A New Adaptive Merging and Growing Algorithm for Designing Artificial Neural Networks

A New Adaptive Merging and Growing Algorithm for Designing Artificial Neural Networks

Md. Monirul Islam, Md. Abdus Sattar, Md. Faijul Amin, Xin Yao, *Fellow, IEEE*, and Kazuyuki Murase

***Abstract*—This paper presents a new algorithm, called adaptive merging and growing algorithm (AMGA), in designing artificial neural networks (ANNs). This algorithm merges and adds hidden neurons during the training process of ANNs. The merge operation introduced in AMGA is a kind of a mixed mode operation, which is equivalent to pruning two neurons and adding one neuron. Unlike most previous studies, AMGA puts emphasis on autonomous functioning in the design process of ANNs. This is the main reason why AMGA uses an adaptive not a predefined fixed strategy in designing ANNs. The adaptive strategy merges or adds hidden neurons based on the learning ability of hidden neurons or the training progress of ANNs. In order to reduce the amount of retraining after modifying ANN architectures, AMGA prunes hidden neurons by merging correlated hidden neurons and adds hidden neurons by splitting existing hidden neurons. The proposed AMGA has been tested on a number of benchmark problems in machine learning and ANNs, including breast cancer, Australian credit card assessment, and diabetes, gene, glass, heart, iris, and thyroid problems. The experimental results show that AMGA can design compact ANN architectures with good generalization ability compared to other algorithms.**

***Index Terms*—Adding neurons, artificial neural network (ANN) design, generalization ability, merging neurons, retraining.**

I. Introduction

ARTIFICIAL neural networks (ANNs) have been used widely in many application areas such as system identification, signal processing, classification, and patternrecognition. Most applications use feedforward ANNs and the backpropagation (BP) learning algorithm [1]. The central issue in using ANNs is to choose their architectures appropriately. A too large architecture may overfit the training data, owing to its excess information processing capability. On the other hand, a too small architecture may underfit the training data, owing to its limited information processing capability. Both overfitting and underfitting cause bad generalizations, an undesirable aspect of using ANNs. It is therefore necessary to design ANNs automatically so that they can solve different problems efficiently.

There have been many attempts in designing ANNs automatically, such as various constructive, pruning, constructive–pruning, and regularization algorithms [2]–[4]. A constructive algorithm adds hidden layers, neurons, and connections to a minimal ANN architecture. A pruning algorithm does the opposite, i.e., it deletes unnecessary hidden layers, neurons, and connections from an oversized ANN. A constructive–pruning algorithm is a hybrid approach that executes a constructive phase first and then a pruning phase. In addition, evolutionary approaches, such as genetic algorithms [5], evolutionary programming [6], [7], and evolution strategies [8], have been used extensively in designing ANNs automatically. The detailed description of designing ANNs using constructive, pruning, constructive–pruning, and evolutionary approaches are presented in Section II.

A regularization algorithm [4], [9], [10] adds a penalty term to the objective function to be minimized. The objective function looks like $E = E_t + \lambda E_c$, where E_t is the training error, E_c is the penalty term, and λ is a positive constant that controls the influence of the penalty term. The difficulty of using such an objective function lies in choosing a suitable coefficient λ, which often requires trial-and-error experiments. The Bayesian approach [11] allows the values of λ to be automatically tuned during training. However, this approach is developed based on simplifying assumptions that do not take into account the multiple minima of an error function [12] (although some techniques are introduced in [13] to moderate this statement). The main problem with constructive, pruning, constructive–pruning, and regularization algorithms is that they use a predefined, fixed, and greedy strategy in designing ANNs. Thus, these algorithms are susceptible to becoming trapped at *architectural local optima* [14].

This paper presents a new algorithm, called adaptive merging and growing algorithm (AMGA), in designing ANNs. This algorithm merges and adds hidden neurons repeatedly or alternatively during ANNs' training. The decision when to merge or add hidden neurons is completely dependent on the improvement of hidden neurons' learning ability or the training progress

Manuscript received March 20, 2008; revised July 3, 2008. This work was supported in part by the Japanese Society for Promotion of Science (JSPS), by the Yazaki Memorial Foundation for Science and Technology, and by the University of Fukui through grants given to K. Murase. The work of Md. M. Islam was supported by the JSPS through a fellowship. This paper was recommended by Associate Editor S. Hu.

Md. M. Islam is with the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka 1000, Bangladesh, and also with the Department of Human and Artificial Intelligence Systems, Graduate School of Engineering, University of Fukui, Fukui 910-8507, Japan (e-mail: monirul@synapse.his.fukui-u.ac.jp).

Md. A. Sattar is with the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka 1000, Bangladesh (e-mail: masattar@cse.buet.ac.bd).

Md. F. Amin is with the Department of Computer Science and Engineering, Khulna University of Engineering and Technology, Khulna 9203, Bangladesh, and also with the Department of Human and Artificial Intelligence Systems, Graduate School of Engineering, University of Fukui, Fukui 910-8507, Japan (e-mail: amin@synapse.his.fukui-u.ac.jp).

X. Yao is with the Centre of Excellence for Research in Computational Intelligence and Applications, University of Birmingham, B15 2TT Birmingham, U.K., and also with the University of Science and Technology of China, Hefei 230026, China (e-mail: x.yao@cs.bham.ac.uk).

K. Murase is with the Department of Human and Artificial Intelligence Systems, Graduate School of Engineering, University of Fukui, Fukui 910-8507, Japan (e-mail: murase@synapse.his.fukui-u.ac.jp).

Digital Object Identifier 10.1109/TSMCB.2008.2008724

1083-4419/$25.00 © 2009 IEEE

of ANNs. It is argued in this paper that such an adaptive strategy is better than a predefined greedy strategy. AMGA's emphasis on using an adaptive and nongreedy strategy can avoid the architectural local optima problem in designing ANNs. It is well known that the nongreedy approach has a lesser chance to be trapped into local optima [14], [15]. The proposed AMGA described in Section III indicates that AMGA does not use any predefined and greedy strategy in designing ANNs.

Our algorithm, AMGA, differs from previous works in designing ANNs on a number of aspects. First, it emphasizes on adaptive functioning of the design process by not guiding the process in a predefined, fixed, and same way for all problems. This strategy is quite different from the one used in constructive, pruning, constructive–pruning, and regularization algorithms [4]–[9], [16]–[30]. Unlike AMGA, these algorithms guide the ANN design process in a predefined, fixed, and same way for all problems, although different problems may have different characteristics. Even for the same problem, different strategies may be needed at different stages of the design process. This is the main reason for the better performance of BP [1] with different strategies (i.e., learning rates) than with a fixed strategy [31]. Since a number of user-specified parameters, some for a design algorithm and some for a learning algorithm, is necessary in designing ANNs, it is reasonable to think that ANN design algorithms with adaptive strategies may yield a good performance.

Second, AMGA gives importance on reducing retraining epochs after modifying ANN architectures by merge and add operations. It is well known that, when a design algorithm adds or prunes any parameter to or from existing ANN architectures, it retrains the modified architectures to adapt their connection weights. Since the design algorithm may modify ANN architectures several times during their entire training process, this may result to overtraining, which has a detrimental effect on the generalization performance of ANNs [32]. The issue of overtraining is ignored in most existing works. Thus, no technique is developed to reduce retraining epochs in designing ANNs.

The proposed AMGA acknowledges the importance of reducing the retraining epochs in designing ANNs. To achieve this goal, this algorithm merges two correlated hidden neurons in such a way that the effect of the merged neuron to an ANN is nearly same as the combined effect of its two parent neurons. This can be considered as compensating the effect of one pruned neuron by its correlated counterpart. In the neuron addition process, AMGA adds a hidden neuron not with random connection weights but with mutating the weights of an existing hidden neuron. The essence of using such pruning and addition techniques is that a small number of epochs would be required for retraining after the modification of the ANN architecture. Our empirical studies presented in Section IV-C confirm the aforementioned intuition.

Third, AMGA encourages compactness in designing ANNs by several ways. For example, although AMGA can add one neuron at each step, it can prune several neurons by combining several correlated pairs of hidden neurons. The encouragement of decorrelation by merging correlated hidden neurons is also beneficial for producing compact ANN architectures. When hidden neurons are less correlated, ANNs with a small number of hidden neurons can process more information because decorrelated hidden neurons do not learn redundant information. Moreover, AMGA also tries to execute the merge operation before the add operation. The way and sequence used by AMGA for merge and add operations reflect clearly the algorithm's preference in designing compact ANNs. These kinds of techniques are rarely used in existing algorithms for designing ANNs. Although constructive–pruning algorithms execute a pruning phase in association with a constructive phase to produce compact ANNs, no emphasis is given in pruning. Hidden neurons are generally pruned by constructive–pruning or pruning algorithms without considering their correlation.

The rest of this paper is organized as follows. Section II discusses different approaches to design ANN architectures and their potential problems. Section III describes AMGA in detail and gives motivations behind various design choices and ideas. Section IV presents experimental results on AMGA and some discussions. Finally, Section V concludes with a summary of this paper and few remarks.

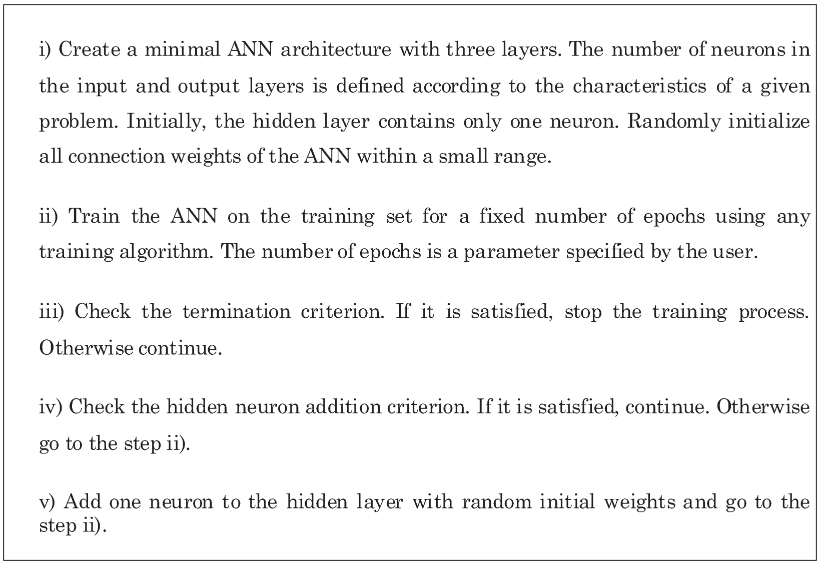
II. Previous Work

Since the performance of any ANN is greatly dependent on its architecture, a myriad of algorithms have been proposed for designing ANNs. Most algorithms are either constructive or pruning in nature, and they determine the number of hidden neurons in three-layered ANNs. It has been known that three-layered ANNs can solve any linear or nonlinear problems [33]–[37]. A review for constructive algorithms can be found in [2], while that for pruning algorithms in [3]. There are also few algorithms that combine constructive and pruning approaches in one algorithm [27]–[30]. In addition, evolutionary approaches have been used in designing ANNs. There has been a great interest in combining learning and evolution with ANNs. A large number of ANN design algorithms have been proposed based on evolutionary approaches (see the review papers [38]–[40]). This section delineates the features of these four classes of algorithms so that the reasons behind different techniques and ideas used in AMGA can easily be understood.

A. Constructive Algorithm

A constructive algorithm starts with a minimal ANN architecture, for example, a single-hidden-layered ANN with one neuron in the hidden layer. Dynamic neuron creation [41] is probably the first-ever constructive algorithm in designing ANNs. A large number of constructive algorithms following the dynamic-creation algorithm were developed (e.g., [16]–[18], [42]–[46]). Hidden neurons with sigmoid or complicated transfer functions are generally added in constructive algorithms. The whole network or only the newly added hidden neuron is trained after each addition step. Fig. 1 shows how a constructive algorithm works to find a near optimal ANN architecture for a given problem. This algorithm can add either sigmoid or other neurons and can train either the whole network or only the newly added hidden neuron.

There are several advantages of using constructive algorithms. It is relatively easy for an inexperienced user to specify



i) Create a minimal ANN architecture with three layers. The number of neurons in the input and output layers is defined according to the characteristics of a given problem. Initially, the hidden layer contains only one neuron. Randomly initialize all connection weights of the ANN within a small range.

