JOURNAL OF AIRCRAFT
Vol. 39, No. 3, May–June 2002

Designing Compact Feedforward Neural Models with Small Training Data Sets

Roxana M. Greenman*
Lawrence Livermore National Laboratory, Livermore, California 94551
Slawomir W. Stepniewski†
SONY Electronics, San Jose, California 95134
and
Charles C. Jorgensen‡ and Karlin R. Roth§
NASA Ames Research Center, Moffett Field, California 94035

A new hybrid method is presented for designing feedforward, backpropagation neural models with small training data sets. The method minimizes the generalization error, a fundamental quantity that characterizes the effectiveness of the regression models. It combines into one framework a bootstrap technique that estimates network generalization performance and a collection of stochastic and deterministic optimization techniques that adjust neural network interconnection geometry. The approach is derived as a form of multi-objective optimization strategy. This allows for more direct treatment of contradictory design criteria than traditionally employed single-objective techniques. A stochastic optimization method such as a genetic algorithm is used to select activation functions for hidden-layer nodes, whereas fast deterministic techniques, optimal brain surgeon and singular value decomposition, are used to perform connection and node pruning. The method is demonstrated by optimizing neural networks that model the high-lift aerodynamics of a multi-element airfoil. The neural model is constructed using a small computational data set consisting of 227 data points. In the numerical experiments presented, the solutions produced by this hybrid approach exhibit an improvement in the generalization ability on the average of five to six times when compared to the pruned models with only one type of activation function. When traditional fully connected networks with hyperbolic tangent activation functions are considered, the improvement in the generalization performance of the new models is even greater. The neural models exhibit superior generalization qualities that are virtually impossible to find by manual trial-and-error approaches.

Introduction

FEEDFORWARD, backpropagation neural networks are well known for their robust classification and approximation capabilities. Although they can be employed in a variety of applications, it is especially advantageous to utilize them in problems with intrinsically small and/or noisy data. The parameters or weights of neural networks can be adjusted offline by an array of well-understood first- or second-order optimization algorithms using training exemplars acquired beforehand.[1] A somewhat more complicated issue is how to choose the optimal neural topology for a given problem. In real-world scenarios, this question cannot be answered by a simple mathematical formula but through extensive experimental testing and/or utilization of sophisticated optimization techniques that attempt to discover the best performing networks.

Construction of accurate neural models that generalize well is neither a fast nor a trivial task. This observation is especially valid for problems where traditional regression models were utilized, revised, and carefully tuned for many years. Simple replacement of those models with standard feedforward architecture equipped with one or two hidden layers and sufficient nonlinear units may lead to unsatisfactory results. The reason for this is that the neural network models are not expected to perform nearly as well as the existing solutions. To be considered a serious alternative in industrial applications, they have to function significantly better. Otherwise, there is no justification for undertaking technological and financial risks associated with implementing alternative modeling techniques.

A typical requirement in function approximation tasks is to obtain a mathematical description of the relationship between regressors with a deviation from the actual observations not exceeding a certain threshold and the model response surface being maximally flat between training points. When an actual distribution of the input vectors is encompassed, this requirement may be specified more formally, in terms of generalization error, that is, an expected error of a given model for new, unseen inputs. Generalization error quantifies the effectiveness of the regression models, and as such, this value can be used to discriminate between various neural networks.

Because feedforward neural networks are nonlinear, complex mathematical models, their fine tuning can seldom be performed manually. An automatic approach to the structural optimization of neural networks is important because such a technique can ease and shorten the design process. In addition, in some situations, ensembles or mixtures of several well-fitted neural networks can to be utilized to further increase approximation accuracy over a given input domain. Automatic procedures are well suited for rapid creation of such models.

The ensemble should be built from the models that are accurate but different, that is, they have minimally correlated errors. It is interesting that a generalization performance estimator can be utilized not only in the search for the best neural networks but also to bind together several models. A weighted average of responses

$$y_{\text{ensemble}} = \frac{\sum_{i=1}^{R} \tilde{\theta}(\boldsymbol{x}_i) \big/ E_i^G}{\sum_{i=1}^{R} E_i^G} \tag{1}$$

Presented as Paper 00-1970 at the AIAA 38th Aerospace Sciences Meeting and Exhibit; received 20 November 1999; revision received 20 July 2000; accepted for publication 17 January 2002.

*Aerospace Engineer, New Technologies Engineering Department. Senior Member AIAA.
†Staff Engineer System LSI Design, Semi-Conductor Design Center.
‡Chief Scientist, Neuro-Engineering.
§Chief, Aerospace Operations Modeling. Associate Fellow AIAA.

combines expertise of individual networks. In Eq. (1), $\tilde{\theta}(\boldsymbol{x}_i)$ is the network response, E_i^G is an estimate of the generalization error of the ith neural network, and R is the number of models in the ensemble.

Problem Statement and Training Set Generation

Feedforward neural networks are used to model high-lift aerodynamics of a multi-element airfoil. An example of the three-element airfoil is shown in Fig. 1a. This multi-element airfoil was used in studies conducted by Greenman[2] and Greenman and Roth[3,4] to optimize the high-lift performance. The high-lift system of an aircraft is a crucial part of design because it influences takeoff and landing performance. The importance of a well-designed high-lift system is seen with increased payloads, which also increase operational flexibility by extending ranges and by decreasing takeoff and landing distances. Traditionally, high-lift designs have been produced by extensive wind-tunnel and flight programs, which are expensive and difficult due to the large design space. Recently, computational fluid dynamics (CFD) has been incorporated in high-lift design. For high-lift applications, CFD can also be expensive because the entire design space is large, grids must be generated around geometrically complicated high-lift devices, and complex phenomena must be resolved. To achieve optimum, rapid designs, neural networks are investigated as a tool for fast and efficient analysis of high-lift configurations.

The neural network models used flap riggings, that is, flap deflections, overlap, and gap (Fig. 1b), and angles of attack as inputs (four variables) and the lift coefficient as output. The training set is generated using a two-dimensional, incompressible Navier–Stokes solver. The entire training data set consists of 227 input/output pairs. Such a small data set is fairly typical in aerospace problems; generation of larger sets is often an expensive and time-consuming process. In some instances, it is virtually impossible to reproduce previous experiments to add more data as required.

