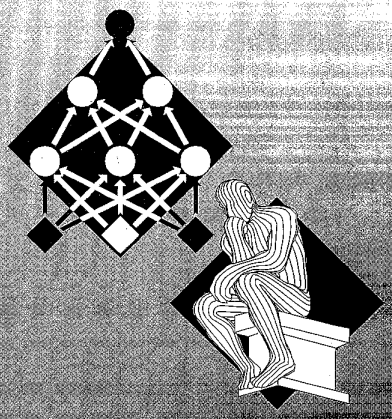

Neuro-Fuzzy Support for Problem-Solving Environments:

A Step Toward Automated Solution of PDEs

Anupam Joshi, Sanjiva Weerawarana, Narendran Ramakrishnan, Elias N. Houstis, and John R. Rice
Purdue University

In the "problem-solving environments" that will one day free scientists and engineers from many algorithmic and computing details, computational intelligence techniques such as neural networks and fuzzy logic can help automate choice of the best solution methods. Classifying the problems illustrated here using partial differential equations as an example is a first step.

HIGH-PERFORMANCE COMPUTING SYSTEMS LAG FAR BEHIND THEIR workstation and PC cousins in ease of use. In fact, Dianne O'Leary goes so far as to compare parallel computing today to the "prehistory" of computing, where computers were used by a select few who understood the details of the architecture and operating system, where programming was complex, and debugging required reading hexadecimal dumps.[1] Computer time had to be reserved, jobs were submitted in batches, and crashes were common. Users were never sure whether an error was due to a bug in their code or in the system. Most users of HPC find themselves in this situation today. If the HPC-based computational paradigm for the scientific process is to succeed and become ubiquitous, it must provide the simplicity of access that became popular with the advent of point-and-click capability in PCs.

An important recent advance in this direction is the development of *problem-solving environments*. A PSE is a computer system that provides the user with a high-level abstraction of the complexity of the underlying computational facilities. It provides all the computational tools necessary to solve a target class of problems.[2] These features include advanced solution methods, automatic or semiautomatic selection of solution methods, and ways to easily incorporate new solution methods. Moreover, good PSEs use the vernacular of the target class of problems and provide a "natural" interface, so people can use them without specialized knowledge of the underlying computer hardware or software. This is important since one cannot expect every user to be well versed in selecting the appropriate numerical, symbolic, and parallel systems, along with their associated parameters, that are needed to solve a problem. Though PSEs are still in an early stage of development, high-performance computers combined with better algorithms and better understanding of computational science have put them within our reach.

A PSE should be able to accept a user's high-level description of a problem, and then automatically select the appropriate hardware and software resources needed to solve it. Thus the problem-solving environment must use "intelligent" techniques incorporating knowledge about the problem domain and reasoning strategies. Such knowledge is built into *Pythia*, an intelligent as-

1070-9924/96/$5.00 © 1996 IEEE

sistant we have developed at Purdue University that will serve as a part of various PSEs. (The name derives from Greek mythology. People would put questions to the oracle at Delphi through a priestess called the Pythia, a name which perhaps unsurprisingly has now been adopted for several computer programs, including a Web browser and a particle physics application completely unrelated to what we are discussing here.) The methodologies used in Pythia are general and could assist in a wide range of scientific computing applications. Here we will show how Pythia can help in a specific task: selecting solution techniques for partial differential equations.

Pythia: An intelligent assistant

Pythia attempts to determine an optimal strategy for solving a given problem within user-specified requirements for resources and accuracy. By "strategy" here we mean a solution method and its parameters; resource requirements refer to limits on such things as execution time and, indirectly, memory usage. While the techniques Pythia uses are general, our current specific implementation of it operates in conjunction with Parallel Ellpack (often written *//Ellpack*), a system for solving elliptic partial differential equations.[3] Other efforts in the works will incorporate Pythia into the PDELab and SciAgents problem-solving environments. In the rest of this article, whenever we refer to a "problem" in the context of implementation and testing, we mean a PDE problem.