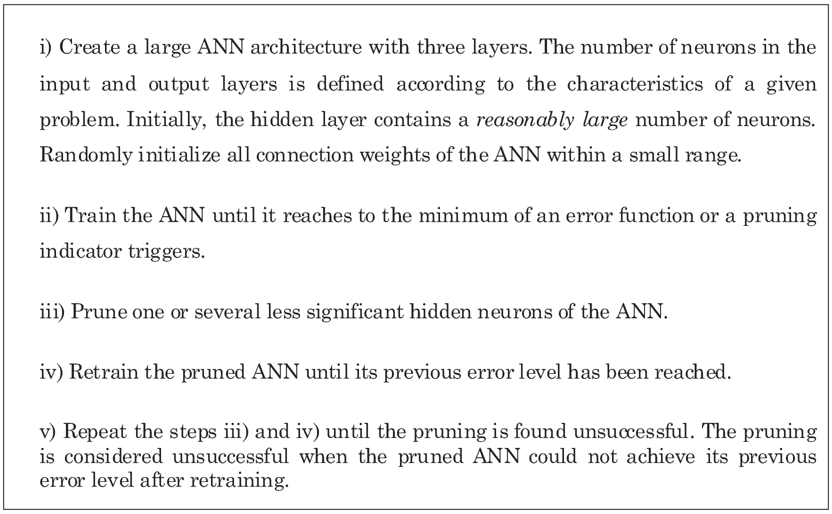
ii) Train the ANN on the training set for a fixed number of epochs using any training algorithm. The number of epochs is a parameter specified by the user.

iii) Check the termination criterion. If it is satisfied, stop the training process. Otherwise continue.

iv) Check the hidden neuron addition criterion. If it is satisfied, continue. Otherwise go to the step ii).

v) Add one neuron to the hidden layer with random initial weights and go to the step ii).

Fig. 1. Typical constructive algorithm in designing ANNs.



i) Create a large ANN architecture with three layers. The number of neurons in the input and output layers is defined according to the characteristics of a given problem. Initially, the hidden layer contains a *reasonably large* number of neurons. Randomly initialize all connection weights of the ANN within a small range.

ii) Train the ANN until it reaches to the minimum of an error function or a pruning indicator triggers.

iii) Prune one or several less significant hidden neurons of the ANN.

iv) Retrain the pruned ANN until its previous error level has been reached.

v) Repeat the steps iii) and iv) until the pruning is found unsuccessful. The pruning is considered unsuccessful when the pruned ANN could not achieve its previous error level after retraining.

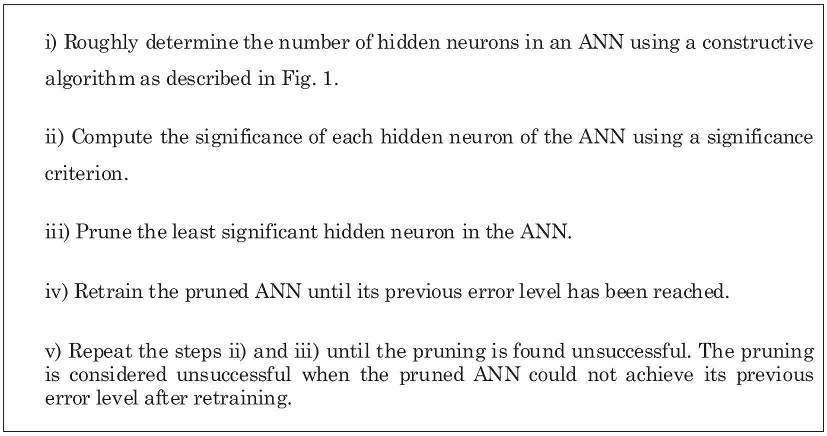
Fig. 2. Typical pruning algorithm in designing ANNs.

the initial conditions in constructive algorithms. For example, the number of hidden layers and neurons can simply be set to one. Constructive algorithms are computationally efficient because they always search small solutions first. Furthermore, they are very easy to implement. There are also hurdles that constructive algorithms need to overcome [2]. For example, one has to determine when to add hidden neurons and when to stop the neuron addition process.

B. Pruning Algorithm

Unlike constructive algorithms, a pruning algorithm starts with an oversized ANN architecture, for example, a single-hidden-layered ANN with a large number of hidden neurons. Most pruning algorithms start the removal of irrelevant parameters (connections and/or hidden neurons) after training an oversized ANN to a minimum error (see the review paper [3]). They generally prune one neuron in each pruning step. There are only few recent works that prune several neurons in each pruning step during training (e.g., [12] and [47]). The decision to prune hidden neurons is taken based on the neuron's significance or relevance. A number of pruning criteria have been proposed to decide which hidden neurons are to be pruned from a trained ANN [see the review paper [3] and some recent works (e.g., [12] and [26])]. When an ANN is trained, each of its hidden neurons learns some information during training, although they may be insignificant for some neurons. The ANN is usually retrained after pruning so that its existing hidden neurons can adjust their weights to compensate the roles played by pruned hidden neurons (e.g., [25], [26], and [29]).

Fig. 2 shows how a pruning algorithm deletes hidden neurons from an oversized ANN either during training (e.g., [12] and [47]) or after converging to a minimum error (e.g., [24]–[26]). There are several advantages of a pruning algorithm. This algorithm can learn the target function very quickly and has a lesser chance to trap into local minima [3], [26], [48]. These two facilities are obtained due to extra parameters (e.g., hidden neurons) in the initial ANN architecture of the pruning algorithm.



i) Roughly determine the number of hidden neurons in an ANN using a constructive algorithm as described in Fig. 1.

ii) Compute the significance of each hidden neuron of the ANN using a significance criterion.

iii) Prune the least significant hidden neuron in the ANN.

iv) Retrain the pruned ANN until its previous error level has been reached.

v) Repeat the steps ii) and iii) until the pruning is found unsuccessful. The pruning is considered unsuccessful when the pruned ANN could not achieve its previous error level after retraining.

Fig. 3. Typical constructive–pruning algorithm in designing ANNs.

However, one does not know in practice how big the initial architecture should be for a given problem [2]. Furthermore, it should be kept in mind that the pruning algorithm is not *completely free* from the local optima problem because it uses a greedy strategy in determining ANN architectures [14].

C. Constructive–Pruning Algorithm

As seen from Fig. 1, a constructive algorithm adds hidden neurons to a minimal ANN architecture one by one during training. The number of hidden neurons may become ridiculously large in some cases if the addition process is not stopped properly. Some algorithms (e.g., [27]–[30]) try to solve the aforementioned problem by employing a pruning phase in conjunction with a constructive phase. These algorithms first determine the number of hidden neurons and/or connections in an ANN roughly using a constructive approach. A pruning approach is then applied for refining the selection, i.e., removing the irrelevant hidden neurons and/or connections.

Fig. 3 shows how a constructive–pruning algorithm adds and prunes hidden neurons in order to find a near optimal ANN architecture for a given problem. The advantage of the constructive–pruning algorithm is that one can add more than one hidden neuron in each addition step. This may help to speed up the training process. Furthermore, this algorithm may produce compact architectures. These two facilities are obtained due to the inclusion of the pruning phase in the constructive algorithm. However, one needs to determine when to stop the pruning process.

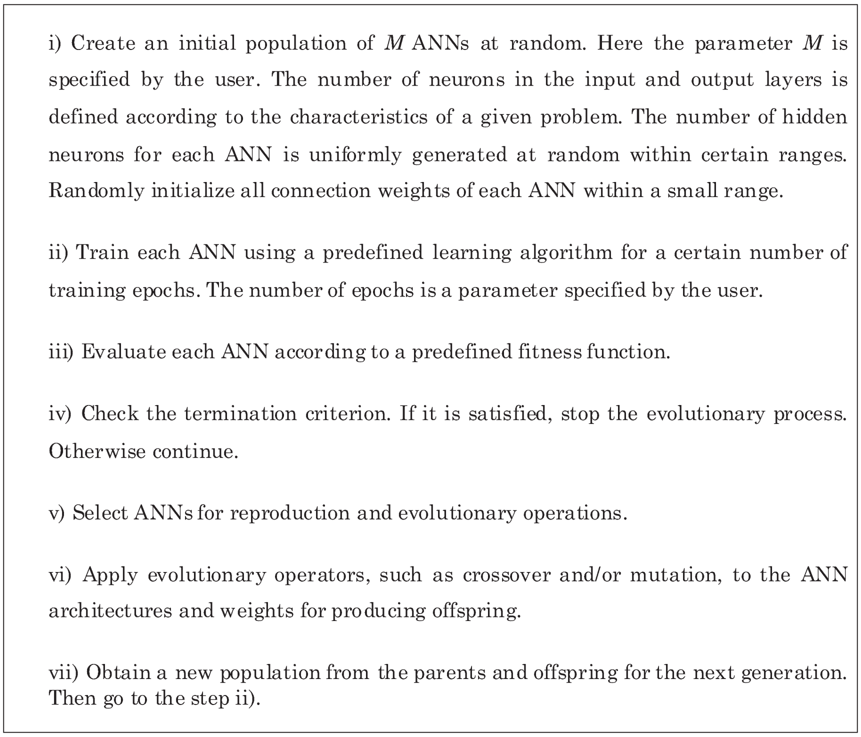
D. Evolutionary Algorithm

Evolutionary algorithms [5]–[8] have been used widely to adapt various parameters of ANNs, such as connection weights, architectures, and learning rules. These stochastic search algorithms are developed from ideas and principles of natural evolution. They apply a number of operators borrowed from natural genetics, like recombination, crossover, and mutation, on existing solutions to produce better solutions. Evolutionary algorithms differ from constructive, pruning, and constructive–pruning algorithms in that they involve a search from a *population* of solutions, not from a single solution.

The evolution of connection weights introduces an adaptive and global approach to training ANNs (e.g., [49]–[51]). The evolution of architectures enables ANNs to adapt their architectures and thus provides an automatic approach in designing ANNs (e.g., [52]–[54]). The evolution of learning rules can be regarded as a process of *learning to learn* in ANNs where the adaptation of learning rules is achieved through evolution (e.g., [55]–[57]). In addition, some algorithms evolve architectures and connection weights simultaneously (e.g., [58]–[60]). We have already seen that constructive, pruning, and constructive–pruning algorithms are used in conjunction with a learning algorithm in which the former approach is used to adapt network architectures while the later one for their connection-weight adaptation. Similarly, an evolutionary approach can be used in conjunction with a learning algorithm [61], [62].

A typical evolutionary approach that combines the architectural evolution with the weight learning is shown in Fig. 4. The main advantage of this approach is that it can avoid the architectural local optima problem [14], [40]. However, the evolutionary approach is quite demanding in both time and user-defined parameters [2]. Moreover, it is necessary to find a set of optimal control parameters so that an evolutionary process can balance exploration and exploitation in finding good quality solutions. For example, if crossover and mutation rates are chosen very high, much of the search space will be explored, but there is a high probability of losing good solutions and failing to exploit existing solutions.

In addition to the problem mentioned for the constructive, pruning, and constructive–pruning algorithms, these algorithms use a kind of predefined and greedy search strategy in designing near optimal ANN architectures. Such a predefined and greedy strategy may be suitable for some simple problems. However, this strategy may not be suitable for other problems as its search process may be trapped into architectural local optima, an inherent problem of any greedy approach [14], [15]. It has been shown for linear networks with Ψ inputs and Ψ outputs that up to Ψ local minima are possible; for multilayer ANNs, the situation is even worse [63]. Although evolutionary algorithms can avoid the local optima problem, this avoidance ability is dependent on the autonomous functioning of an evolutionary process [64].



i) Create an initial population of M ANNs at random. Here the parameter M is specified by the user. The number of neurons in the input and output layers is defined according to the characteristics of a given problem. The number of hidden neurons for each ANN is uniformly generated at random within certain ranges. Randomly initialize all connection weights of each ANN within a small range.

ii) Train each ANN using a predefined learning algorithm for a certain number of training epochs. The number of epochs is a parameter specified by the user.

iii) Evaluate each ANN according to a predefined fitness function.

iv) Check the termination criterion. If it is satisfied, stop the evolutionary process. Otherwise continue.

v) Select ANNs for reproduction and evolutionary operations.

vi) Apply evolutionary operators, such as crossover and/or mutation, to the ANN architectures and weights for producing offspring.

vii) Obtain a new population from the parents and offspring for the next generation. Then go to the step ii).

Fig. 4. Typical evolutionary algorithm in designing ANNs.

III. AMGA

In order to avoid the architectural local optima problem, AMGA uses an adaptive search strategy in designing ANNs. This strategy merges and adds hidden neurons based on the learning ability of hidden neurons and the training progress of ANNs, respectively. The algorithm AMGA is used here to determine the number of hidden neurons for three-layered feedforward ANNs with a sigmoid transfer function. This is, however, not an inherent constraint. In fact, AMGA has little constraints on the type of ANN architectures and transfer functions used in its design process. Cascade [65] or other types of ANN architectures, and hyperbolic tangent [66], Hermite polynomial [67], or any other type of transfer function can be used in AMGA.