Availability of a limited number of samples creates certain complications in computing and validating neural network models. The entire set of data points cannot be simply divided into the training and testing subsets because this can severely undermine the performance of the neural network. Moreover, when only a small data subset is used for model verification, a large optimistic bias in estimation is likely to occur. Another important issue is how to split the available data so the underlying distribution will not be seriously corrupted. Because of the small number of samples, all available data have to be utilized in the primary task of neural network training. The lack of an explicit verification set can be compensated to a certain extent by employing more sophisticated statistical techniques such as cross validation or bootstrapping (see "Generalization Error" section).

The geometry studied was a three-element airfoil, whose sections consist of a 12%c LB-546 slat[5] NACA 63_2-215 Mod B main element and a 30%c Fowler flap, where c is the chord of the cruise wing. The slat has a deflection of $\delta_s = 6.0$ deg, $\text{gap}_s = 2.0\%c$, and overlap of $\text{ol}_s = -0.05\%c$ (Fig. 1c). For the computational database, 51 different flap riggings are created for each slat deflection. The flap riggings are combinations of the flap deflection, gap, and overlap, which are defined in Fig. 1b. The flap deflection angle's range is $25.0 \le \delta_f \le 38.5$ deg. The gap setting's range is $1.5\%c \le \text{gap}_f \le 2.7\%c$, whereas the overlap setting's range is $0.4\%c \le \text{ol}_f \le 1.5\%c$. All gap and overlap values are expressed in terms of percent chord.

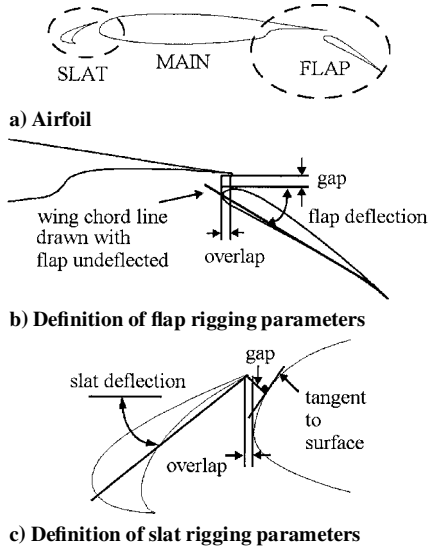

Fig. 1 Three-element airfoil.

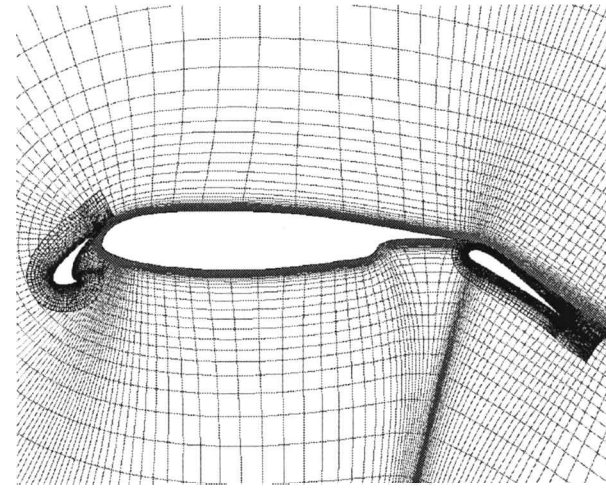

Fig. 2 Grid around three-element airfoil (every other point shown for clarity).

The grids around the three-element airfoil are generated by using OVERMAGG,[6] which is an automated script system used to perform overset multi-element airfoil grid generation. The rule-based high-lift grid generation techniques contained within OVERMAGG have evolved from the high-lift CFD applications by Rogers.[7] OVERMAGG takes as input the surface definition of the individual elements of the airfoil. It then creates a surface grid for each individual element by generating and redistributing points from the given surface definition. It calls the HYPGEN code[8] to generate volume grids about each element. OVERMAGG also automatically calls the PEGSUS code[9] to unite the individual meshes into an overset grid system, which is the final output of OVERMAGG.

Figure 2 shows the grid system. A total of 121,154 grid points is used, consisting of a 242×81 C grid around the slat, a 451×131 C grid around the main element, and a 351×121 embedded grid around the flap, which is used to help resolve the merging wake in this region. The normal wall spacing for all grids is 5×10^{-6} chords, and $Re_c = 3.7 \times 10^6$. Grid density studies, documented in Ref. 2, show that this system adequately captures the flow.

To generate the necessary computational training data, each high-lift configuration is analyzed at different angles of attack in the range $0.0 \le \alpha \le 12.0$ deg. The incompressible Navier–Stokes equations in two-dimensional generalized coordinates are solved using the INS2D-UP[10,11] flow solver. This code has been used extensively to predict multi-element airfoil flow.[2,5] The equations are solved using a generalized minimum residual implicit scheme. Because the flow is turbulent, the Spalart–Allmaras turbulence model[12] is used. This turbulence model has been successfully used to compute high-lift flowfields.[5,13] The flow was treated as fully turbulent for all elements in the computations. An empirically based criterion[14] designated the pressure difference rule, was applied to the training set because numerical inaccuracies for the computational method were identified near maximum lift. (For specific details, refer to Refs. 2–4.)

Optimization Framework

The procedure of tuning a neural network architecture is an optimization problem that typically involves multiple criteria. The most common requirement is that the network should accurately reproduce the training data. Also, it is important that the model should

generate adequate approximations for the input vectors that were not directly included in the training set. (The new input stimuli are assumed to be drawn from the same distribution as the learning set.) Moreover, the network is expected to have a compact architecture expressed, for example, in terms of the number of weights, hidden nodes, and/or input sensors. Smaller networks are important from the aesthetic as well as the economic point of view. (Complex solutions are often more expensive.) Structurally optimized networks are also less prone to learning spurious corrections present in the data, a feature that is particularly important in adaptive systems, in online learning, and in situations when the number of training data is small.