Pythia accepts as input the description of a problem, and determines the method or methods appropriate to solve it. (Examples of such methods are the five-point star and the seven-point star.) Its strategy is similar to that believed to underlie human problem-solving skills. A wealth of evidence from psychology suggests that people compare new problems to ones they have seen before, using some metric of similarity to make that judgment. They use the experience gained in solving "similar" previous problems to devise a strategy for solving the present one. This strategy has been termed *case-based reasoning* in the AI literature. In effect, Pythia compares a given problem to ones it has seen before, and then uses its knowledge about how certain solution methods performed on the old problems to choose a method for the new one and estimate how well it will perform.

Thus, to recommend a strategy to solve a given PDE problem p, Pythia needs

- a database P of previously solved problems along with data on the effectiveness of various solution methods on those problems,
- a mechanism to identify the problems from the database that are similar to p, and
- comparative data on the effectiveness of various methods on the problems in the database.

With this information, Pythia selects the solution method by using an algorithm of the following general form:

(1) Analyze the PDE problem and identify its characteristics. This involves applying symbolic analysis to extract some characteristics and asking the user about characteristics that cannot be determined automatically.
(2) Identify the problem $q \in P$ whose characteristics most closely match that of the new problem p.
(3) Use the performance data for q to predict the best method to use for p and the values for appropriate parameters (such as the size of the grid or mesh in the computation) to achieve the specified computational and performance objectives.

This strategy is somewhat naive, however. As the size of the database P increases, comparing the new problem to all those in P takes more and more time. An alternate strategy splits the previously seen problems into classes. Rather than search all the previous problems to find the most similar one, we confine the search to the ones that are in the same class as the new problem. Thus the strategy becomes

(1) Analyze the PDE problem and identify its characteristics.
(2) Identify the set $C \subset P$, where C is the class of problems whose characteristics are similar to those of p.
(3) Identify the problem $q \in C$ whose characteristics most closely match those of p.
(4) Analyze the performance data for the problems in the class C and rank the applicable methods to select the "best" method for solving the problem within the given computational and performance objectives. Then use the performance data available for the specific problem q to predict the values for appropriate computational parameters to achieve the specified objectives.

This revised strategy, further illustrated in Figure 1, is the one we have implemented in Pythia. The entire process is in place, though improving the way Pythia actually selects solution methods (Part 4 of the strategy) is a subject of our ongoing research. For the rest of this article we will focus on the some-

what more developed phase of classifying the problems prior to choosing solution methods. This is a task well-suited to the grouping and clustering abilities of various neural and neuro-fuzzy approaches.

Remember that while we are talking here about partial differential equations, the overall Pythia framework is general and could apply to other mathematical problems. The classification task is even more general and the methods by obvious extension could apply to arbitrary objects, including nonmathematical ones.

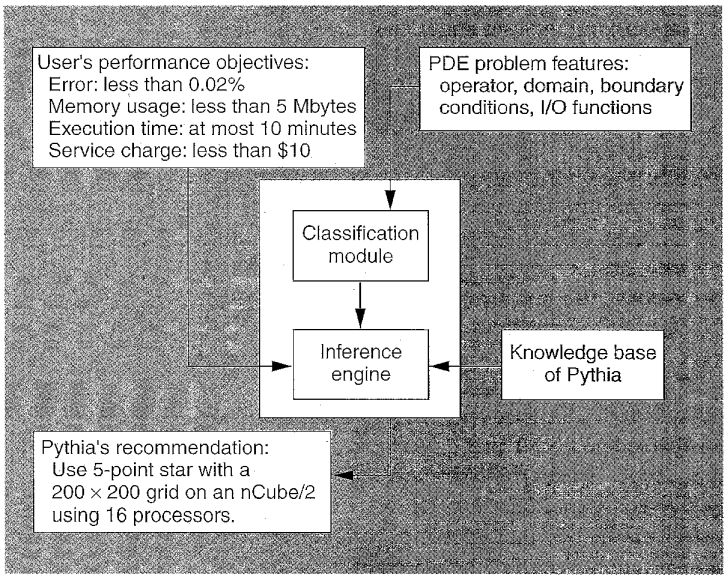

Figure 1. In the Pythia system, first the classification module uses characteristics of a PDE problem to assign it to one or more classes. Then the classification results, the user's performance objectives, and input from the Pythia knowledge base (about various PDE solvers and their ranges of applicability) are used to drive an inference engine that recommends a solution method.

Grouping problems into classes with Pythia

When grouping objects into classes, we must make some decisions about the criteria used for grouping. We need some idea about what characteristics to pay attention to, and what classes it makes sense to have.

