation: Towards a New Philosophy of Machine Intelligence, 2nd Edition, IEEE Press, New York, 1999.

[33] J.F. Fontanari, R. Meir, Evolving a learning algorithm for the binary perceptron, Network 2 (1991) 353–359.
[34] M. Forti, A note on neural networks with multiple equilibrium points, IEEE Trans. Circuits Syst.-I: Fundam. Theory 43 (5) (1996) 487.
[35] M. Frean, The upstart algorithm: a method for constructing and training feed forward neural networks, Neural Comput. 2 (1990) 198–209.
[36] B. Fullmer, R. Miikkulainen, Using marker-based genetic encoding of neural networks to evolve finite-state behaviour, in: F.J. Varela, P. Bourgine (Eds.), Proceedings of the First European Conference on Artificial Life, France, 1992, pp. 255–262.
[37] N. Funabiki, J. Kitamichi, S. Nishikawa, An evolutionary neural network approach for module orientation problems, IEEE Trans. Syst., Man Cybernet.—Part B: Cybernet. 28 (6) (1998) 849–855.
[38] F. Grau, Genetic synthesis of boolean neural networks with a cell rewriting developmental process, in: D. Whitely, J.D. Schaffer (Eds.), Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks, IEEE Computer Society Press, CA, 1992, pp. 55–74.
[39] G. Gutierrez, P. Isasi, J.M. Molina, A. Sanchis, I.M. Galvan, Evolutionary cellular configurations for designing feedforward neural network architectures, in: M. Jose, et al. (Eds.), Connectionist Models of Neurons, Learning Processes, and Artificial Intelligence, Lecture Notes in Computer Science, Vol. 2084, Springer, Germany, 2001, pp. 514–521.
[40] S.A. Harp, T. Samad, A. Guha, Towards the genetic synthesis of neural networks, in: J.D. Schaffer (Ed.), Proceedings of the Third International Conference on Genetic Algorithms and their Applications, Morgan Kaufmann, Los Altos, CA, 1989, pp. 360–369.
[41] K.U. Hofgen, Computational limits on training sigmoidal neural networks, Inform. Process. Lett. 46 (1993) 269–274.
[42] J.S.R. Jang, ANFIS: adaptive network based fuzzy inference systems, IEEE Trans. Syst. Man Cybernet. Vol. 23, 1993, pp. 665–685.
[43] G.A. Jayalakshmi, S. Sathiamoorthy, R. Rajaram, An hybrid genetic algorithm—a new approach to solve traveling salesman problem, Int. J. Comput. Eng. Sci. 2 (2) (2001) 339–355.
[44] L. Jones, The computational intractability of training sigmoidal neural networks, IEEE Trans. Inform. Theory 43 (1997) 167–173.
[45] S. Judd, Neural Network Design and the Complexity of Learning, MIT Press, Cambridge, MA, 1990.
[46] N. Kasabov, Foundations of Neural Networks, Fuzzy Systems and Knowledge Engineering, The MIT Press, Cambridge, MA, 1996.
[47] N. Kasabov, Evolving fuzzy neural networks—algorithms, applications and biological motivation, in: T. Yamakawa, G. Matsumoto (Eds.), Methodologies for the Conception, Design and Application of Soft Computing, World Scientific, Singapore, 1998, pp. 271–274.
[48] H.B. Kim, S.H. Jung, T.G. Kim, K.H. Park, Fast learning method for back-propagation neural network by evolutionary adaptation of learning rates, Neurocomputing 11 (1) (1996) 101–106.
[49] H. Kitano, Designing neural networks using genetic algorithms with graph generation system, Complex Syst. 4 (4) (1990) 461–476.
[50] J.N. Kok, E. Marchiori, M. Marchiori, C. Rossi, Evolutionary training of CLP-constrained neural networks, Second International Conference on Practical Application of Constraint Technology, London, UK, 1996, pp. 129–142.
[51] Y. Liu, X. Yao, Evolutionary design of artificial neural networks with different node transfer functions, Proceedings of the Third IEEE International Conference on Evolutionary Computation, Nagoya, Japan, 1996, pp. 670–675.
[52] Y. Liu, X. Yao, Towards designing neural network ensembles by evolution, in: A.E. Eiben, M. Schoenauer, H.P. Schwefel (Eds.), Proceedings of the Fifth International Conference on Parallel Problem Solving from Nature (PPSN-V), Lecture Notes in Computer Science, Vol. 1498, Springer, Berlin, 1998, pp. 623–632.
[53] M.C. Mackey, L. Glass, Oscillation and chaos in physiological control systems, Science 197 (1977) 287–289.
[54] W.G. Macready, D.H. Wolpert, The No Free Lunch theorems, IEEE Trans. Evol. Comput. 1 (1) (1997) 67–82.

[55] E.H. Mamdani, S. Assilian, An experiment in linguistic synthesis with a fuzzy logic controller, Int. J. Man-Mach. Stud. 7 (1) (1975) 1–13.

[56] F. Mascioli, G. Martinelli, A constructive algorithm for binary neural networks: the oil spot algorithm, IEEE Trans. Neural Networks 6 (3) (1995) 794–797.