The steps of AMGA can be described by the flowchart shown in Fig. 5, which are explained further as follows.

Step 1) Create an initial ANN architecture consisting of three layers. The number of neurons in the input and output layers is the same as the number of inputs and outputs of a given problem, respectively. The number of neurons M in the hidden layer is generated at random. Initialize all connection weights of the ANN uniformly at random within a small range.

Step 2) Initialize an epoch counter $\mu_i = 0,\ i = 1, 2, \ldots, M$, for each hidden neuron h_i of the ANN. This counter is used to count the number of epochs for which a hidden neuron is trained so far.

Step 3) Partially train the ANN on the training set for a fixed number of epochs using BP [1]. The number of epochs τ is specified by the user.

Step 4) Increment the epoch counter as follows: for $i = 1, 2, \ldots, N$

$$\mu_i = \mu_i + \tau \tag{1}$$

where N is the number of hidden neurons in the existing ANN architecture. Initially, N and M are the same.

Step 5) Compute the error of the ANN on the validation set. If the termination criterion is satisfied, stop the training process, and the current network architecture is the final ANN. Otherwise, continue. According to Prechelt [68], the error E is computed as follows:

$$E = 100 \frac{o_{\max} - o_{\min}}{KV} \sum_{v=1}^{V} \sum_{i=1}^{K} \left(Y_i(v) - Z_i(v)\right)^2 \tag{2}$$

where $o_{\max}$ and $o_{\min}$ are the maximum and minimum values of the output coefficients in a problem representation, respectively, V is the number of examples in the validation set, and K is the number of output neurons. $Y_i(v)$ and $Z_i(v)$ are the actual and desired outputs, respectively, of the ith output neuron for a validation example v. The aforementioned error equation is normalized by V and K to make the error measure less dependent on the size of the validation set and the number of output neurons [68]. According to Prechelt [68], this error measure is called squared error percentage.

Step 6) Remove the label of hidden neurons, if there exists, and compute the significance η_i of each hidden

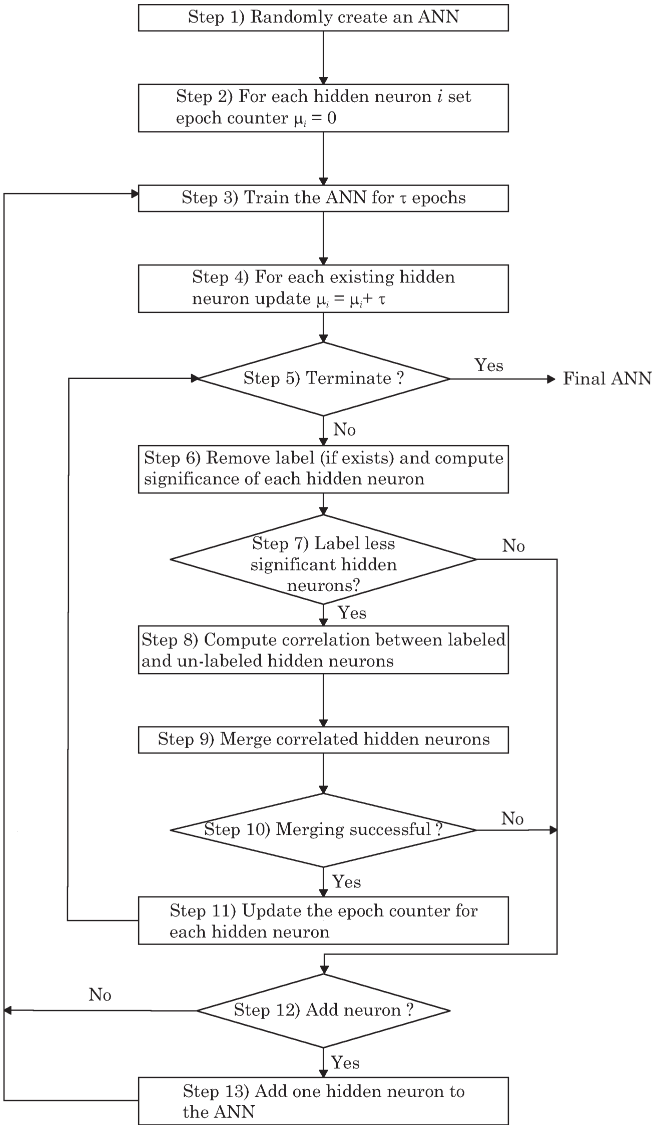

Step 1) Randomly create an ANN
Step 2) For each hidden neuron i set epoch counter $\mu_i = 0$
Step 3) Train the ANN for τ epochs
Step 4) For each existing hidden neuron update $\mu_i = \mu_i + \tau$
Step 5) Terminate ?
Yes
Final ANN
No
Step 6) Remove label (if exists) and compute significance of each hidden neuron
Step 7) Label less significant hidden neurons?
No
Yes
Step 8) Compute correlation between labeled and un-labeled hidden neurons
Step 9) Merge correlated hidden neurons
Step 10) Merging successful ?
No
Yes
Step 11) Update the epoch counter for each hidden neuron
No
Step 12) Add neuron ?
Yes
Step 13) Add one hidden neuron to the ANN

Fig. 5. Flowchart of AMGA.

neuron h_i using an empirical formula as follows [47]: for $i = 1, 2, \ldots, N$

$$\eta_i = \frac{\sigma_i}{\sqrt[3]{\mu_i}} \tag{3}$$

where σ_i is the standard deviation, which is computed based on the outputs of h_i for the examples in the training set.

Step 7) If the significance of one or more hidden neurons is less than a predefined threshold, label those neurons with S and continue. Otherwise, go to Step 12). The significance threshold η_{th} is a parameter specified by the user. It is important here to note that AMGA puts a label at most $N/2$ hidden neurons of the ANN.

Step 8) Compute the correlation between each S-labeled hidden neuron and other unlabeled hidden neurons in the ANN. Like σ, AMGA computes the correlation between two hidden neurons based on their outputs over the examples in the training set.

Step 9) Merge each S-labeled hidden neuron with its most correlated unlabeled counterpart. It is assumed here that the S-labeled hidden neuron is not only less significant but also redundant. Thus, AMGA produces one new hidden neuron by merging the S-labeled hidden neuron with its unlabeled counterpart. This new neuron does not contain any label S, and AMGA initializes a new epoch counter with zero for it.

Step 10) Retrain the modified ANN, which is obtained after merging hidden neurons, until its previous error level has been reached. If the modified ANN is able to reach its previous error level, continue. Otherwise, restore the unmodified ANN and go to Step 12).

Step 11) Update the epoch counter for each hidden neuron in the modified ANN and go to Step 5). The epoch counter is updated as follows:

$$\mu_i = \mu_i + \tau_r, \qquad i = 1, 2, \ldots, N \tag{4}$$

where τ_r is the number of epochs for which the modified ANN is retrained after the merge operation.

Step 12) Check the neuron addition criterion that monitors the progress of the training error reduction. If this criterion is satisfied, continue. Otherwise, go to Step 3) for further training. It is assumed here that, since the merge operation is found unsuccessful (or cannot be applied) and the neuron addition criterion is not satisfied, the performance of the ANN can only be improved by training.

Step 13) Add one neuron to the current ANN architecture and go to Step 3). Since the error of the ANN does not reduce significantly after training and the merge operation is found unsuccessful (or cannot be applied), the performance of the ANN can only be improved by adding hidden neurons. In this work, AMGA adds a hidden neuron by splitting an existing hidden neuron of the ANN. The splitting operation produces two new hidden neurons from an existing hidden neuron h_i. The algorithm AMGA initializes two epoch counters for the new neurons. The epoch counters are initialized by $\mu_i/2$, where μ_i is the number of epochs for which the existing hidden neuron h_i is trained so far.

It is now clear that AMGA can merge or add hidden neurons, depending on the hidden neurons' significance or the progress of the training error reduction. In other words, AMGA can execute merge and add operations repeatedly, alternatively, or in any other sequence. This execution behavior indicates that AMGA does not guide the ANN design process in a predefined and fixed way. Although the design scheme used in AMGA seems to be a bit complex, its essence is the use of an adaptive search strategy with four components: neuron merging, neuron addition by splitting, architecture adaptation, and termination criterion based on validation error. Details about each component of AMGA are given in the following sections.

A. Neuron Merging

Unlike conventional pruning algorithms, AMGA uses a different pruning scheme, called merging. This scheme is equivalent to pruning two neurons and adding one neuron. The merge operation uses a significance criterion and correlation information in merging hidden neurons. If the significance of a hidden neuron is found less than a predefined threshold level η_{th}, AMGA merges this less significant hidden neuron with another more significant and correlated hidden neuron. The algorithm AMGA computes the significance of hidden neurons after every τ training epochs or after every successful merge operation. It merges one or more hidden neurons if their significance is less than the threshold level. The merge operation in AMGA is accomplished in two steps.

In the first step, AMGA computes the significance of each hidden neuron in an ANN using (3). The significance is computed based on the variation of the hidden neuron's output over all examples in the training set. The standard deviation is used here to determine how much the neuron's output deviates for different examples in the training set. Since the output of the hidden neuron is connected only to the neurons in the output layer of a three-layered feedforward ANN, the variation in the hidden neuron's output may produce a variation in the ANN's output. This is the main reason for using the variation of the hidden neuron's output for computing its significance, which can be computed with a small cost. The hidden neuron is considered less significant if its output does not vary much for different inputs.

The inclusion of μ in (3) is necessary to measure the significance of a hidden neuron, although it is not used in existing pruning algorithms. This is because AMGA not only merges but also adds hidden neurons at different stages in the training process of an ANN. This means that AMGA may train different hidden neurons for a different number of training epochs. The number of training epochs is very important to make hidden neurons significant. For example, if a hidden neuron is trained for a small number of epochs, it may be unable to distinguish different training examples because of insufficient training. The hidden neuron, therefore, may respond in a similar way for different training examples, resulting in a small standard deviation.

The significance η_i, which AMGA computes using (3), of the hidden neuron h_i is small when its standard deviation σ_i and/or its number of training epochs μ_i is large. The smaller the value of η_i is, the less significant the h_i is. A less significant hidden neuron delivers almost constant information (because of the small σ) to the neurons of the output layer. In other words, the characteristics of such a hidden neuron become known. Thus, the hidden neuron's effects to an ANN can easily be compensated by modifying the connection weights of its correlated counterpart. The proposed AMGA detects the less significant hidden neurons in the ANN and labels them with S if their significance is found less than a predefined threshold η_{th}. Here, the parameter η_{th} is specified by the user.

In the second step, AMGA computes the correlation between each S-labeled hidden neuron and other hidden neurons in the ANN. Correlation is one of the most common and useful statistics that describes the degree of relationship between two variables. A number of criteria have been proposed in statistics to estimate correlation. In this paper, AMGA uses the best known *Pearson product-moment correlation coefficient* to measure correlation between different hidden neurons in the ANN. The correlation coefficient C_{ij} between the S-labeled hidden neuron i and the unlabeled hidden neuron j is

$$C_{ij} = \frac{\sum_{p=1}^{P} \left(h_i(p) - \bar{h_i}\right)\left(h_j(p) - \bar{h_j}\right)}{\sigma_i \sigma_j} \tag{5}$$

where $h_i(p)$ and $h_j(p)$ are the outputs of hidden neurons i and j, respectively, for the example p in the training set. The variables $\bar{h_i}$ and $\bar{h_j}$ represent the mean values of h_i and h_j, respectively, averaged over all training examples. The standard deviations σ_i and σ_j of the hidden neurons h_i and h_j are also computed over all training examples.

The algorithm AMGA merges each S-labeled hidden neuron with its most correlated unlabeled counterpart. The most correlated hidden neuron pair can be found by ordering the correlation coefficient in descending order. It is important here to note that, if an unlabeled hidden neuron maintains the highest correlation with more than one S-labeled hidden neuron, the unlabeled hidden neuron is merged with only one S-labeled hidden neuron. The other S-labeled hidden neuron is merged with its another correlated counterpart. In this scenario, the maximum number of hidden neurons that can be pruned by merging is $N/2$ for an ANN consisting of N hidden neurons. In order to reduce retraining epochs after merging, AMGA merges two correlated neurons in such a way that the effect of the merged neuron to the ANN is nearly the same as the combined effect of two correlated neurons. Consider that AMGA produces a neuron h_m by merging two neurons h_a and h_b. The algorithm assigns the input and output connection weights of the h_m in the following way:

$$w_{mi} = \frac{w_{ai} + w_{bi}}{2}, \qquad i = 1, 2, \ldots, p \tag{6}$$

$$w_{jm} = w_{ja} + w_{jb}, \qquad j = 1, 2, \ldots, q \tag{7}$$

where p and q are the number of neurons in the input and output layers of the ANN, respectively. The weights w_{ai} and w_{bi} are the ith input connection weights of h_a and h_b, respectively, while w_{ja} and w_{jb} are their jth output connection weights. The weights w_{mi} and w_{jm} are the ith input and jth output connection weights of h_m, respectively. Since h_a and h_b are highly correlated, it can be easily shown that h_m delivers almost the same amount of information to the output layer as it is delivered together by h_a and h_b. The pseudocode of the merge operation is shown in Fig. 6.