Before classifying: Extracting a PDE's significant characteristics

When grouping problems into classes, what characteristics of a PDE are really significant? That is, what is Pythia looking for in Step 1 of the preceding algorithm? A PDE problem has characteristics of two main types: those of the problem components and those of the solution. We assume that the PDE problem is defined in terms of the following components: the PDE operator and right-hand side, the initial and boundary conditions, and the spatial and time domains of definition. The PDE characteristics include some yes-or-no classification information (such as whether the operator is homogeneous or not) and some quantitative information about the behavior—for example, smoothness and local variation—of the PDE functions (that is, coefficients of the operators, right-hand side of the operators, boundary and initial conditions, and the solution).

Characteristics we have selected to characterize a PDE problem and its solution in the Pythia system include

- Operator: Poisson, Laplace, Helmholtz, self-adjoint, homogeneous
- Boundary conditions: Dirichlet, Neumann, mixed, homogeneous
- PDE functions: smooth, oscillatory, wave front, singular, peak
- Solution: singular, analytic, oscillatory

Each characteristic is also associated with a value a, where with one exception $0 \le a \le 1$; that is, a is in the interval [0, 1]. Pure absence of a particular property is indicated by $a = 0$, and pure presence by $a = 1$. For logical characteristics (for example, whether the boundary conditions are Dirichlet or not), we use the values 0 and 1 for false and true, respectively. The set of characteristics of a PDE problem is represented as a *characteristic vector* V, which Pythia uses to identify a similar PDE problem or a class of related PDE problems from P. The PDE operator is described by 16 characteristics, which are derived from the bulleted list above; exactly how is not important to this discussion. To describe the whole problem a 32-vector is required. The 16-vector

(2,0,0,0,1,0,0,0,0,1,0,1,0,0,0.0,0.0)

represents the various type characteristics of the PDE operator of the problem specified in Figure 2, which is from a population of problems defined elsewhere.[4] The first entry in the vector represents the dimension of the problem, and is the exception to the [0, 1] range. The last two characteristics are continuous, rather than binary, values and so are represented decimally. In this study, we have assumed only linear PDE problems and solvers.

Currently, the Pythia user specifies the characteristics of a PDE problem manually through a graphical interface. In a system whose purpose is to simplify and automate problem solving this is less than ideal; automatic extraction of most characteristics is part of our ongoing research.

The whole business of extracting characteris-

tics raises another question: how do we *know* which things are important? That is, once we construct the vectors, will they be useful for grouping problems into classes that facilitate the choice of proper solution methods? The provisional answer is yes, but the only evidence here is empirical; characterizing PDE problems is notoriously hard. Even within our team, the AI experts believe they are working with the right characteristics mainly because the numerical analysts tell them so, based on experience. This issue touches on the very general and difficult subject of *representation and extraction of knowledge*, a whole field in itself. As part of our ongoing work, we are investigating the efficacy of various PDE characteristics.

Before classifying: Defining classes

The classes into which P should be grouped can be determined in two ways. First, we could apply one of several clustering mechanisms to distinguish clusters in the database, based on some arbitrary metric of similarity or difference in the problems' characteristic vectors, and call each cluster a class.

Second, domain experts can define some classes a priori based on the characteristics of the type of problem at hand—in our case, PDEs. This is the method we used. For some simplistic examples, one can precisely define a mapping of characteristics of the problem into a class. For most classes, however, such mappings cannot be defined in a simple analytic manner. Human experts who have worked with PDE problems for years can develop a feel for the problem types and, given the time, can map them into classes using a combination of factors—objective and subjective, exact and fuzzy. Given examples of the mapping, an artificial neural network (or any other function approximator) can "learn" the mapping function. This is the appeal of using ANNs in Pythia.

The success of our approach relies heavily on having available a reasonably large population of PDE problems whose characteristics span most of the space of all characteristic vectors. In the present implementation, we use as our training data a collection of 167 linear second-order elliptic PDEs defined elsewhere.[4] The database created using this problem population contains information about the properties of the problems plus performance data obtained by solving each equation with several different solution methods. Several different values of several computational parameters are used with each solution method on each problem. The database thus contains information on about 15,000 problem solutions.