At least three major optimization objectives are considered, either explicitly or implicitly, in designing neural network models: training error, generalization performance, and network size or interconnection complexity. In many cases, the just iterated criteria are contradictory: Larger networks with more adjustable parameters are easier to train, but this occurs at the expense of generalization. Networks with inadequately small structure also exhibit deteriorated generalization performance. A well-documented relationship between generalization and training error suggests that excessive training may not benefit generalization. Commonly employed approaches developed for designing feedforward neural networks convert multicriteria optimization problems into the scalar minimization. This is the simplest solution to multi-objective problems. By the addition of two or more appropriately scaled objectives, a vector optimization is converted to the scalar (parametric) optimization task. If $E_1(\boldsymbol{w})$ and $E_2(\boldsymbol{w})$ denote training accuracy and network complexity measure, respectively, then

$$\min_{w} \Phi(\boldsymbol{w}), \qquad \Phi(\boldsymbol{w}) = (1-\alpha)E_1(\boldsymbol{w}) + \alpha E_2(\boldsymbol{w}) \tag{2}$$

where $\boldsymbol{w}$ is a vector of all network weights and biases. The cost function $\Phi(\boldsymbol{w})$ may be extended by adding additional components such that $\alpha_i \geq 0$ and $\sum_i \alpha_i = 1$.

For example, in Ref. 15, a differentiable estimation of generalization error is investigated.

Usually the L_2 norm or mean square error is used as a measure of learning accuracy $E_1(\boldsymbol{w})$. This is done under the assumption of normal distribution of noise present in the training data. A complexity measure such as weight decay [Eq. (3a)] or weight elimination [Eq. (3b)] can be used in conjunction with $E_1(\boldsymbol{w})$, that is,

$$E_2(\boldsymbol{w}) = \sum_i w_i^2 \tag{3a}$$

$$E_2(\boldsymbol{w}) = \sum_i \frac{w_i^2}{\beta + w_i^2} \tag{3b}$$

where $\beta > 0$ is a user-adjustable parameter. Although Eqs. (3) do not express the model size directly because a number of active weights $E_2(\boldsymbol{w})$ becomes smaller as $w_i \to 0$. Moreover, the function has the very desirable feature of being differentiable in respect to vector $\boldsymbol{w}$. This allows a wider range of efficient, gradient-based optimization methods to be applicable in searching for the optimal weight settings.

A disadvantage of the parametric approach is that α_i provides rather limited control of the relative ratio between various objectives. It is fairly difficult to adjust α_i, especially when more than two objective functions are involved and the values of each objective are not known precisely. The parametric approach also hides that, in multi-objective optimization, often not just one but many global optimal solutions may exist. This is possible due to the slightly different concept of optimum also known as Pareto optimality. The solution $\boldsymbol{w}$ is considered optimal in the Pareto sense when any improvement in one objective function is possible only at the expense or deterioration of at least one other criterion.

In the case presented here, a closed-form solution does not exist because the main optimization criterion, generalization error, is computed by the bootstrap method. As a matter of fact, only an estimation of generalization performance can be evaluated, and its value is contaminated by noise. To make the situation even more complex, the generalization error depends on the weight vector $\boldsymbol{w}$ as well as on interconnection geometry and node activation functions. This leads to a mixed continuous/discrete optimization problem because the main objective function depends on the vector,

$$\boldsymbol{x} = \lfloor w_1, w_2, \ldots, w_L, c_1, \ldots, c_L, f_1, \ldots, f_K \rfloor \tag{4}$$

in which w_i are continuous variables, L is the number of weights and biases in the unpruned network, c_i are binary codes that specify interconnection pattern, and f_i are discrete arguments that define hidden node activation functions, and K is the number of hidden nodes in the fully connected model.

Finding the optimal solution with respect to vector $\boldsymbol{x}$ is difficult because bootstrap generalization estimation absorbs substantial CPU resources due to multiple neural network retraining and recall sessions. In addition, the continuous/discrete nature of the problem requires that it is decomposed into subproblems, which would be solved separately by a multilevel hybrid optimization strategy and iteratively coordinated at a higher level to account for coupling between subsets of variables.[16]

To simplify the optimization procedure, it was decided to conduct the search only in respect to node activation functions $f_1, \ldots, f_K$ using genetic algorithms for predefined threshold values ε_j. This stochastic, zero-order (nongradient) method can handle noisy, multimodal response surfaces in discrete decision spaces. The interconnection geometry $c_1, \ldots, c_L$ is established by a combination of fast deterministic techniques: optimal brain surgeon (OBS) and singular value decomposition (SVD) methods that perform connection and node pruning.[17] Reference 17 also contains important modifications to the basic algorithm that result in faster and more robust execution. The weight settings $w_1, \ldots, w_L$ are inherited from the pruning phase. Note that the pruning is conducted until the training does not exceed the error level of the parent, fully connected network. The network is accepted by the genetic algorithm if at least 30% of its weights are removed and if the model could be successfully trained to the desired accuracy level. This approach accounts for the two other objectives, namely, the training error and the network complexity. The entire iterative optimization process is shown on the flow diagram in Fig. 3.

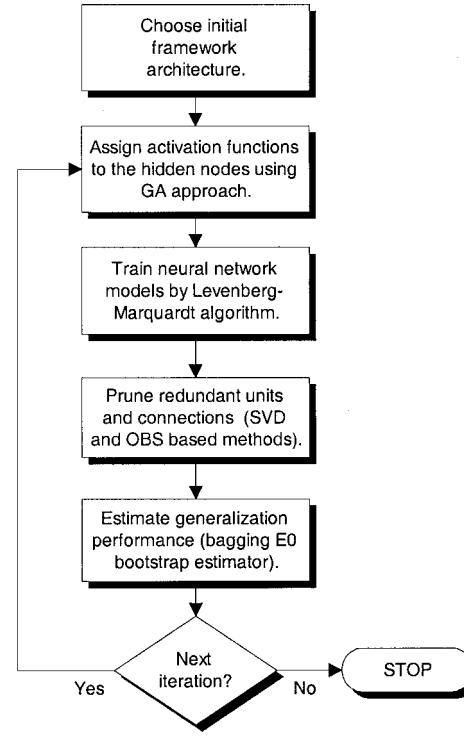

Fig. 3 Iterative process of selecting neural network architecture.

In this study, a genetic algorithm (GA) is also used in the optimization framework. A genetic algorithm is an evolutionary algorithm that generates each individual from some encoded form known as a chromosome or genome. The chromosomes are combined or mutated to breed new individuals. Crossover, the kind of recombination of chromosomes found in sexual reproduction in nature, is used in the GA. Here, an offspring's chromosome is created by joining segments chosen alternately from each of two parents' chromosomes, which are of fixed length. GAs are useful for multidimensional optimization problems in which the chromosome can encode the values for the different variables being optimized.

GAs differ from linear search methods: 1) They work with the coding of the set parameters instead of the parameters themselves. 2) They search a population of points instead of one single point. 3) They avoid auxiliary knowledge and using specific objective function information. 4) GAs use probabilistic rules instead of deterministic rules. Thus, GAs can search great amounts of data very efficiently.