B. Neuron Addition

The proposed AMGA uses a simple criterion to add a hidden neuron in an ANN. This criterion is based on the training progress of the ANN. When the training error of the ANN does not reduce by an amount ε after training epochs τ, AMGA assumes that the information processing capability of the ANN

```
Input:
  Set of hidden neurons $H$
  Standard deviation of hidden neuron $\sigma_i$, $1 \le i \le |H|$
  Epoch counter of hidden neuron $\mu_i$, $1 \le i \le |H|$
  Threshold significance $\eta_{th}$
Output:
  Set of hidden Neurons $H'$ such that $|H|/2 \le |H'| \le |H|$

Define $S$ : A set of insignificant neurons
$S = \phi$
$H' = \phi$
for each $h_i \in H$
    compute significance $\eta_i$      [according to Eq. (3)]
    if $\eta_i < \eta_{th}$ then
        $S = S \cup h_i$
    end
end

Sort $S$ by the increasing order of hidden neuron's significance
if $|S| > |H|/2$ then
    $S$ = first $|H|/2$ hidden neurons of $S$
end

$S' = H - S$
for each $h_a \in S$
    for each $h_b \in S'$
        $c_{ab} = \mathbf{Corr}(h_a, h_b)$       [ according to Eq. (5) ]
    end
    $h_k = \arg\max_{h_b}(c_{ab})$
    Merge $h_a$ and $h_k$ to $h_m$    [according to Eqs. (6) and (7)]
    $S' = S' - \{h_k\}$
    $H' = H' \cup \{h_m\}$
end

$H' = H' \cup S'$
```

Fig. 6. Pseudocode of merge operation used in AMGA.

is insufficient. It is therefore necessary to add hidden neurons in the ANN. Here, ε and τ are two user-specified parameters. The neuron addition criterion can be expressed as

$$E(t) - E(t+\tau) <= \epsilon, \qquad t = \tau, 2\tau, 3\tau, \ldots \tag{8}$$

where $E(t)$ and $E(t+\tau)$ are the training errors at epochs t and $t+\tau$, respectively. This simple addition criterion is used widely in many constructive algorithms (e.g., [18], [23], [27], [29], and [69]). Our algorithm AMGA tests this neuron addition criterion if the merging criterion is not satisfied or the execution of the merge operation is found unsuccessful. If the neuron addition criterion is satisfied, AMGA adds one neuron to the existing network architecture.

Unlike most constructive algorithms, AMGA adds a hidden neuron by splitting an existing hidden neuron of an ANN. This addition scheme can be considered as pruning one neuron and adding two neurons. The process of a neuron splitting is called "cell division" [70]. In addition to the reasons given by Odri *et al.* [70], growing an ANN by splitting an existing hidden neuron can preserve the behavioral link between the parent ANN and the newly grown ANN better than growing the ANN by adding a hidden neuron with random connection weights. It is expected that the grown ANN will be able to learn the target function more quickly, because it has more processing power (hidden neurons) than its parent network.

In AMGA, a neuron that is going to be split is chosen uniformly at random among all hidden neurons in an ANN. Two new neurons created by splitting an existing neuron have the same number of connections as the parent neuron. The weights of the new neurons are calculated as [70]

$$w^1 = (1+\alpha)w \tag{9}$$

$$w^2 = -\alpha w \tag{10}$$

where w is the weight vector of the parent (existing) neuron, w^1 is the weight vector of the first daughter (new) neuron, and w^2 is the weight vector of the second daughter (new) neuron. The mutation parameter α may take either a fixed or random value according to a certain distribution rule. However, in any case, the value of α needs to be chosen small for avoiding a large change in the existing network functionality [70]. In this paper, AMGA uses the Gaussian distribution with a mean of zero and a variance of one in selecting the value of α.

C. Architecture Adaptation

As mentioned before, AMGA is used here for designing three-layered feedforward ANNs. The algorithm assigns the number of neurons in the input and output layers of an ANN according to the characteristics of a given problem. These numbers are not modified during the adaptation process. Initially, AMGA assigns a random number of neurons in the hidden layer of the ANN. The aim of the architecture adaptation process is to find a near optimal number of the hidden neurons and connection weights of the ANN.

The algorithm AMGA uses neuron merging and addition operations to find a near optimal number of hidden neurons, while it uses the BP learning algorithm [1] to find near optimal connection weights. The merge operation reduces the network size by pruning correlated hidden neurons. This operation thereby helps to undermine the overfitting effect. The add operation increases the information processing capacity by adding hidden neurons to an existing network architecture. This operation thereby helps to reduce the underfitting effect.

The adaptation process of AMGA starts by training an initial ANN architecture. In AMGA, the merge or add operation is applied when the criterion of the operation is satisfied during training. This indicates that AMGA adapts both the numbers of hidden neurons and connection weights simultaneously. The merge operation is applied when hidden neurons in the ANN are found less significant after training them. The add operation is applied when the training and the merge operation fail to improve the performance of the ANN. Since two different criteria are used for merge and add operations, AMGA can apply these operations in any sequence, depending on when the criteria are satisfied. By applying the merge and add operations again and again during training, AMGA is expected to find a near optimal number of hidden neurons and connection weights of the ANN for solving a given problem.

D. Termination Criterion

The training error of an ANN may reduce as its training process progresses. However, at some point, usually in the later stages of training, the ANN may start to take advantage of idiosyncrasies in the training data. Consequently, its generalization performance may start to deteriorate even though the training error continues to decrease. Chauvin [32] describes an example of this type of overfitting caused by overtraining. In [71], a number of plausible criteria are proposed for terminating the training process of the ANN automatically.

One common approach to avoid overfitting is to estimate the validation error during training and stop the training process using a criterion based on the validation error. The simplest method to achieve this goal is to divide the training data into training and validation sets. The training set can be used to modify the weights of the ANN, while the validation set can be used to terminate the training process of the ANN.

The proposed AMGA uses a very simple criterion that terminates the training process of the ANN when its validation error increases for T successive times measured at the end of each of T successive strips [71]. In each strip, AMGA adds one hidden neuron and retrains the modified ANN architecture, or prunes several hidden neurons and retrains the modified ANN architecture or trains the existing ANN architecture. The idea behind the termination criterion is to stop the training process of the ANN when its validation error increases not just once but during T consecutive times independent of how large the increases actually are. It can be assumed that such increases indicate the beginning of the final overfitting not just the intermittent. Thus, AMGA stops the training process of the ANN when its validation error increases for T consecutive times. This criterion can be expressed by the following way:

$$E(i) < E(i+j), \qquad j = 1, 2, \ldots, T \tag{11}$$

where $E(i)$ and $E(i+j)$ are the errors of the ANN at strips i and $i+j$, respectively, and T is a parameter specified by the user. The proposed algorithm computes these errors on the validation set using (2) and tests the termination criterion after completion of every strip.

IV. Experimental Studies

This section presents AMGA's performance on several well-known benchmark classification problems, including breast cancer, Australian credit card assessment, and diabetes, gene, glass, heart disease, iris, and thyroid problems. These problems have been the subject of many studies in ANNs and machine learning. The characteristics of these problems are summarized in Table I, which show a considerable diversity in the number of examples, attributes, and classes. The detailed description of these problems can be obtained from the University of California Irvine Machine Learning Repository and [68].

A. Experimental Methodology

In this work, the data sets of different problems were partitioned into three sets: a training set, validation set, and testing set. The number of examples in these sets are shown in the Table I. The training set was used to train and modify ANN architectures. The validation set was used for stopping the training process of ANNs, while the testing set for measuring their generalization ability. In all data sets, the first P_1 examples were used for the training set, the following P_2 examples for the validation set, and the final P_3 examples for the testing set. These partitions were used according to the suggestion provided in benchmarking methodologies [72], [73].

TABLE I
CHARACTERISTICS OF EIGHT BENCHMARK CLASSIFICATION PROBLEMS

Problem	Number of				
	input attributes	output classes	training examples	validation examples	testing examples
Cancer	9	2	350	175	174
Card	51	2	345	173	172
Diabetes	8	2	384	192	192
Gene	120	3	1588	794	793
Glass	9	6	107	54	53
Heart	35	2	460	230	230
Iris	4	3	75	38	37
Thyroid	21	3	3600	1800	1800

TABLE II
PERFORMANCE OF AMGA ON EIGHT BENCHMARK CLASSIFICATION PROBLEMS. ALL RESULTS WERE AVERAGED OVER 50 INDEPENDENT RUNS

Problem	Number of		TER
	hidden neurons	epochs	
Cancer	1.80	234.5	1.30
Card	1.66	131.8	12.67
Diabetes	4.14	390.7	21.97
Gene	2.14	440.3	12.05
Glass	5.02	499.5	30.56
Heart	2.56	130.6	18.87
Iris	2.62	165.3	1.89
Thyroid	5.60	630.1	2.44

For all experiments, one bias neuron with a fixed input $+1$ was connected to the neurons of the hidden and output layers. The logistic sigmoid function was used for the neurons in the hidden and output layers. The learning rate and momentum term for BP [1] were chosen as 0.10 and 0.9, respectively. The weights of ANNs were initialized to random values in the range between -0.5 and $+0.5$. The number of training epochs for partial training, i.e., τ, was set to 20. The values for ε and η were set to 1E-03 and 0.06, respectively. The value of T used in the termination criterion was set to 3. We have chosen these parameters after some preliminary runs. They were not meant to be optimal.

B. Experimental Results

Table II shows the performance of AMGA over 50 independent runs. The testing error rate (TER) refers to the percentage of wrong classifications produced by ANNs on the testing set. The number of epochs refers to the total number of training cycles required in designing final from initial ANN architectures.

It is evident that AMGA was able to design ANNs with a good generalization ability, i.e., small TER. For example, for the card problem, the average TER of ANNs designed by AMGA was 12.67. It was only 1.89 for the easy iris problem. A moderate TER was achieved for the diabetes and glass problems, which were recognized as the difficult problems in machine learning and ANNs [68].

The ability of AMGA in designing ANNs for different problems using a different number of epochs and hidden neurons is clear from the results presented in Table II. The proposed AMGA spent the highest number of epochs to design ANNs for the thyroid problem, which was the largest compared to the other problems we tested in this work (Table I). In terms of average results, AMGA spent 630.1 epochs in designing ANNs with 5.60 hidden neurons for the thyroid problem, while it spent only 131.8 epochs in designing ANNs with 1.66 hidden neurons for the card problem. The number of training examples for the thyroid problem was 3600, while it was 345 for the card problem. It is natural to require more computational resources, i.e., large architectures and training epochs, to process a large amount of information. This may be the main reason that AMGA used more computational resources for the thyroid problem.

The close observation of Tables I and II reveals that the size of the training set is not only the factor in determining the computational resources needed for solving different problems but also the characteristics of the problems. It is interesting that AMGA produced the second largest ANN architecture (next to thyroid) for one of the smallest problems, the glass problem. The difficulty of the glass problem lies with its small training set with respect to the number of output classes. For example, the number of training examples and output classes of the glass problem were 107 and 6, respectively, while they were 350 and 2, respectively, for the cancer problem (Table I). In terms of average results, AMGA spent 234.5 epochs in designing ANNs with 1.80 hidden neurons for the cancer problem, while it spent 499.5 epochs in designing ANNs with 5.02 hidden neurons for the glass problem. It is natural to require more hidden neurons and training epochs for learning and partitioning more classes. This may be the main reason that AMGA spent more computational resources for the glass problem. The complexity of the glass and cancer problems can be understood from their TERs. The average TER of ANNs designed by AMGA for the glass problem was 30.56, while it was 1.30 for the cancer problem. The aforementioned examples illustrate AMGA's ability in designing ANNs based on the complexity of problems.

To observe how AMGA designs ANNs, Fig. 7 shows, as the number of training epochs increases, the training error and the number of hidden neurons in ANNs for three different problems. This figure represents the results of a single run for each problem. Several observations can be made from this figure.