To establish a mapping, human experts grouped the population of 167 problems into five classes manually. These are not necessarily the only reasonable classes, they are just the ones we chose for testing Pythia. We defined the following nonexclusive classes, with the number of problems belonging to each class given in brackets:

(1) *Solution-singular:* problems whose solutions have at least one singularity [6]
(2) *Solution-analytic:* problems whose solutions are analytic [35]
(3) *Solution-oscillatory:* problems whose solutions oscillate [34]
(4) *Solution-boundary-layer:* problems with a boundary layer in their solutions [32]
(5) *Boundary-conditions-mixed:* problems that have mixed boundary conditions [74]

Problem #28

$$(wu_x)_x + (wu_y)_y = 1, \text{ where } w = \begin{cases} \beta & \text{if } 0 \le x, y \le 1 \\ 1 & \text{otherwise} \end{cases}$$

Domain:	$[-1, 1] \times [-1, 1]$
Boundary condition:	$u = 0$
True:	Unknown
Operator:	Self-adjoint, discontinuous coefficients
Right side:	Constant
Properties of boundary conditions:	Dirichlet, homogeneous
Parameter:	β adjusts size of discontinuity in operator coefficients which introduces large, sharp jumps in solution.
Solution:	Approximate solutions given for $\beta = 1$, 10, 100. Strong wave fronts for $\beta \gg 1$.

Figure 2. A problem from the PDE population, showing some of the significant characteristics used to group equations into classes so that proper solution methods can be chosen.

Classifying PDEs: Methods

We have used several different methods, neural and non-neural, to identify the class or classes to which a *new* problem belongs. In describing the methods here we will omit details in the cause of brevity. Later, in a separate section, we describe experimental results of how well each method works.

In performing our experiments, we used one set of PDE data for training the neural networks (that is, in the modeling stage) and another to test how well the networks had learned—that is, whether they could correctly classify "new" PDEs. The entire Pythia data set consists of 167 ordered pairs $\{A_b, d_b\}$, where $A_b = (a_{b1}, a_{b2}, \ldots, a_{b32}) \in I^{32}$ is the input pattern (the 32-vector encoding each of the

167 PDE problems) and $d_b \in \{1, 2, \ldots, 5\}$ is the index of one of the five classes. The overall data set is split into two parts. The first contains 111 exemplars, approximately two-thirds of the total. The second part has 56 exemplars, representing the other third of the PDE population. Each neural network paradigm that we describe in this section was trained using each part; we refer to them as the larger and smaller training sets. After training, the learning of the paradigm was tested by applying it to the entire Pythia data set. The "training" problems in this set thus tested recall, and the "new" ones tested generalization.

Traditional method

We originally used a naive, non-neural heuristic to implement Pythia. This method represented a problem class as the centroid of all known exemplars of the class. The characteristic vector for a problem class was the average, computed element-by-element, of the characteristic vectors of all the members. That is, the ith element of the characteristic vector V of a class C was computed as

$$(V(C))_i = \frac{1}{|C|}\sum_{p \in C}(V(p))_i$$

where $|C|$ denotes the number of PDE problems in class C (which is a subset of the existing problems p in the whole database P). The distance from some *new problem* p to a defined class C is defined as the norm of the difference of the two characteristic vectors:

$$d(p,C) = \| V(p) - V(C) \|$$

The norm can be chosen as any reasonable distance measure. Then, we say that p belongs to class C if $d(p, C) < \sigma$, where σ is some neighborhood area around C that can be adjusted depending on the reliability of the characteristic vectors.

Feedforward neural nets: Gradient-descent algorithms

To perform more sophisticated mappings of new problems into the defined classes, we experimented with artificial neural networks. Each artificial "neuron" in an ANN is essentially a small computational processing element. The neuron models a function that computes the weighted sum of its inputs and "squashes" or compresses it by a nonlinearity, often a sigmoid function. Combining this neuron with other neurons creates a *layer* of nodes, which can be connected with other layers. The *multilayer perceptron* is a three-layer (or more) structure of artificial neurons: an input layer, an output layer, and one or more "hidden" layers. The neurons in the hidden layer are presumed to form internal representations of the input domain patterns.