Neural Network Architecture

In efforts to model the data of interest accurately, it is a good practice to visualize the training data as well as to test a few heuristic preprocessing transformations to learn how they affect the training process and the overall model performance. During initial investigations, it was found that simple standardization of input and target values was able to accelerate significantly the training procedure and to lead to lower training errors when compared to the appropriately rescaled error values obtained from the raw training set. The standardization causes the input and target vectors to have zero mean and standard deviation equal to one, that is,

$$x_i = (\hat{x}_i - \bar{x})/\sigma_x, \qquad t_i = (\hat{t}_i - \bar{t})/\sigma_t, \qquad i = 1, \ldots, N \quad (5)$$

where $\hat{x}_i$ and $\hat{t}_i$ are raw input and target training values, $\bar{x}$ and $\bar{t}$ are the means of the $\hat{x}$ and $\hat{t}$ variables, respectively, and σ_x and σ_t are the corresponding standard deviations.

Moreover, the preliminary tests indicated the presence of a strong linear trend between certain input/output variables. To reduce model bias caused by the apparent mismatch between the original process and the model structure, this linear trend can be either removed from the data set or the neural network architecture can be extended to accommodate the linear relationship explicitly. Fortunately, the linear model can be easily incorporated into the classical feedforward neural structure with linear output units by adding direct connections (weights) between network sensors and output nodes. An advantage of this approach is that most training or pruning methods are able to handle the new connections in the same manner as the weights between adjacent layers.

A technique of extending traditional, strictly layered feedforward neural architecture may be important when dealing with linear relationship between input and output data. Also, in many technical problems, models linear in parameters are frequently utilized. For example, it may be known that a suitable model has the following form:

$$y(x_1, x_2, x_3) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4\left(x_1 x_2 + x_3^2\right) \quad (6)$$

where $w_i (i = 0, \ldots, 4)$ are adjustable parameters. This nonlinear model (with linearly separable weights) can be combined with the feedforward neural network, as presented in Fig. 4. Thus, empirical knowledge about essential nonlinear components can be effectively utilized in the newly created neural architecture. Choosing appropriate preprocessing techniques and initial network structure are crucial aspects of almost every modeling procedure. Even when supported by a tedious series of experiments, the resulting fully connected neural networks are usually suboptimal because they tend to contain redundant connections. Many of those weights can be relatively efficiently identified and can be removed by pruning algorithms such as OBS, principal component analysis, and regularization or hybrid techniques. In this research, a much less explored impact of changing the configuration of activation functions associated with the hidden nodes is investigated. The common practice is to use sigmoidal functions, such as hyperbolic tangent or logistic sigmoid $f(a) = 1/[1 + \exp(-a)]$ for all of the hidden nodes. Although these functions produce robust neural network models, in certain applications it may be beneficial to employ other types of nonlinearities. A potential improvement in the network performance from modifying the set of activation functions may be comparable or even exceed the effects of connection pruning.

Alternative activation functions used in the feedforward neural network may have a well-known saturated S shape and the values bounded between -1 and 1 or 0 and 1. An infinite number of bipolar sigmoidal functions, with nontrivially different slope curvature can be generated, for example, by solving first-order differential equations,[18]

$$\frac{\mathrm{d}F(a)}{\mathrm{d}a} = \beta[1 - |F(a)|^2]^r \qquad \text{or} \qquad \frac{\mathrm{d}F(a)}{\mathrm{d}a} = \beta[1 - |F(a)|^s] \quad (7)$$

for different values of r and s (r and $s > 0$). The β parameter is adjusted in such a way that $F(a)$ is bounded between -1 and 1. Unfortunately, for arbitrarily selected r and s, the solution to Eqs. (7)

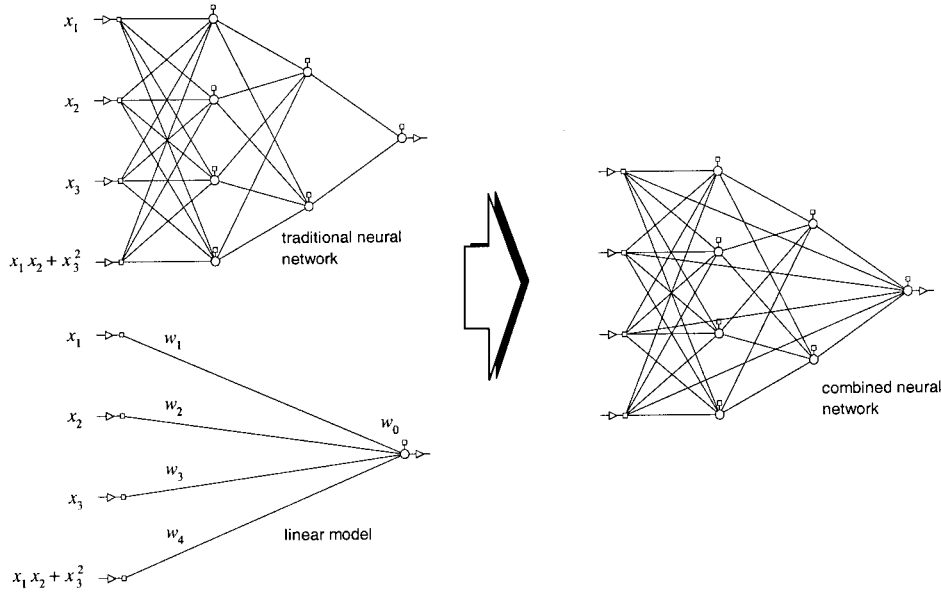

Fig. 4 Combined neural network architecture with linear model.