First, AMGA could dynamically prune or add hidden neurons at different stages of the training process. This means that AMGA applied the merge or add operation when the criteria for these operations were satisfied during training. It is noticeable that AMGA tried to solve problems with compact ANN architectures. When compact ANN architectures failed to solve problems, AMGA added neurons at the very end of the training process. This scenario can be observed in designing ANNs for the glass and thyroid problems (Fig. 7).

Second, it is clear from Fig. 7 that AMGA did not follow a fixed and straightforward strategy in solving different problems. For example, for the diabetes problem, AMGA pruned and added hidden neurons from the beginning to nearly the middle of the training process. After that, AMGA used only training to solve the diabetes problem. For the glass problem, AMGA tried to solve the problem first by pruning a few hidden neurons, then by training, and finally, by adding and pruning hidden neurons several times. A quite different scenario is observed for the easy thyroid problem. The algorithm AMGA first pruned several hidden neurons at the beginning of the training process, then trained the pruned ANN for a long time, and finally, added one neuron at the very end of the training process. These examples illustrate our intuition that an adaptive search strategy is needed in designing near optimal ANN architectures for solving different problems. It can be observed that the characteristics and complexity of all problems are not same (Tables I and II). Thus, the adaptive strategy is very much necessary to solve different problems efficiently.

Third, the training errors of ANNs reduced as their training processes progressed. However, there were some instances when the training errors increased. This was due to the merge operation that prunes hidden neurons. It is evident from Fig. 7 that the training error did not increase too much when AMGA pruned hidden neurons at any stage of the training. As a result, the pruned ANNs could reach to the previous error level after a small number of training epochs. This result indicates that, when ANNs were trained, some of their hidden neurons maintained much correlation with other neurons. This is quite natural because hidden neurons could not communicate with each other during training. The merge operation that produces one hidden neuron by combining two correlated hidden neurons seems to be a good alternative of the pruning operation in designing ANNs.

C. *Effect of Merging and Addition Strategies*

The previous section gives an idea about the performance of AMGA for different classification problems. However, the effects of pruning neurons by merging and adding neurons by splitting are not clear. To observe the effects of merging and splitting, we performed a set of new experiments. The setup of these experiments was exactly the same as those described previously. The only difference was that AMGA did not use here merging and splitting for pruning and adding hidden neurons, respectively. Rather, AMGA pruned neurons directly and added new neurons with random initial weights. This variant of AMGA is referred to as adaptive pruning and growing algorithm (APGA). To make a fair comparison, APGA used (3) and (8), respectively, to prune and add hidden neurons.

Table III shows the average results of our new experiments over 50 independent runs. The positive effect of pruning neurons by merging and adding neurons by splitting can be clearly understood from these results. For example, for the diabetes problem, the number of hidden neurons and epochs achieved by APGA were 5.36 and 420.7, respectively. These values were larger compared to those achieved by AMGA (Table II), which employed merge and split operations. It is worth mentioning that the amount of CPU time spent by AMGA in designing ANNs was also smaller compared to that spent by APGA. Although the merge operation needs some time to compute

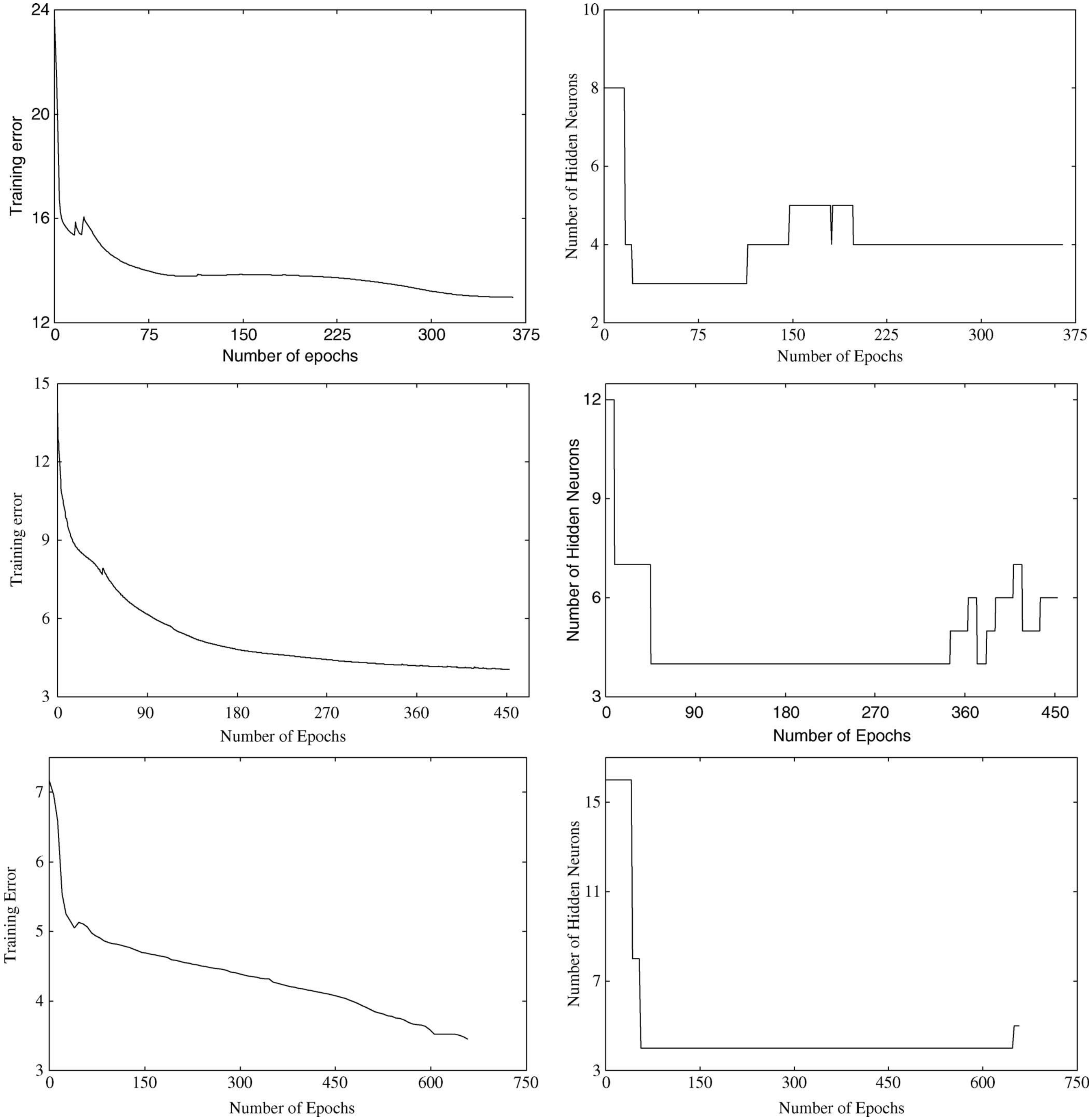


Fig. 7. ANN design process of AMGA for (top row) diabetes, (middle row) glass, and (bottom row) thyroid problems.

TABLE III
PERFORMANCE OF APGA, A VARIANT OF AMGA, ON EIGHT BENCHMARK CLASSIFICATION PROBLEMS. ALL RESULTS WERE AVERAGED OVER 50 INDEPENDENT RUNS

Problem	Number of		TER
	hidden neurons	epochs	
Cancer	1.88	249.1	1.32
Card	1.78	143.3	13.02
Diabetes	5.36	420.7	23.22
Gene	2.92	477.2	12.43
Glass	5.98	530.5	31.69
Heart	3.02	152.2	19.65
Iris	2.86	172.4	3.28
Thyroid	6.72	670.8	2.86

correlation information, AMGA needed a small amount of CPU time because it took a smaller number of epochs in designing ANNs.

It can also be observed that the average TER achieved by APGA was worse (larger) compared to that achieved by AMGA for different problems (Tables II and III). The larger TER achieved by APGA might be due to its requirement for solving problems using a large number of hidden neurons and epochs. The lower values of these two numbers have been widely recognized as important factors for obtaining a small TER. As mentioned earlier, the merge operation increases the capability of hidden neurons and encourages decorrelation among neurons.

TABLE IV
PERFORMANCE OF AMGA ON THREE BENCHMARK CLASSIFICATION PROBLEMS FOR DIFFERENT PARAMETER VALUES. ALL RESULTS WERE AVERAGED OVER 50 INDEPENDENT RUNS

Problem	Parameters				Number of		TER
	τ	ϵ	η_{th}	T	hidden neurons	epochs	
Diabetes	10	1E-02	0.03	1	4.52	360.6	22.39
	30	1E-04	0.04	4	4.16	430.3	21.61
	40	1E-06	0.07	7	4.64	710.5	25.26
	40	1E-06	0.07	10	6.08	1003.8	27.86
Glass	10	1E-02	0.03	1	4.84	440.5	30.18
	30	1E-04	0.04	4	5.02	512.3	31.13
	40	1E-06	0.07	7	5.73	901.5	33.96
	40	1E-06	0.07	10	6.17	1177.5	40.56
Thyroid	10	1E-02	0.03	1	6.10	596.3	2.58
	30	1E-04	0.04	4	5.92	627.5	3.16
	40	1E-06	0.07	7	6.62	1309.8	4.94
	40	1E-06	0.07	10	8.50	1786.1	5.63

It is natural to require less computational resources, i.e., a smaller number of hidden neurons and epochs, when hidden neurons are less correlated. This is because decorrelated hidden neurons would learn less redundant information than correlated hidden neurons. Since the split operation facilitates to add hidden neurons with some information, it is also beneficial for reducing training epochs.

In order to observe the significance in performance difference, we conducted a t-test between the results of AMGA and APGA. The t-test based on the number of hidden neurons, epochs, and TER indicates that AMGA was significantly better than APGA at 95% confidence interval, with the exception for the cancer problem. The TERs achieved by AMGA and APGA were found similar for the cancer problem. These results elucidate the essence of using merge and splitting operations in designing ANNs for different problems.

D. Effect of Different Parameter Values

As seen from Section III, AMGA introduces four parameters in training neural networks so that they can adapt their architectures during training. These parameters are τ, ε, η_{th}, and T. The results presented in Tables II and III were obtained for the specific values of these parameters. Thus, the effects of different parameter values are not known. The aim of this section is to observe such effects.

We performed a set of new experiments using AMGA with three different values for τ, ε, η_{th}, and T. The values for τ and ε were chosen in the range of 10–40 and 1E-02–1E-06, respectively, while those for η_{th} and T were chosen in the range of 0.03–0.07 and 1–10, respectively. The average results of the new experiments over 50 runs are presented in Table IV. It is seen that a small or moderate value of τ, η_{th}, and T, and a large value of ε are beneficial for both training epochs and TER. The proposed AMGA performed very badly when the value of T was chosen very large (e.g., ten). This is reasonable because AMGA terminated only when the validation error increased ten consecutive times in each of the ten consecutive strips. Thus, AMGA needed a large number of training epochs to satisfy the termination criterion. This allowed ANNs to learn very detailed information from the training data, resulting in poor generalization ability, i.e., a large TER (Table IV).

TABLE V
PERFORMANCE OF AGMA, A VARIANT OF AMGA, ON FIVE BENCHMARK CLASSIFICATION PROBLEMS. ALL RESULTS WERE AVERAGED OVER 50 INDEPENDENT RUNS

Problem	Number of		TER
	hidden neurons	epochs	
Cancer	3.04	220.5	2.87
Gene	3.22	445.7	14.37
Glass	6.40	480.5	32.07
Heart	4.58	138.6	18.98
Thyroid	10.12	650.8	4.95

E. Effect of Merge and Add Operations' Sequence

It is seen from Section III that AMGA executes the merge operation before the add operation. The reason for using such a sequence is to encourage AMGA in producing compact network architectures. The merge operation reduces the size of ANNs, while the add operation increases the size. As seen from Section III, the criterion for both merge and add operations is based on the training data. If the add operation is executed before the merge operation, the add operation is likely to be successful because it increases the processing power (hidden neurons) of ANNs. Thus, further training after the add operation will reduce the training error, and consequently, the chance for applying the merge operation will reduce. The result is the development of large ANN architectures.

We performed a new set of experiments to visualize the aforementioned fact. In these experiments, the add operation was executed before the merge operation. We call this variant of AMGA as adaptive growing and merging algorithm (AGMA). To make fair comparisons, the same experimental setup as described in Section IV-A was used here. Table V shows the average results of AGMA over 50 independent runs for five benchmark classification problems. The comparisons of the results presented in Table V with those presented in Table II indicate the essence of trying to execute the merge operation before the add operation. In terms of the average number of hidden neurons and TER, the performance of AMGA that tried to execute the merge operation before the add operation was found better than AGMA that tried to execute the add operation before the merge operation (Tables II and V). However, there were some instances when AMGA took more epochs than AGMA in designing ANNs (Tables II and V). This is reasonable in the sense that AMGA explores a large search space to produce compact network architectures.