To apply ANNs to Pythia's task, let us view the PDE classification problem as a mapping problem and suppose that we represent the m classes by a vector of size m (in our current example $m = 5$). Suppose a 1 in the ith position of the vector indicates membership in the ith class. Our problem now becomes one of mapping the characteristic vector of size n (in our case, the 32-vector) into the classification vector of size m.[5]

We can use a *feedforward* neural network trained using *backpropagation* to determine the mapping of n into m (see the sidebar). This network is a multilayer perceptron with *supervised learning*—we tell it when it is getting the right answers, or correct it when it isn't. Each neuron has a state s_i which is the weighted sum of all its inputs from other neurons it is connected to. The connection between the ith and jth neurons has a weight w_{ij}. (By convention neuron j is in the layer preceding neuron i—for instance, j might be in the input layer and i in the first hidden layer.) Some squashing function f—a function that maps a large input space into a small output space—is applied to the neuron's state to give its output o_i. Mathematically,

$$s_i = \sum_j w_{ij} o_j; \quad o_i = f(s_i) = \frac{1}{1+e^{-s_i}} \tag{1}$$

Using the backpropagation algorithm, the weights are then changed so as to reduce the difference between the desired and actual outputs of the network. The weight changes Δw_{ij} are given by

$$\Delta w_{ij} = \eta \delta_j o_i$$

where the weight-dependent error measure δ_j in the neuron's state, which we are trying to reduce by varying the weights, is defined by

$$\delta_j = \begin{cases} f_j'(net_j)(t_j - o_j) & \text{if unit } j \text{ is an output neuron} \\ f_j'(net_j)(\Sigma_k \delta_k w_{jk}) & \text{if unit } j \text{ is a hidden neuron} \end{cases}$$

In the above, t_j is the teaching input of unit j and net_j is the net input to unit j, f' denotes the derivative of f, and η is the "learning rate." This is essentially using gradient descent on the error surface with respect to the weight values. For more details, see the classic text by Rumelhart and McClelland.[6] Since the numbers of neurons in the input and output layers of the network, 32 and 5, are fixed by the problem, the only layer whose size has to be deter-

mined is the hidden layer. In our experiments we chose this to have 10 elements. Also, since we had no a priori information on how the various input characteristics affect the classification, we chose not to impose any structure on the connection patterns in the network. Our network was thus *fully connected*, that is, each element in one layer was connected to each element in the next layer. Thus there are only $32 \times 10 + 10 \times 5 = 370$ connections in the network, a relatively small number.

The second algorithm we consider for training a feedforward neural net modifies backpropagation by adding a fraction (the momentum parameter α) of the previous weight change during the computation of the new weight change.[7] This simple artifice helps moderate changes in the search direction, reducing the notorious oscillation problems common with gradient descent. To take care of "plateaus," a "flat-spot elimination constant" λ is added to the derivative of f. Typical values of the momentum parameter are in the range [0, 1]; the flat-spot elimination constant λ takes values from 0 to 0.25. The net effect of these enhancements is (1) flat spots of the error surface are traversed relatively fast with few big steps, (2) the step size decreases as the surface gets rougher, and (3) the search direction changes more slowly. This increases learning speed significantly.

A third training algorithm, Quickpropagation,[8] uses information about the curvature (via the second derivative) of the error surface to compute the weight change. QuickProp assumes the error surface to be locally quadratic and attempts to jump in one step from the current position directly into the minimum of the quadratic. This helps take care of "ridges" in the error surface. The important parameters here are the maximum growth parameter μ, which is the maximum amount of the weight change that is added to the current change, and the weight decay term ν, a factor to shrink the weights. We add ν to the slope S computed for each weight. This keeps the weights within an acceptable range and prevents problems like floating-point overflow errors during computations. Values of μ are usually between 1.75 and 2.25, and ν typically assumes low values like 0.0001 because QuickProp is very sensitive to it.

The final gradient-descent algorithm that we consider for training our feedforward ANN is called "resilient backpropagation" (RProp)[9] because it uses the local topology of the error surface to make a more appropriate weight change. In other words, we introduce a "personal update value" for each weight, which evolves during the learning process according to its local view of the error function. Thus we have two sets of learning equations, one for the weights and one for the update values themselves. RProp is very powerful and efficient because the size of the weight step taken is no longer influenced by the size of the partial derivative. It is uniquely determined by the sequence of the signs of the derivatives, which provides a reliable hint about the topology of the local error function. At the beginning of the training, all the update values are set to an initial value, say Δ_0. The choice of Δ_0 is not critical, because it is adapted as learning proceeds. However, we set an upper bound Δ_{max} on the update values, so that learning avoids any unreasonably high values of weight steps. In our experiments we set a default value of 0.1 to Δ_0, and varied Δ_{max} in the range [0.1, 25].