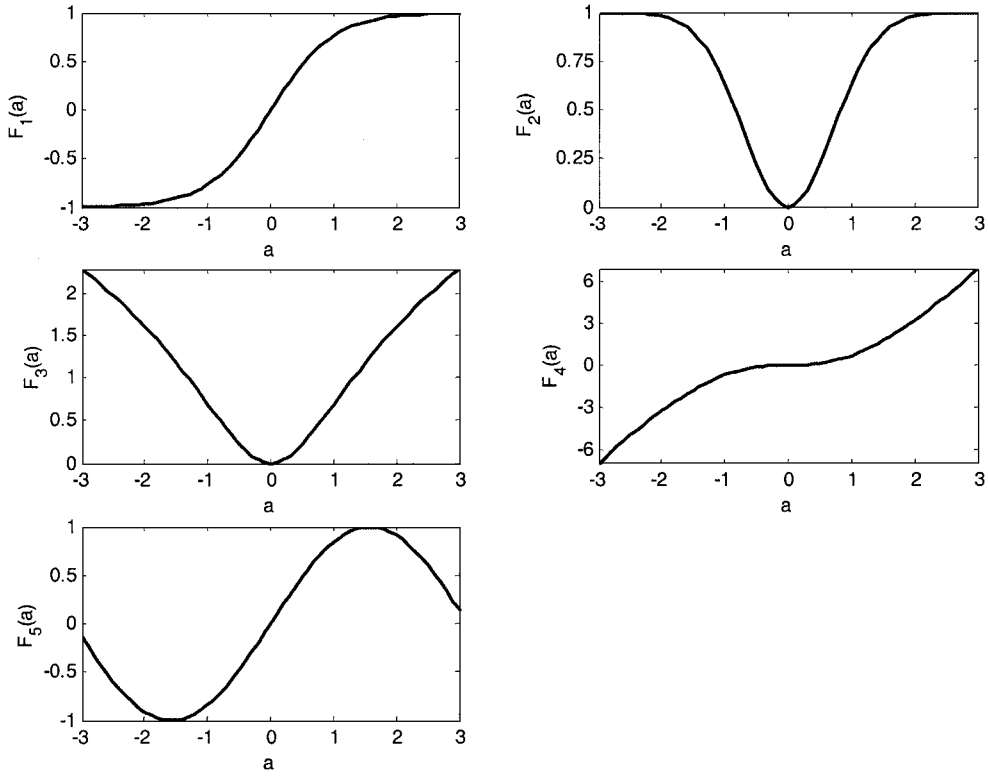

Fig. 5 Different activation functions used in creation of the modified neural models.

cannot be expressed by elementary functions typically implemented in computer hardware, but can be only approximated numerically. Precise functional approximations tend to slow down neural network training. This is a serious disadvantage for the methods that rely on multiple retraining sessions to estimate generalization performance.

Another alternative explored in these experiments is to use activation functions that are significantly different from a typical sigmoidal shape. Besides traditional hyperbolic tangent, $F_1(a) = \tanh(a)$, four other activation functions that have been successful in other studies were tested, that is,

$$F_2(a) = 1 - \exp(-a^2) \qquad F_3(a) = \log(1 + a^2)$$
$$F_4(a) = a\log(1 + a^2) \qquad F_5(a) = \sin(a) \tag{8}$$

Activation functions $F_1(a)$–$F_4(a)$ are presented in Fig. 5. Indeed, the additional functions are nonmonotonic or exhibit no saturation. (In the case of trigonometric sine, there is a periodicity feature as well.) When different activation functions are selected, a strategy of searching for an improved solution needs to be addressed. Even with a few hidden nodes and activation functions, the number of possible function configurations grows exponentially. For example, for a network with 14 hidden nodes and 5 different activations, the number of different function arrangements is $L = 5^{14} > 10^9$, an amount that clearly prohibits an exhaustive search.

Generalization Error

The generalization error E^G can be defined as the expected error of the regression model in response to new stimuli that were not utilized for in parameter evaluation (training). When the neural network output $\tilde{y} = \tilde{\theta}(\boldsymbol{x})$ approximates some noisy functional relationship, $y = \theta(\boldsymbol{x}) + \varepsilon$, the generalization error can be defined as

$$E^G = \int_{x \in \Omega} [\theta(\boldsymbol{x}) - \tilde{\theta}(\boldsymbol{x})]^2 p(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} + \sigma_\varepsilon^2 \tag{9}$$

where σ_ε^2 is noise variance. It is assumed that the input vectors $\boldsymbol{x}$ drawn from $\boldsymbol{\Omega}$ are independently and identically distributed according to some unknown probability density function $p(\boldsymbol{x})$. There are several statistical techniques to estimate E^G. Obviously, the simplest technique is a direct testing of a given model on a representative pool of M samples that were not used for training. For the mean square error, the prediction risk can be approximated by

$$E^G \approx \frac{1}{M}\sum_{i=1}^{M}[y_i - \tilde{\theta}(\boldsymbol{x}_i)]^2 \tag{10}$$

When the number of available data points is small in comparison to the model size and no observation vectors can be used exclusively for model verification, cross-validation or bootstrap techniques can be employed to assess neural network generalization ability. The performance of the original model, trained on the full data set, is estimated by subdividing or resampling the existing data, recomputing weights, and testing derived models. The new models have exactly the same architecture (identical activation functions and pattern of connections) as the original neural network but different weight settings due to changes in the training sets. Both cross-validation and bootstrap methods are computationally expensive, involving multiple retraining sessions. However, they do not require explicit knowledge about $p(\boldsymbol{x})$ to compute E^G approximation.

Cross-validation techniques are older than the bootstrap methods.[19] Practical experiments indicate that they tend to exhibit higher variance in comparison to the carefully chosen bootstrap algorithms, but this is not always true. Cross-validation methods are often used in assessing network generalization due to their simplicity and intuitive interpretation. In the K-fold version of cross validation, the existing data set is divided into K approximately equal segments. During the subsequent K iterations, one segment is removed from the original data set, then the network is retrained on the remaining $K - 1$ subsets and tested on the excluded samples. The procedure is repeated K times for every data segment, and an average test error is computed that estimates network generalization performance. The segments could be as small as one data point. In the latter case, a so-called hold-one-out cross-validation method is obtained.

Bootstrapping is another technique for estimating a desired statistics when the underlying distribution is not known.[20] In the bootstrap method, it is assumed that the information about $p(\boldsymbol{x})$ is adequately preserved in the original data set. The true statistics can be inferred

from the relationship between the original data set and pseudo-(bootstrap) subsets generated by sampling the original training data with replacement. The size of bootstrap sets is the same as the source set. Informally speaking, in the "bootstrap world" the entire network input domain is replaced with the only available data set, whereas bootstrap subsets imitate the multiple original sets that are not attainable in reality. Estimations obtained in the bootstrap world are then projected into the real problem domain.