F. Comparison

There are many algorithms in designing ANNs that one could compare against. However, a direct comparison with other algorithms using statistical tests is impractical at present, because different algorithms use different experimental methodologies. Moreover, the results of independent runs, which are necessary for statistical tests, are generally not available. Thus, it is impossible to compare different algorithms fairly unless one reimplements all algorithms under the same experimental setup. Since the aim of our experimental comparison here is to understand the strengths and weaknesses of AMGA, we implemented the basic constructive algorithm (BCA), basic pruning algorithm

TABLE VI
COMPARISON BETWEEN AMGA, BCA, BPA, AND BCPA ON EIGHT CLASSIFICATION PROBLEMS BASED ON THE NUMBER OF HIDDEN NEURONS IN ANN ARCHITECTURES DESIGNED BY THESE ALGORITHMS. ALL RESULTS WERE AVERAGED OVER 50 INDEPENDENT RUNS

Algorithm	Cancer	Card	Diabetes	Gene	Glass	Heart	Iris	Thyroid
AMGA	1.80	1.66	4.14	2.14	5.02	2.56	2.62	5.60
BCA	2.20	2.22	5.96	3.02	6.42	3.42	2.98	7.12
BPA	1.60	1.98	5.56	2.88	6.12	3.12	2.88	6.80
BCPA	2.12	2.02	5.80	2.92	6.02	3.26	2.88	6.92

TABLE VII
COMPARISON BETWEEN AMGA, BCA, BPA, AND BCPA ON EIGHT CLASSIFICATION PROBLEMS BASED ON THE NUMBER OF EPOCHS USED BY THESE ALGORITHMS IN DESIGNING ANN ARCHITECTURES. ALL RESULTS WERE AVERAGED OVER 50 INDEPENDENT RUNS

Algorithm	Cancer	Card	Diabetes	Gene	Glass	Heart	Iris	Thyroid
AMGA	234.5	131.8	390.7	440.3	499.5	130.6	165.3	630.1
BCA	290.2	126.3	467.5	411.5	580.7	173.4	188.9	732.3
BPA	263.3	143.5	409.1	423.5	543.8	161.4	175.9	690.3
BCPA	311.5	157.8	501.3	443.4	610.3	190.7	201.6	759.2

TABLE VIII
COMPARISON BETWEEN AMGA, BCA, BPA, AND BCPA ON EIGHT CLASSIFICATION PROBLEMS BASED ON THE TER OF THE ANNS DESIGNED BY THESE ALGORITHMS. ALL RESULTS WERE AVERAGED OVER 50 INDEPENDENT RUNS

Algorithm	Cancer	Card	Diabetes	Gene	Glass	Heart	Iris	Thyroid
AMGA	1.30	12.67	21.97	12.05	30.56	18.87	1.89	2.44
BCA	1.92	13.41	26.04	12.73	32.45	20.34	2.71	3.00
BPA	1.89	13.83	26.25	12.40	32.83	19.93	1.84	2.67
BCPA	1.95	13.72	26.22	15.59	33.20	20.43	2.71	2.83

(BPA), and basic constructive–pruning algorithm (BCPA) as shown in Figs. 1–3.

The algorithms BCA and BCPA employed the same neuron addition criterion as the one used in AMGA, but they added neurons with random initial connection weights. Since BPA and BCPA do not use merging, they deleted hidden neurons by pruning instead by merging. For pruning, BPA and BCPA used (3) without its denominator to compute the significance of hidden neurons in an ANN. The denominator of (3) was not needed for pruning because BPA and BCPA started pruning after the ANN converges to a minimum error. The same experimental setup as described in Section IV-A was used for performing experiments using BCA, BPA, and BCPA. The initial ANN architecture for BCA and BCPA consisted of a three-layered ANN with one hidden neuron. However, the number of hidden neurons in the initial ANN of BPA was chosen two times the number of hidden neurons shown in Table II. All these arrangements were made to make a fair comparison so that the essence of using an adaptive strategy in designing ANNs can be understood.

Tables VI–VIII presents the average results of AMGA, BCA, BPA, and BCPA over 50 independent runs for eight benchmark classification problems. It can be seen that the ANNs designed by AMGA had the smallest number of hidden neurons for seven out of eight problems and the second smallest (next to BPA) for one problem (Table VI). In terms of average training epochs, AMGA took the smallest number of training epochs for six out of eight problems and the second smallest for two problems (Table VII). This is reasonable because AMGA explored more search space in designing ANNs than the other three algorithms. However, the ANNs designed by AMGA achieved the lowest TER for all eight problems (Table VIII).

The t-test based on the number of hidden neurons, epochs, and TER shows that AMGA was significantly better than BCA, BPA, and BCPA at 95% confidence level for all problems, with the exception of the cancer and gene problems. The BPA was found significantly better than AMGA with respect to the number of hidden neurons of the cancer problem, while BCA outperformed AMGA with respect to the number of epochs of the gene problem. Since we implemented AMGA, BCA, BPA, and BCPA under the same experimental setup, the results of the t-test indicate the essence of the three components, i.e., adaptive search strategy, pruning neurons by merging, and adding neurons by splitting, used in AMGA for designing ANNs.

The previous comparison shows that AMGA is better than its classical counterparts. It is interesting to compare the performance of AMGA with state-of-the-art algorithms. We therefore compared the results of AMGA with those of feedforward neural network construction algorithm (FNNCA) [18], constructive feedforward neural networks (CFNN) [23], optimal brain damage (OBD) [24], optimal brain surgeon (OBS) [25], variance nullity pruning (VNP) [26], and optimization methodology for neural network (OMNN) [74]. The CFNN and FNNCA used a constructive approach in designing ANNs, while VNP, OBD, and OBS used a pruning approach. The algorithm OMNN used a hybrid simulated annealing [75] and tabu [76] search methods in designing ANNs.

Tables IX–XI present the results of the aforementioned algorithm along with AMGA. The results of AMGA were averaged over 50 independent runs, while they were averaged over 40 and 30 independent runs for CFNN and OMNN, respectively. The result of FNNCA was the best result of five independent runs, while those for OBD, OBS, and VNP were not mentioned in [26]. In terms of TER, AMGA was found better compared to all other algorithms. In terms of hidden neurons, AMGA was also found better than all other algorithms for seven out of eight problems. For one (the cancer) problem, the ANN

TABLE IX
COMPARISON BETWEEN AMGA, FNNCA [18], CFNN [23], OBD [24], OBS [25], AND VNP [26] ON THE CANCER PROBLEM. THE RESULTS OF AMGA WERE AVERAGED OVER 50 INDEPENDENT RUNS, WHILE THOSE OF CFNN WERE AVERAGED OVER 40 INDEPENDENT RUNS. THE RESULTS OF FNNCA [18] WERE THE BEST RESULTS OF FIVE RUNS, WHILE IT WAS NOT MENTIONED IN [26] WHETHER THOSE FOR VNP [26], OBD [24], AND OBS [25] WERE THE AVERAGE OR THE BEST

	AMGA	FNNCA	CFNN	OBD	OBS	VNP
Number of hidden neurons	1.80	2.0	2.0	8.0	7.0	1.0
TER	1.30	1.40	3.30	7.50	10.0	2.20

TABLE X
COMPARISON BETWEEN AMGA, OBD [24], OBS [25], VNP [26], AND OMNN [74] ON THE DIABETES AND IRIS PROBLEMS. THE RESULTS OF AMGA WERE AVERAGED OVER 50 INDEPENDENT RUNS, WHILE THOSE OF OMNN WERE AVERAGED OVER 30 INDEPENDENT RUNS. IT WAS NOT MENTIONED WHETHER THE RESULTS OF OBD, OBS, AND VNP WERE THE AVERAGE OR THE BEST

Problem		AMGA	OBD	OBS	VNP	OMNN
Diabetes	Number of hidden neurons	4.14	16.0	26.0	8.0	4.53
	TER	21.97	31.40	34.60	30.90	25.87
Iris	Number of hidden neurons	2.62	4.0	4.0	2.0	2.65
	TER	1.89	2.00	2.00	2.30	4.61

TABLE XI
COMPARISON BETWEEN AMGA AND OMNN [74] ON THE THYROID PROBLEM. THE RESULTS OF AMGA WERE AVERAGED OVER 50 INDEPENDENT RUNS, WHILE THOSE OF OMNN WERE AVERAGED OVER 30 INDEPENDENT RUNS

	AMGA	OMNN
Number of hidden neurons	5.60	6.39
TER	2.44	7.33

TABLE XII
COMPARISON BETWEEN AMGA AND EPNET [61] ON THREE CLASSIFICATION PROBLEMS. THE RESULTS OF BOTH ALGORITHMS WERE AVERAGED OVER 50 INDEPENDENT RUNS

Problem		AMGA	EPNet
Cancer	Number of hidden neurons	1.80	2.00
	TER	1.30	1.37
Diabetes	Number of hidden neurons	4.14	3.40
	TER	21.97	22.37
Thyroid	Number of hidden neurons	5.60	5.90
	TER	2.44	2.13

architecture designed by VNP had a lesser number of hidden neurons compared to that designed by AMGA.

In order to compare the performance of AMGA with evolutionary approaches used in designing ANNs, EPNet [61] is considered for comparisons. The algorithm EPNet can add or prune hidden neurons as AMGA does during the training process of an ANN. However, there are two different ways that AMGA and EPNet execute the pruning and addition operations. First, EPNet uses a predefined, fixed, and greedy strategy in executing prune and add operations, while AMGA uses an adaptive strategy. For example, EPNet executes the prune operation first. If this operation is found unsuccessful, EPNet then executes the add operation. At this point, one may think that AMGA also uses a greedy strategy. This is not the case because AMGA checks the merging (pruning) criterion first and then the addition criterion. Based on the fulfillment of these criteria, AMGA executes the merging and/or addition operations. For different problems and learning parameters, AMGA, therefore, may execute merge and add operations repeatedly, alternatively, or in any other sequences. Second, EPNet selects a number of hidden neurons randomly for pruning. This number is specified by a user. Pruning hidden neurons by random selection may deteriorate the performance of ANNs significantly, because it is not known whether the pruned neurons are unimportant. In contrast, AMGA prunes hidden neurons indirectly by merging. It merges an insignificant neuron with a significant neuron that maintains the highest correlation with the insignificant one. In AMGA, the number of hidden neurons to be pruned is dependent on the significance of hidden neurons. Finally, EPNet is an evolutionary approach, while AMGA is a nonevolutionary approach. The ideas used in AMGA, however, can be applied to evolutionary approaches. One of the future works could be to apply the ideas used by AMGA to an evolutionary approach for designing ANNs.

Table XII presents the average results of AMGA and EPNet over 50 independent runs. In terms of average number of hidden neurons and TER, AMGA was found better than EPNet for two out of three problems we compared here. However, in terms of epochs, AMGA was found far better than EPNet for all three problems. For example, EPNet required on average of 109 000 epochs for a single run [61], while AMGA required less than 1000 epochs (Table II). The highest number of epochs spent by AMGA was for the thyroid problem, and it was 670.8 epochs. It is known that the important parameters of any ANN design algorithm are the consideration of generalization ability and training time [29], [77]. However, both of these parameters are important in many application areas; improving one at the expense of the other becomes a crucial decision [77].

G. Discussion

This section briefly explains why the performance of AMGA is better than those of the other algorithms we compared in Tables VI–XI. There are three major differences that might contribute to the better performance of AMGA compared to other algorithms.

The first reason is that AMGA uses an adaptive strategy in designing ANNs, while BCA, BPA, BCPA, FNNCA [18], CFNN [23], OBD [24], OBS [25], VNP [26], and OMNN [74] use a predefined and same greedy strategy for all problems. It can be observed that the characteristics and complexity of all problems are not same (Table I, Tables VI–IX). Thus, the use of the same strategy may not be suitable for all problems.

The significance of using an adaptive strategy can be easily understood by comparing the performances of BCA, BPA, BCPA, VNP, OBD, and OBS with that of an adaptive-strategy-based algorithm OMNN. For example, for the complex diabetes problem, the numbers of hidden neurons in ANN architectures designed by BCA, BPA, BCPA, OBD, OBS, and VNP were 5.96, 5.56, 5.80, 16.00, 26.00, and 8.00, respectively, and the TERs achieved by these ANNs were 26.04, 26.25, 26.22, 31.40, 34.60, and 30.90, respectively. In terms of average results, the number of hidden neurons in ANN architectures designed by OMNN was 4.53, and the TER achieved by these ANNs was 25.87. However, AMGA performed better than OMNN. In terms of average results, the number of hidden neurons in ANN architectures designed by AMGA was 4.14, and the TER achieved by these ANNs was 21.97.