[57] T. Masters, Signal and Image Processing with Neural Networks: A C++ Sourcebook, Wiley, New York, 1994.

[58] J.W.L. Merril, R.F. Port, Fractally configured neural networks, Neural Networks 4 (1) (1991) 53–60.

[59] M. Mezard, J.P. Nadal, Learning in feed forward layered networks: the Tiling algorithm, J. Phys. A 22 (1989) 2191–2204.

[60] G.F. Miller, P.M. Todd, S.U. Hedge, Designing neural networks using genetic algorithms, in: J.D. Schaffer (Ed.), Proceedings of the Third International Conference on Genetic Algorithms, Virginia, USA, 1989, pp. 379–384.

[61] A.F. Moller, A scaled conjugate gradient algorithm for fast supervised learning, Neural Networks (6) (1993) 525–533.

[62] D.E. Moriarty, R. Miikkulainen, Forming neural networks through efficient and adaptive coevolution, Evol. Comput. 5 (1997) 373–399.

[63] C.W. Omlin, C.L. Giles, Pruning recurrent neural networks for improved generalization performance, Technical Report No 93-6, CS Department, Rensselaer Institute, Troy, NY, 1993.

[64] V.V. Phansalkar, M.A.L. Thathachar, Local and global optimization algorithms for generalized learning automata, Neural Comput. 7 (1995) 950–973.

[65] D. Polani, R. Miikkulainen, Fast reinforcement learning through eugenic neuro-evolution, Technical Report AI99-277, Department of Computer Sciences, University of Texas at Austin, 1999.

[66] V.W. Porto, D.B. Fogel, L.J. Fogel, Alternative neural network training methods, IEEE Expert 10 (4) (1995) 16–22.

[67] A. Refenes (Ed.), Neural Networks in the Capital Markets, Wiley, Chichester, England, 1995.

[68] W. Schiffmann, M. Joost, R. Werner, Comparison of optimized backpropagation algorithms, Proceedings of the European Symposium on Artificial Neural Networks, Brussels, Belgium, 1993, pp. 97–104.

[69] A.V. Sebald, K. Chellapilla, On making problems evolutionarily friendly part 1: evolving the most convenient representations, The Seventh International Conference on Evolutionary Programming, EP98, San Diego, 1998, pp. 271–280.

[70] A.V. Sebald, K. Chellapilla, On making problems evolutionarily friendly part 2: evolving the most convenient representations, The Seventh International Conference on Evolutionary Programming, EP98, San Diego, 1998, pp. 281–290.

[71] R. Sexton, R. Dorsey, J. Johnson, Toward global optimization of neural networks: a comparison of the genetic algorithm and backpropagation, Decision Support Syst. 22 (1998) 171–185.

[72] R. Sexton, R. Dorsey, J. Johnson, Optimization of neural networks: a comparative analysis of the genetic algorithm and simulated annealing, Eur. J. Oper. Res. 114 (1999) 589–601.

[73] Y. Shang, B. Wah, Global optimization for neural network training, Computer 29 (1996) 45–55.

[74] K.K. Shukla, Raghunath, An efficient global algorithm for supervised training of neural networks, Comput. Electrical Eng. 25 (2) (1999) 195.

[75] S.W. Stepniewski, A.J. Keane, Pruning back-propagation neural networks using modern stochastic optimization techniques, Neural Comput. Appl. 5 (1997) 76–98.

[76] M. Sugeno, Industrial Applications of Fuzzy Control, Elsevier Science Pub Co., Amsterdam, 1985.

[77] A.P. Topchy, O.A. Lebedko, Neural network training by means of cooperative evolutionary search, Nucl. Instrum. Methods 389 (1–2) (1997) 240–241.

[78] M.M. Vidhyasagar, The Theory of Learning and Generalization, Springer, New York, 1997.

[79] A.S. Weigend, N.A. Gershenfeld (Eds.), Time Series Prediction: Forecasting the Future and Understanding the Past, Addison-Wesley, Reading, MA, 1994.

[80] D. Whitley, Modeling hybrid genetic algorithms, in: G. Winter, J. Periaux, M. Galan, P. Cuesta (Eds.), Genetic Algorithms in Engineering and Computer Science, Wiley, New York, 1995, pp. 191–201.

[81] X. Yao, Designing artificial neural networks using co-evolution, Proceedings of IEEE Singapore International Conference on Intelligent Control and Instrumentation, 1995, pp. 149–154.

[82] X. Yao, Evolving artificial neural networks, Proc. IEEE 87 (9 Suppl. 1) (1999) 423–1447.

[83] X. Yao, Y. Liu, A new evolutionary system for evolving artificial neural networks, IEEE Trans. Neural Networks 8 (3) (1997) 694–713.
[84] X. Yao, Y. Liu, Making use of population information in evolutionary artificial neural networks, IEEE Trans. Syst. Man Cybernet. Part B: Cybernet. 28 (3) (1998) 417–425.
[85] X. Yao, Y. Liu, Towards designing artificial neural networks by evolution, Appl. Math. Comput. 91 (1) (1998) 83–90.
[86] X.M. Zhang, Y.Q. Chen, Ray-guided global optimization method for training neural networks, Neurocomputing 30 (2000) 333–337.