Several modifications of the bootstrap techniques exist to compute the generalization or prediction error. The simplest and not recommended version of the bootstrap prediction error estimator trains B neural models using B bootstrap subsets and then tests them on the original data set. An estimation of generalization or prediction error is defined as an average,

$$E^G = \frac{1}{B}\sum_{i=1}^{B} E_i^0 \tag{11}$$

where E_i^0 is the error of the ith bootstrap model tested on the original data. A refined modification of the bootstrap procedure does not estimate the generalization error directly but instead assesses bias ω_B between the error of the original neural model (so-called apparent error) and unknown value E^G. An estimation of the generalization error can be formally expressed as

$$E^G = E_0^0 + \omega_B \tag{12}$$

where E_0^0 is an error of the original model tested on the original data set. An expected difference between the apparent error E_0^0 and the generalization error is estimated from the bootstrap models:

$$\omega_B = \frac{1}{B}\sum_{i=1}^{B}\left(E_i^0 - E_i^*\right) \tag{13}$$

where E_i^* is defined as an error of the ith bootstrap model evaluated on the ith pseudoset and E_i^0 is the error of the same model computed on the full data set. Equation (13) expresses how much, on average, the training error of the bootstrap model underestimates the error computed for the entire data set. This quantity is then projected onto the relationship between the apparent error E_0^0 and the generalization error E^G yielding an estimate

$$E^G = E_0^0 + \frac{1}{B}\sum_{i=1}^{B}\left(E_i^0 - E_i^*\right) \tag{14}$$

Formulas (14) are sometimes presented in more compact forms that conceal an intuitive meaning of the update shown in Eq. (13) but avoid evaluating network responses for data points shared between the original and bootstrap sets.

The third option for obtaining E^G computes the generalization estimate directly from the data points that were not included in the bootstrap training sets. This is the method that is used in the current study. A combined error is evaluated over all B bootstrap models by summing those error components that are associated with data points not occurring in the corresponding bootstrap subsets. The aggregate error is then divided by the total number of components encompassed in the calculations. The new estimator E^G can be written as

$$E^G = \frac{\sum_{b=1}^{B}\sum_{\{i:\, z_i \in D \wedge z_i \notin D_b^*\}} e_{b,i}^2(z_i)}{\sum_{b=1}^{B} \#\left\{i : z_i \in D \wedge z_i \notin D_b^*\right\}} \tag{15}$$

where $D = \{\boldsymbol{z}_i \equiv (\boldsymbol{x}_i, y_i);\ i = 1 \ldots N\}$ is the original set of observations and D^* is a bootstrap subset derived from D. Equation (15) defines the Efron $E0$ estimator.[20]

The $E0$ estimator typically provides conservative assessment of the generalization error.[19] It was found that the variance of Eq. (15) may not be necessary lower in comparison to cross-validation methods. For a large number of samples, N, the bootstrap method would tend, on average, to leave out more than 35% of the data points. The probability that any given point will not be included in a single pseudosubset is

$$P\left(\boldsymbol{z}_i \notin D^*\right) = \left(1 - \frac{1}{N}\right)^N = \frac{1}{[1 + 1/(N+1)]^N} \xrightarrow[N\to\infty]{} \frac{1}{e} \approx 0.3679 \tag{16}$$

For smaller N, the relative size of the testing set would oscillate around $1/e$, for example, 0.3682 ± 0.0205 for the simulating sampling with replacement from $N = 227$ data points. When the number of data points is decreased, as much as 40% or even more of available observations could be excluded from the pseudoset. The removal of so many training vectors from an originally small data set may have profound impact on the network response surface. As a result, instead of lower estimation variance, actually higher estimation variance can be observed. In this study, it was found that the simple cross-validation procedure, which puts aside about 15% of the randomly selected data points and retrains the network on the remaining pool of data, can reduce the standard deviation of the prediction risk estimator.

Another way to increase stability of the bootstrap generalization estimator is to employ a bagging strategy.[21] Bagging is a simple statistical technique that forms multiple versions of the same estimator and then averages them. Bagging improves accuracy and stability of the estimator when the procedure used to compute a single estimator, for example, network training tends to be erratic. A simple proof of this important property is provided in Ref. 21.

A crucial but rather neglected detail of the cross-validation or bootstrap methods applied to neural network verification is an implementation of the retraining procedure. Should the test networks be trained from randomly selected points or the weights of the original model be used to initiate retraining? When the weights of the original model are used to start training, very few iterations can often reduce the error to the previous level. As a result, the new weights may be very similar to the original model, which was trained on the full data set. Occasionally, the removal of several data points can by itself lower the learning error so that retraining would be unnecessary. Obviously, this kind of scenario will cause serious underestimation of the generalization error because the influence of the excluded data points is not fully eliminated. Unfortunately, the usage of the randomly chosen starting point may bring other types of complications. Highly optimized neural networks with no redundant connections may be difficult to retrain from randomly selected initial states. It is well known that the retraining performed by gradient-based procedures on nonlinear models have a tendency to stagnate in local minima or saddle points. Thus, inaccurate weight settings obtained in the retraining phase would incorrectly indicate that smaller networks exhibit poorer generalization performance and relatively high estimation variance. Oversized networks that are easier to train would have discriminatory advantage over smaller models.

To avoid such deceptive results, in this approach the networks are retrained from the initial weights that were obtained by random perturbation of the original weight settings, that is,

$$\boldsymbol{w}^* = \boldsymbol{w} + \lambda \mathrm{diag}(\boldsymbol{r})\boldsymbol{w} \tag{17}$$

where $\lambda = 10$ is a scalar representing the level of perturbation and $\boldsymbol{r}$ is a random, uniformly distributed vector with elements between -1 and 1. The level of perturbation λ could be adjusted by the estimation procedure. The perturbation level is initially driven as high as $\lambda = 10$. If the retraining procedure fails to regain a desired error level several times in a row, the perturbation level is gradually decreased by an arbitrary value of $\sqrt{2}$. The lowest weight perturbation level is $\lambda = 0.1$. This simple tuning strategy is able to generate improved starting points for retraining. High disturbance levels are observed for oversized fully connected networks, whereas smaller deviations from the original weight settings are applied for the pruned neural models. Because the retraining conditions are chosen to be maximally demanding, the overall retraining time and CPU load increased. Nevertheless, the networks are able to better reveal their approximating capabilities.