The second reason is the training method adopted in designing ANNs. The algorithms BCA, BCPA, CFNN [23], and FNNCA [18] add hidden neurons to an ANN one by one and train the ANN after each addition operation. The problem of this scheme is the long-time training of some hidden neurons that are added in the earlier part of a training process. These neurons, therefore, may learn redundant information and also overfit the training data. On the other hand, BPA, OBD, OBS, and VNP start the training process with a large ANN architecture. The OBD and OBS start pruning when the training error of the ANN reaches to a minimum value, while BPA and VNP start pruning when overfitting begins, which is monitored by utilizing a validation set. Although the use of a large ANN architecture expedites the initial training process, the retraining of the pruned ANN after each pruning step may need many training epochs to compensate the effect of pruning. This problem is acute when an initial ANN architecture is assigned to be very large, because in this case, an algorithm needs to execute many pruning steps to find a near optimal ANN architecture. Since a pruning algorithm needs to start the training process with a large ANN architecture and one does not precisely quantify such a large architecture in advance, it is very likely that the initial ANN architecture can be very large. To overcome the problem associated with the use of a minimal and a large architecture used by constructive and pruning algorithms, respectively, AMGA allows the training process of an ANN to start with any number of hidden neurons. This is possible because AMGA can prune or add hidden neurons any time during training. To reduce the amount of retraining, AMGA tries to compensate the effect of pruning by increasing the capability of some remaining hidden neurons in the ANN and adds hidden neurons by splitting existing hidden neurons. Our experimental studies reveal the essence of using the compensation technique and addition with splitting in designing ANNs (Section IV-C).

The third reason is the encouragement of decorrelation among hidden neurons in an ANN. The proposed AMGA encourages decorrelation among hidden neurons in the ANN by merging correlated hidden neurons. This is beneficial in the sense that the ANN with a small number of hidden neurons can learn more information when hidden neurons are less correlated. In other words, decorrelation among hidden neurons may help to produce compact ANN architectures. This may be the reason that ANNs designed by AMGA had lesser numbers of hidden neurons compared to those of other algorithms (Tables IX–XI). The only exception is the cancer problem in which an ANN designed by VNP [26] had a lesser number of hidden neurons than that designed by AMGA. Since the number of hidden neurons needed to solve the cancer problem was small, the encouragement of decorrelation did not work. In addition, it is not known whether the result presented in [26] for VNP was the best result or the average result of several runs. The best ANN designed by AMGA had also one hidden neuron.

V. Conclusion

The generalization ability of ANNs is greatly dependent on their architectures. Although a number of algorithms exist to design ANNs automatically, most existing algorithms use a kind of greedy strategy in determining a near optimal ANN architecture for a given problem. This paper describes a new algorithm, AMGA, in designing ANNs automatically. The idea behind AMGA is to put more emphasis on an adaptive strategy and to reduce retraining epochs in the design process of ANNs. The adaptive strategy is better suited due to its ability to cope with different conditions that may arise at different stages during the design process of ANNs. This strategy also has a lesser chance to trap into architectural local optima, a common problem suffered by the greedy strategy. The reduction of retraining epochs is suitable for undermining the effect of overtraining, which has a detrimental effect on the generalization ability of ANNs.

The adaptive strategy of AMGA allows the pruning or addition of hidden neurons any time during the ANNs' training, depending on the condition of the learning ability of hidden neurons or the training progress of ANNs, respectively. A merging operation is used in AMGA to prune hidden neurons. This operation produces one new neuron by merging two correlated neurons in such a way that the effect of the new neuron to the ANN is nearly the same as the combined effect of its two parent neurons. This kind of pruning is completely different from the one used in existing algorithms. The essence of this pruning mechanism is that it encourages decorrelation among hidden neurons in ANNs, resulting in compact ANN architectures. Moreover, this pruning mechanism reduces the retraining epochs. In adding hidden neurons, AMGA adds a hidden neuron by splitting an existing hidden neuron in an ANN. All these techniques are adopted in AMGA for designing compact ANN architectures with good generalization ability.

The extensive experiments reported in this paper have been carried out to evaluate how well AMGA performed on different problems compared to other algorithms. In almost all cases, AMGA outperformed the others (Tables VI–IX). In its current implementation, AMGA has a few user-specified parameters although this is not unusual in the field. These parameters, however, are not very sensitive to moderate changes. One of the future improvements to AMGA would be to reduce the number of parameters or make them adaptive. In addition, the use of a different significance criterion in the merging operation of AMGA would also be an interesting future research topic. Since AMGA has been applied to the classification problems, it would be interesting to study how well AMGA would perform on regression problems.

Acknowledgment

The authors would like to thank the Associate Editor and the anonymous reviewers for their constructive comments.

References

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing*, vol. I, D. E. Rumelhart and J. L. McClelland, Eds. Cambridge, MA: MIT Press, 1986, pp. 318–362.

[2] T. Y. Kwok and D. Y. Yeung, "Constructive algorithms for structure learning in feedforward neural networks for regression problems," *IEEE Trans. Neural Netw.*, vol. 8, no. 3, pp. 630–645, May 1997.

[3] R. Reed, "Pruning algorithms—A survey," *IEEE Trans. Neural Netw.*, vol. 4, no. 5, pp. 740–747, Sep. 1993.

[4] F. Girosi, M. Jones, and T. Poggio, "Regularization theory and neural networks architectures," *Neural Comput.*, vol. 7, no. 2, pp. 219–269, Mar. 1995.

[5] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: Univ. Michigan Press, 1975.

[6] L. J. Fogel, A. J. Owens, and M. J. Walsh, *Artificial Intelligence Through Simulated Evolution*. New York: Wiley, 1966.

[7] D. B. Fogel, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. New York: IEEE Press, 1995.

[8] H.-P. Schwefel, *Numerical Optimization of Computer Models*. Chichester, U.K.: Wiley, 1981.

[9] A. S. Weigend, D. E. Rumelhart, and B. A. Huberman, "Generalization by weight-elimination with application to forecasting," in *Proc. Advances Neural Inform. Process. Syst.*, R. Lippmann, J. Moody, and D. S. Touretzky, Eds., 1991, vol. 3, pp. 875–882.

[10] C. Schittenkopf, G. Deco, and W. Brauer, "Two strategies to avoid overfitting in feedforward neural networks," *Neural Netw.*, vol. 10, pp. 804–818, 1997.

[11] D. J. C. MacKay, "Bayesian interpolation," *Neural Comput.*, vol. 4, no. 3, pp. 415–447, May 1992.

[12] P. Lauret, E. Fock, and T. A. Mara, "A node pruning algorithm based on a Fourier amplitude sensitivity test method," *IEEE Trans. Neural Netw.*, vol. 17, no. 2, pp. 273–293, Mar. 2006.

[13] D. J. C. MacKay, "A practical Bayesian framework for backpropagation networks," *Neural Comput.*, vol. 4, no. 3, pp. 448–472, May 1992.

[14] P. J. Angeline, G. M. Saunders, and J. B. Pollack, "An evolutionary algorithm that constructs recurrent neural networks," *IEEE Trans. Neural Netw.*, vol. 5, no. 1, pp. 54–65, Jan. 1994.

[15] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

[16] J. P. Nadal, "Study of a growth algorithm for a feedforward network," *Int. J. Neural Syst.*, vol. 1, no. 1, pp. 55–59, 1989.

[17] N. Burgess, "A constructive algorithm that converges for real-valued input patterns," *Int. J. Neural Syst.*, vol. 5, no. 1, pp. 59–66, Mar. 1994.

[18] R. Setiono and L. C. K. Hui, "Use of a quasi-Newton method in a feedforward neural network construction algorithm," *IEEE Trans. Neural Netw.*, vol. 6, no. 1, pp. 273–277, Jan. 1995.

[19] T. Y. Kwok and D. Y. Yeung, "Objective functions for training new hidden units in constructive neural networks," *IEEE Trans. Neural Netw.*, vol. 8, no. 5, pp. 1131–1148, Sep. 1997.

[20] M. Lehtokangas, "Fast initialization for cascade-correlation learning," *IEEE Trans. Neural Netw.*, vol. 10, no. 2, pp. 410–414, Mar. 1999.

[21] M. Lehtokangas, "Modified cascade-correlation learning for classification," *IEEE Trans. Neural Netw.*, vol. 11, no. 3, pp. 795–798, May 2000.

[22] D. Liu, T.-S. Chang, and Y. Zhang, "A constructive algorithm for feedforward neural networks with incremental training," *IEEE Trans. Circuits Syst. I, Fundam. Theory Appl.*, vol. 49, no. 12, pp. 1876–1879, Dec. 2002.

[23] L. Ma and K. Khorasani, "Constructive feedforward neural networks using Hermite polynomial activation functions," *IEEE Trans. Neural Netw.*, vol. 16, no. 4, pp. 821–833, Jul. 2005.

[24] Y. Le Cun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Proc. Advances Neural Inform. Process. Syst.*, D. S. Touretzky, Ed. San Mateo, CA: Morgan Kaufmann, 1990, vol. 2, pp. 598–605.

[25] B. Hassibi and D. G. Stork, "Second-order derivatives for network pruning: Optimal brain surgeon," in *Proc. Advances Neural Inform. Process. Syst.*, C. Lee, S. Hanson, and J. Cowan, Eds. San Mateo, CA: Morgan Kaufmann, 1993, vol. 5, pp. 164–171.

[26] A. P. Engelbrecht, "A new pruning heuristic based on variance analysis of sensitivity information," *IEEE Trans. Neural Netw.*, vol. 12, no. 6, pp. 1386–1399, Nov. 2001.

[27] Y. Hirose, K. Yamashita, and S. Hijiya, "Back-propagation algorithm which varies the number of hidden units," *Neural Netw.*, vol. 4, no. 1, pp. 61–66, 1991.

[28] Md. M. Islam and K. Murase, "An algorithm for automatic design of two hidden layered artificial neural networks," in *Proc. IJCNN*, 2000, pp. 467–472.

[29] Md. M. Islam and K. Murase, "A new algorithm to design compact two-hidden-layer artificial neural networks," *Neural Netw.*, vol. 14, no. 9, pp. 1265–1278, Nov. 2001.

[30] I. Rivals and L. Personnaz, "Neural-network construction and selection in nonlinear modeling," *IEEE Trans. Neural Netw.*, vol. 14, no. 4, pp. 804–819, Jul. 2003.

[31] R. A. Jacobs, "Increased rates of convergence through learning rate adaptation," *Neural Netw.*, vol. 1, no. 4, pp. 295–307, 1988.

[32] Y. Chauvin, "Generalization performance of overtrained backpropagation networks," in *Proc. EUROSZP Workshop*, 1990, pp. 46–55.

[33] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Netw.*, vol. 2, no. 5, pp. 359–366, 1989.

[34] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Netw.*, vol. 4, no. 2, pp. 251–257, 1991.

[35] J. Park and I. W. Sandberg, "Universal approximation using radial basis function networks," *Neural Comput.*, vol. 3, no. 2, pp. 246–257, 1991.

[36] K. Hornik, "Some new results on neural-network approximation," *Neural Netw.*, vol. 6, pp. 1069–1072, 1993.

[37] J. Park and I. W. Sandberg, "Approximation and radial-basis-function networks," *Neural Comput.*, vol. 5, no. 2, pp. 305–316, Mar. 1993.

[38] J. D. Schaffer, D. Whitley, and L. J. Eshelman, "Combinations of genetic algorithms and neural networks: A survey of the state of the art," in *Proc. Int. Workshop COGANN*, D. Whitley and J. D. Schaffer, Eds., 1992, pp. 1–37.

[39] X. Yao, "A review of evolutionary artificial neural networks," *Int. J. Intell. Syst.*, vol. 8, no. 4, pp. 539–567, 1993.

[40] X. Yao, "Evolving artificial neural networks," *Proc. IEEE*, vol. 87, no. 9, pp. 1423–1447, Sep. 1999.

[41] T. Ash, "Dynamic node creation in backpropagation networks," *Connect. Sci.*, vol. 1, no. 4, pp. 365–375, 1989.

[42] M. Wynne-Jones, "Node splitting: A constructive algorithm for feedforward neural networks," *Neural Comput. Appl.*, vol. 1, no. 1, pp. 17–22, Mar. 1993.

[43] T. M. Nabhan and A. Y. Zomaya, "Toward generating neural network structures for function approximation," *Neural Netw.*, vol. 7, no. 1, pp. 89–99, 1994.

[44] S. Young and T. Downs, "CARVE—A constructive algorithm for real-valued examples," *IEEE Trans. Neural Netw.*, vol. 9, no. 6, pp. 1180–1190, Nov. 1998.