Numerical Experiments

In the benchmark problem, high-lift aerodynamics modeling, the initial neural network has a 4–8–6–1 topology (two hidden layers of 8 and 6 nodes, respectively) with additional direct connections between network sensors and output (linear model). In the first experiment, the generalization performance of the neural models with a single type of activation function (F_1–F_5) is compared. The networks included in the comparison are both fully connected and pruned. The experiment was repeated five times using five different starting points for training. All of the networks were allowed to be trained using maximum 100 epochs of the Levenberg–Marquardt method (see Ref. 22); the training was terminated earlier if the mean square error error of 10^{-4} was reached. The generalization error was computed by the bagging version of the $E0$ Efron[20] estimator with 200 successful retraining iterations. Table 1 summarizes the results obtained. Table 1 confirms that the pruned networks most often exhibit better generalization properties than the fully connected counterparts. The table also shows that the hyperbolic tangent activation may not always be the best choice of nonlinearity for the neuroregression tasks. Actually, it was somewhat surprising not to see the tanh function (code F_1) as the best node activation in at least one experimental run. However, the pruned networks with the hyperbolic tangent were the smallest in terms of number of weights. The gain in generalization performance varies on average between two and five times.

In the second series of experiments, a GA was used to search for the best configuration of activation functions. In addition, the network was pruned using a combined OBS/SVD approach. The stochastic optimization is a very slow and CPU-intensive process. However, the results produced by an additional layer of optimization are rather encouraging. As shown in Table 2, neural networks with mixed activation functions can achieve further reduction in generalization error. The gains are comparable with those produced by the pruning algorithms. This rather impressive accomplishment may be very important for problems where neural networks strive to achieve the best possible performance without significantly increasing the model complexity. These gains are compared with the activation function tanh because it was used in Refs. 2–4, and it is commonly used.

The final neural architectures from Table 2, found by a genetic optimizer and the parent, fully connected model, are shown in Fig. 6. In Fig. 6, F_n is the final activation function at each node in the hidden layers. The pruned models do not seem to share a lot of common topological features. Undoubtedly, the most important feature is an existence of direct links between network output from the first and the last sensors in all pruned models.

Table 1 Generalization error comparison between pruned and unpruned architectures with single type of activation function[a]

Single activation	Unpruned model	Pruned model	Pruned architecture	Number of weights
Tanh (F_1)	1.782e−02	2.209e−03	4–5–3–1	38
c Gauss (F_2)	1.363e−02	**2.002e− 03**	4–5–4–1	41
Logsqr (F_3)	**8.956e− 03**	7.032e−03	4–5–6–1	61
Asymlog (F_4)	1.019e−02	6.505e−03	4–8–6–1	90
Sin (F_5)	1.686e−02	1.206e−02	4–7–6–1	77
Tanh	1.983e−02	6.551e−03	4–7–3–1	53
c Gauss	1.285e−02	5.311e−03	4–6–3–1	44
Logsqr	1.377e−02	**3.823e− 03**	4–6–4–1	51
Asymlog	**9.104e− 03**	5.251e−03	4–6–6–1	69
Sin	1.156e−02	4.533e−03	4–7–6–1	82
Tanh	1.476e−02	2.990e−03	4–6–3–1	39
c Gauss	1.430e−02	7.641e−03	4–6–6–1	67
Logsqr	1.419e−02	**2.762e− 03**	4–7–4–1	60
Asymlog	2.455e−02	4.388e−03	4–8–4–1	70
Sin	**1.196e− 02**	5.234e−03	4–7–5–1	77
Tanh	1.885e−02	5.194e−03	4–5–6–1	52
c Gauss	**9.595e− 03**	5.777e−03	4–6–6–1	71
Logsqr	1.256e−02	6.447e−03	4–6–5–1	70
Asymlog	1.221e−02	**1.143e− 03**	4–7–6–1	76
Sin	1.666e−02	5.195e−03	4–7–6–1	80
Tanh	1.664e−02	7.967e−03	4–4–3–1	40
c Gauss	**1.204e− 02**	7.638e−03	4–6–6–1	69
Logsqr	1.660e−02	**2.167e− 03**	4–7–4–1	64
Asymlog	1.520e−02	3.923e−03	4–7–6–1	74
Sin	1.301e−02	3.630e−03	4–7–6–1	88

[a]Bold face indicates the best result in each test run.

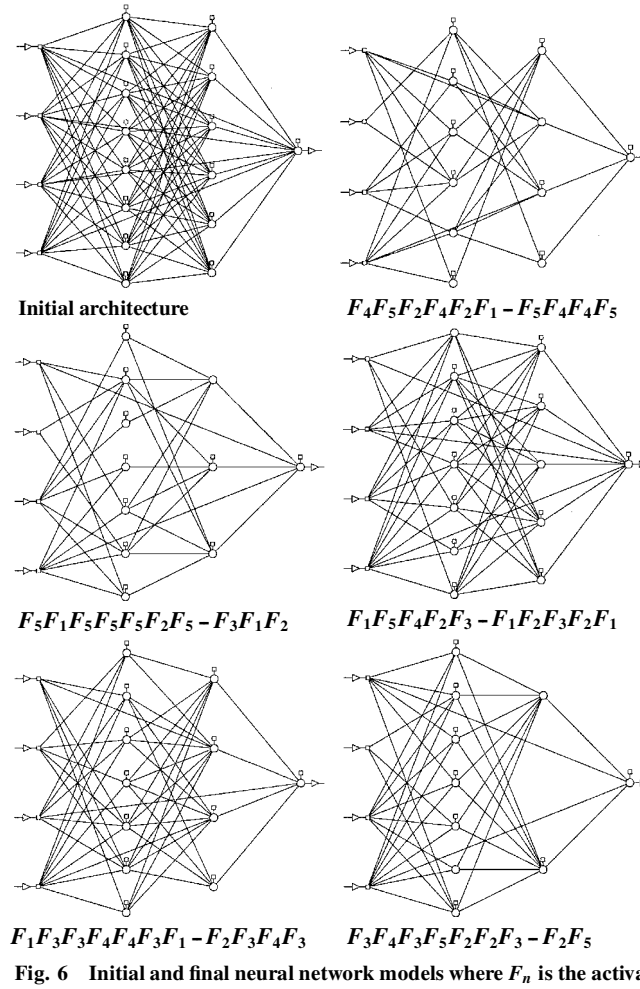

Fig. 6 Initial and final neural network models where F_n is the activation function at each node in the hidden layer.

Table 2 Performance results and architecture description of the final neural networks from Fig. 6

Architecture from Fig. 6	Best pruned architecture with tanh activation	Best pruned model with mixed activation functions	Improvement rate	Final architecture	Number of weights
Fig. 6b	2.209e−03	8.982e−04	2.46	4–7–3–1	45
Fig. 6c	6.551e−03	8.959e−04	7.31	4–7–4–1	63
Fig. 6d	2.990e−03	4.652e−04	6.43	4–6–4–1	44
Fig. 6e	5.194e−03	7.846e−04	6.62	4–7–5–1	67
Fig. 6f	7.967e−03	7.136e−04	11.16	4–7–2–1	48

Conclusions

This paper presents a new hybrid approach for structural optimization of the feedforward, backpropagation neural models. The method combines into one framework a bagging bootstrap technique, which estimates neural network generalization performance, and a stochastic, nongradient search strategy, such as a GA, which uses this estimate to select hidden node activation functions. A distinguished feature of our approach is that the design task is treated, in general, as a multi-objective optimization problem. This allows for a more direct treatment of competing design requirements. The new method is numerically tested by tuning neural networks that model high-lift aerodynamics of a multi-element airfoil under the realistic constrain of a small training data set. Despite the high computational costs of the bootstrap method and hindering variations of the generalization estimator, the method produces final solutions that exhibit on average five to six times smaller generalization error in comparison to the two-stage, scrupulously pruned models with uniform sets of activation functions. A comparison with fully connected networks shows an even bigger gain in favor of the new method. Although the retraining processes were designed to be as difficult as possible, the best models found by this algorithm exhibit remarkably small standard deviation of the generalization estimator. These results confirm that the solutions possess unique generalization qualities that are virtually impossible to find by a manual, trial-and-error approach.

In addition, the following observations were made: 1) For small data sets and demanding modeling problems, bagging versions of the bootstrap techniques should be utilized to reduce estimator variance. 2) The starting point in the retraining phase should be generated in such a way that the influence of the bootstrap testing points is maximally eliminated from the derived models, yet the network can still reach the desired error level.

References

[1]Battiti, R., "First- and Second-Order Methods for Learning: Between Steepest Descent and Newton's Method," *Neural Computation*, Vol. 4, No. 2, 1992, pp. 141–166.

[2]Greenman, R. M., "Two-Dimensional High-Lift Aerodynamic Optimization Using Neural Networks," Ph.D. Dissertation, Aeronautical and Astronautical Engineering Dept., Stanford Univ., Stanford, CA, May 1998, also NASA TM 112233, June 1998.

[3]Greenman, R. M., and Roth, K. R., "High-Lift Optimization Design Using Neural Networks on a Multi-Element Airfoil," *Journal of Fluids Engineering*, Vol. 121, No. 2, 1999, pp. 434–440.

[4]Greenman, R. M., and Roth, K. R., "Minimizing Computational Data Requirements for Multi-Element Airfoils Using Neural Networks," *Journal of Aircraft*, Vol. 36, No. 5, 1999, pp. 777–784.

[5]Rogers, S. E., Menter, F., Durbin, P. A., and Mansour, N. N., "Comparison of Turbulence Models in Computing Multi-Element Airfoil Flows," AIAA Paper 94-0291, Jan. 1994.

[6]Rogers, S., "Manual for the OVERMAGG Script System," NASA Ames Research Center, July 1997.

[7]Rogers, S., "Progress in High-Lift Aerodynamic Calculations," AIAA Paper 93-0194, Jan. 1993.

[8]Chan, W. M., Chui, I. T., and Buning, P. G., "User's Manual for the HYPGEN Hyperbolic Grid Generator and the HGUI Graphical User Interface," NASA TM 108791, 1993.

[9]Suhs, N. E., and Tramel, R. W., "PEGSUS 4.0 User's Manual," Arnold Engineering Development Center, AEDC-TR-91-8, 1991.

[10]Rogers, S. E., and Kwak, D., "An Upwind Differencing Scheme for the Steady State Incompressible Navier–Stokes Equations," *Journal of Applied Numerical Mathematics*, Vol. 8, Aug. 1991, pp. 43–64.

[11]Rogers, S. E., and Kwak, D., "Upwind Differencing Scheme for the Time-Accurate Incompressible Navier–Stokes Equations," *AIAA Journal*, Vol. 28, No. 2, 1990, pp. 253–262.

[12]Spalart, P. R., and Allmaras, S. R., "One-Equation Turbulence Model for Aerodynamic Flows," AIAA Paper 92-0439, Jan. 1992.

[13]Dominik, C., "Application of the Incompressible Navier–Stokes Equations to High-Lift Flows," AIAA Paper 94-1872, June 1994.

[14]Valarezo, W. O., and Chin, V. D., "Method of Prediction of Wing Maximum Lift," *Journal of Aircraft*, Vol. 31, No. 1, 1994, pp. 103–109.

[15]Pedersen, M. W., Hansen, L. K., and Larsen, J., "Pruning with Generalization Based Weight Saliencies: γOBD, γOBS, Advances in Neural Information Processing Systems," *Proceedings of the 1995 Conference*, MIT Press, Cambridge, MA, 1996, pp. 521–527.

[16]Barthelemy, J., and Francois, M., "Engineering Applications of Heuristic Multilevel Optimization Methods," *Discretization Methods and Structural Optimization—Procedures and Applications; Proceedings of the GAMM Seminar*, Springer-Verlag, 1989, pp. 24–31.

[17]Stepniewski, S. W., and Jorgensen, C. C., "Toward a More Robust Pruning Procedure for MLP Networks," NASA TM 112225, April 1998.

[18]Bell, A. J., and Sejnowski, T. J., "An Information-Maximization Approach to Blind Separation and Blind Deconvolution," *Neural Computation*, Vol. 7, No. 6, 1995, pp. 1129–1159.

[19]Ueda, N., and Nakano, R., "Estimating Expected Error Rates of Neural Network Classifiers in Small Sample Situations: A Comparison of Cross-Validation and Bootstrap," Inst. of Electrical and Electronics Engineers, IEEE Paper 0-7803-2768-3, 1995.

[20]Efron, B., and Tibshirani, R. J., *An Introduction to the Bootstrap*, Chapman and Hall, New York, 1993.

[21]Breiman, L., "Bagging Predictors," *Machine Learning*, Vol. 24, No. 2, 1996, pp. 123–140.

[22]Fletcher, R., *Practical Methods of Optimization*, Wiley, New York, 1987.