[45] M. Iqbal Bin Shahid, Md. M. Islam, M. A. H. Akhand, and K. Murase, "A new algorithm to design multiple hidden layered artificial neural networks," in *Proc. Joint 3rd Int. Conf. SCIS, 7th ISIS*, Tokyo, Japan, Sep. 20–24, 2006, pp. 463–468.

[46] Md. A. Sattar, Md. M. Islam, and K. Murase, "A new constructive algorithm for designing and training artificial neural networks," in *Proc. 14th ICONIP*, Kitakyushu, Japan, Nov. 13–16, 2007, pp. 317–327.

[47] Md. M. Islam, Md. F. Amin, S. Ahmmed, and K. Murase, "An adaptive merging and growing algorithm for designing artificial neural networks," in *Proc. Int. Joint Conf. Neural Netw.*, Hong Kong, Jun. 1–8, 2008, pp. 2004–2009.

[48] J. E. Moody, "Prediction risk and architecture selection for neural networks," in *From Statistics to Neural Networks: Theory and Pattern Recognition Applications*, V. Cherkassky, J. H. Friedman, and H. Wechsler, Eds. New York: Springer-Verlag, 1993, pp. 147–165.

[49] J. Torreele, "Temporal processing with recurrent networks: An evolutionary approach,"in *Proc. 4th Int. Conf. Genetic Algorithms*, R. K. Belew and L. B. Booker, Eds., 1991, pp. 555–561.

[50] M. Mandischer, "Evolving recurrent neural networks with non-binary encoding," in *Proc. IEEE Int. Conf. Evol. Comput.*, 1995, pp. 584–589. Part 2(of 2).

[51] F. Heimes, G. Zalesski, W. Land, Jr., and M. Oshima, "Traditional and evolved dynamic neural networks for aircraft simulation," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, 1997, pp. 1995–2000. Part 3(of 5).

[52] D. Whitley, T. Starkweather, and C. Bogart, "Genetic algorithms and neural networks: Optimizing connections and connectivity," *Parallel Comput.*, vol. 14, no. 3, pp. 347–361, Aug. 1990.

[53] D. Whitley and T. Starkweather, "Optimizing small neural networks using a distributed genetic algorithm," in *Proc. Int. Joint Conf. Neural Netw.* Hillsdale, NJ: Lawrence Erlbaum, 1990, vol. I, pp. 206–209.

[54] X. Yao and Y. Shi, "A preliminary study on designing artificial neural networks using co-evolution," in *Proc. IEEE Singapore Int. Conf. Intell. Control Instrum.*, Singapore, Jun. 1995, pp. 149–154.

[55] D. J. Chalmers, "The evolution of learning: An experiment in genetic connectionism," in *Proc. Connectionist Models Summer School*, D. S. Touretzky, J. L. Elman, and G. E. Hinton, Eds., 1990, pp. 81–90.

[56] Y. Bengio and S. Bengio, "Learning a synaptic learning rule," in "Dét. Informatique et de Recherche Operationelle," Univ. Montreal, Montreal, QC, Canada, Tech. Rep. 751, Nov. 1990.

[57] J. F. Fontanari and R. Meir, "Evolving a learning algorithm for the binary perceptron," *Network*, vol. 2, no. 4, pp. 353–359, Nov. 1991.

[58] J. R. Koza and J. P. Rice, "Genetic generation of both the weights and architecture for a neural network," in *Proc. IEEE IJCNN*, Seattle, WA, 1991, vol. 2, pp. 397–404.

[59] S. Bornholdt and D. Graudenz, "General asymmetric neural networks and structure design by genetic algorithms," *Neural Netw.*, vol. 5, no. 2, pp. 327–334, 1992.

[60] S. Oliker, M. Furst, and O. Maimon, "A distributed genetic algorithm for neural network design and training," *Complex Syst.*, vol. 6, pp. 459–477, 1992.

[61] X. Yao and Y. Liu, "A new evolutionary system for evolving artificial neural networks," *IEEE Trans. Neural Netw.*, vol. 8, no. 3, pp. 694–713, May 1997.

[62] A. Mahmood, S. Sharmin, D. Barua, and Md. M. Islam, "Graph matching recombination for evolving neural networks," in *Proc. 4th Int. Symp. Neural Netw.*, Nanjing, China, Jun. 3–7, 2007, pp. 562–568.

[63] P. Baldi and Y. Chauvin, "Temporal evolution of generalization during learning in linear networks," *Neural Comput.*, vol. 3, no. 4, pp. 589–603, 1991.

[64] Md. M. Islam and K. Murase, "A new crossover operator and its application to artificial neural networks evolution," *IEICE Trans. Inf. Syst.*, vol. E84-D, no. 9, pp. 1144–1154, 2001.

[65] S. E. Fahlman and C. Lebiere, "The cascade-correlation learning architecture," in *Proc. Advances Neural Inform. Process. Syst.*, D. S. Touretzky, Ed. San Mateo, CA: Morgan Kaufmann, 1990, vol. 2, pp. 524–532.

[66] S. Haykin, *Neural Networks: A Comprehensive Foundation*. Englewood Cliffs, NJ: Prentice-Hall, 1999.

[67] A. I. Rasiah, R. Togneri, and Y. Attikiouzel, "Modelling 1-D signals using Hermite basis functions," *Proc. Inst. Elect Eng.—Vis. Image Signal Process.*, vol. 144, no. 6, pp. 345–354, Dec. 1997.

[68] L. Prechelt, "PROBEN1—A set of neural network benchmark problems and benchmarking rules," Faculty Informatics, Univ. Karlsruhe, Karlsruhe, Germany, Tech. Rep. 21/94, 1994.

[69] Md. M. Islam, X. Yao, and K. Murase, "A constructive algorithm for training cooperative neural network ensembles," *IEEE Trans. Neural Netw.*, vol. 14, no. 4, pp. 820–834, Jul. 2003.

[70] S. V. Odri, D. P. Petrovacki, and G. A. Krstonosic, "Evolutional development of a multilevel neural network," *Neural Netw.*, vol. 6, no. 4, pp. 583–595, 1993.

[71] L. Prechelt, "Automatic early stopping using cross validation: Quantifying the criteria," *Neural Netw.*, vol. 11, no. 4, pp. 761–767, Jun. 1998.

[72] L. Prechelt, "Some notes on neural learning algorithm benchmarking," *Neurocomputing*, vol. 9, no. 3, pp. 343–347, Dec. 1995.

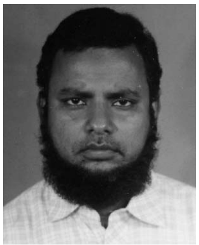
[73] L. Prechelt, "A quantitative study of experimental evaluations of neural network learning algorithms," *Neural Netw.*, vol. 9, no. 3, pp. 457–462, Apr. 1996.

[74] T. B. Ludermir, A. Yamazaki, and C. Zanchettin, "An optimization methodology for neural network weights and architectures," *IEEE Trans. Neural Netw.*, vol. 17, no. 6, pp. 1452–1459, Nov. 2006.

[75] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983.

[76] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Comput. Oper. Res.*, vol. 13, no. 5, pp. 533–549, May 1986.

[77] K. Jim, C. L. Giles, and B. G. Horne, "An analysis of noise in recurrent neural networks: Convergence and generalization," *IEEE Trans. Neural Netw.*, vol. 7, no. 6, pp. 1424–1438, Nov. 1996.



Md. Monirul Islam received the B.E. degree from the Bangladesh Institute of Technology, now Khulna University of Engineering and Technology (KUET), Khulna, Bangladesh, in 1989, the M.E. degree from the Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh, in 1996, and the Ph.D. degree from the University of Fukui, Fukui, Japan, in 2002.

He was a Lecturer and an Assistant Professor from 1989 to 2002 with KUET. He has been with BUET since 2003, where he was an Assistant Professor of Computer Science and Engineering and is currently an Associate Professor. Currently, he is a Visiting Associate Professor at the University of Fukui, supported by the Japanese Society for Promotion of Science. His major research interests include evolutionary robotics, evolutionary computation, neural networks, machine learning, pattern recognition, and data mining. He has more than 80 refereed publications.

Dr. Islam was the recipient of the First Prize in The Best Paper Award Competition of the Joint 3rd International Conference on Soft Computing and Intelligent Systems and 7th International Symposium on Advanced Intelligent Systems.

Md. Abdus Sattar received the B.Sc. degree in electrical and electronic engineering from the Engineering College, Rajshahi (ECR), now the Rajshahi University of Engineering and Technology, Rajshahi, Bangladesh, in 1975 and the M.Sc. degree in computer science and engineering from the Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh, in 1990.

He was a Lecturer with ECR and then moved to serve in industry and as a Consultant Engineer abroad. He then returned back to academy as an Assistant Professor with the Electrical and Electronic Engineering Department, Bangladesh Institute of Technology, Rajshahi, in 1987. Since 1992, he has been an Assistant Professor with the Computer Science and Engineering Department, BUET. His area of research includes neural networks, bioinformatics, and pattern recognition. His special interest is in implementing the Bangla language in computer usage.

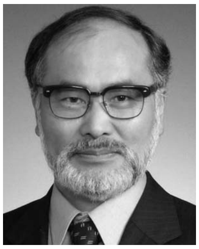
Md. Faijul Amin received the B.Sc. degree in computer science and engineering from the Bangladesh University of Engineering and Technology, Dhaka, Bangladesh, in 2004. He is currently working toward the M.S. degree in the Department of Human and Artificial Intelligence Systems, University of Fukui, Fukui, Japan.

He is also with the Khulna University of Engineering and Technology, Khulna, Bangladesh. His research interest includes artificial neural networks and their mathematical foundations.

Xin Yao (M'91–SM'96–F'03) received the B.Sc. and Ph.D. degrees from the University of Science and Technology of China (USTC), Hefei, China, in 1982 and 1990, respectively, and the M.Sc. degree from the North China Institute of Computing Technology, Beijing, China, in 1985.

He was an Associate Lecturer and Lecturer from 1985 to 1990 with USTC, while working toward the Ph.D. degree on simulated annealing and evolutionary algorithms. He took up a postdoctoral fellowship with the Computer Sciences Laboratory, Australian National University, Canberra, Australia, in 1990 and continued his work on simulated annealing and evolutionary algorithms. He was with the Knowledge-Based Systems Group, Division of Building, Construction and Engineering, Commonwealth Scientific and Industrial Research Organisation, Melbourne, Australia, in 1991, working primarily on an industrial project on automatic inspection of sewage pipes. He returned to Canberra, in 1992, to take up a lectureship with the School of Computer Science, University College, University of New South Wales, Australian Defence Force Academy, where he was later promoted to a Senior Lecturer and Associate Professor. Attracted by the English weather, he moved to the University of Birmingham, Birmingham, U.K., as a Professor of Computer Science, in 1999, where he is currently the Director of the Centre of Excellence for Research in Computational Intelligence and Applications. He is also a Changjiang (Visiting) Chair Professor (Cheung Kong Scholar) at the USTC, Hefei. He is an Associate Editor or Editorial Board Member of ten other journals and the Editor of the World Scientific Book Series on Advances in Natural Computation. He has given more than 50 invited keynote and plenary speeches at conferences and workshops worldwide. His major research interests include evolutionary artificial neural networks (ANNs), automatic modularization of machine learning systems, evolutionary optimization, constraint handling techniques, computational time complexity of evolutionary algorithms, coevolution, iterated prisoner's dilemma, data mining, and real-world applications. He has more than 250 refereed publications.

Dr. Yao is the Editor-in-Chief of the IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION. He was the recipient of the President's Award for Outstanding Thesis by the Chinese Academy of Sciences for his Ph.D. work on simulated annealing and evolutionary algorithms in 1989. He won the 2001 IEEE Donald G. Fink Prize Paper Award for his work on evolutionary ANNs.



Kazuyuki Murase received the M.E. degree in electrical engineering from Nagoya University, Nagoya, Japan, in 1978 and the Ph.D. degree in biomedical engineering from Iowa State University, Ames, in 1983.

He has been a Professor with the Department of Human and Artificial Intelligence Systems, Graduate School of Engineering, University of Fukui, Fukui, Japan, since 1999, where he was an Associate Professor with the Department of Information Science in 1988 and a Professor in 1992. He was a Research Associate with the Department of Information Science, Toyohashi University of Technology, Toyohashi, Japan, in 1984.

Dr. Murase is a member of The Institute of Electronics, Information and Communication Engineers, the Japanese Society for Medical and Biological Engineering, the Japan Neuroscience Society, the International Neural Network Society, and the Society for Neuroscience. He serves on the Board of Directors of the Japan Neural Network Society, is a Councilor of the Physiological Society of Japan, and is a Councilor of the Japanese Association for the Study of Pain.