Feedforward Neural Networks and Backpropagation

It can be seen from Equation 1 in the main text that the outputs of the neurons in one layer (o_j) are multiplied by the weights w_{ij} and then squashed by the function f to calculate the outputs of the neurons in the next layer (o_i). This process is continued and the inputs are propagated through the network to achieve the final outputs. The input neurons normally contain the stimuli presented to the network and the output layer models the "target" being recognized. It can consist, for example, of a representation of the pattern classes in the domain as in the Pythia example, or an action to be taken on successful recognition. In the Pythia domain, the input neurons carry coded information about the PDE to be solved and the output neurons define a vector of membership of the given PDE in various classes. In order to achieve this "mapping" from input to output, the weights w_{ij} have to be determined. This process is termed "learning" and is achieved by presenting a table of input–output pairs to the ANN. The ANN uses this information and attempts to determine the weights w_{ij} that reflect the mapping inherent in the table.

Thus, the ANN has information about the input to the network and the output that must be generated for that input. It does not, however, have any information about the inputs and outputs for the hidden-layer neurons or the weights. Several algorithms have been proposed to automatically determine these weights. Initially, the weights are set to small random values. Then the ANN is presented with inputs and the outputs are calculated as in Equation 1. These outputs are compared with those present in the lookup table. The error is calculated as the difference between the desired and actual ouputs and is used as feedback to the network. The ANN then attempts to reduce this error by "backpropagating" the error from the output layer to the input layer while adjusting the weights. The process is repeated until the error falls below an acceptable threshold.

This kind of learning can be visualized as a search for a global minimum in a space defined by the weights. An effective technique for this purpose is gradient descent, which uses information on the slope of the surface to determine the direction in which the surface should be traversed to reach the global minimum. Backpropagation can be viewed as performing gradient descent on the space defined by the modifiable weights.

Learning vector quantization

Another type of neural network method, completely different from the backpropagation-trained, feedforward multilayer perceptrons discussed above, is *learning vector quantization*. LVQ borrows ideas from classical clustering and vector quantization techniques for signal processing, such as the k-nearest-neighbor algorithm. Signal values are approximated by quantized references or "codebook" vectors m_i which are "representative" members of a class. Several codebook vectors are assigned to each class in the domain, and a new pattern x is said to belong to the same class to which the nearest m_i belongs. LVQ determines effective values for the codebook vectors so that they define the optimal decision boundaries between classes, in the sense of Bayesian decision theory. The accuracy and time needed for learning depend on an appropriately chosen set of codebook vectors and the exact algorithm that modifies the codebook vectors. We used four different implementations of the LVQ algorithm: LVQ1, OLVQ1, LVQ2, and LVQ3. We used LVQ_PAK,[10] an LVQ program training package, in our simulations.

Let x be the input to the LVQ program and let m_c denote the codebook vector closest to x. Then the codebook vectors are updated according to the following simple rules, where t is an iteration step:

$$m_c(t+1) = \begin{cases} m_c(t) + \alpha(t)[x - m_c(t)] \\ \quad \text{if } x \text{ and } m_c \text{ are in the same class} \\ m_c(t) - \alpha(t)[x - m_c(t)] \\ \quad \text{if } x \text{ and } m_c \text{ are in different classes} \end{cases}$$

For all other codebook vectors, $m_i(t+1) = m_i(t)$. The control parameter α is not constant but varies with time. (This α is not to be confused with the momentum parameter α in the backpropagation training of a feedforward network). Normally, a linear decrease in time from a value of, say, 0.1 is used.

OLVQ1 (optimized LVQ1) is a modification of LVQ1 in which each codebook vector m_i has its own learning rate α_i. It has been shown that the optimal value of α can be recursively defined as

$$\alpha_c(t+1) = \begin{cases} \frac{\alpha_c(t)}{1+\alpha_c(t)} & \text{when } x \text{ is classified correctly} \\ \frac{\alpha_c(t)}{1-\alpha_c(t)} & \text{when it is not} \end{cases}$$

In practice, $\alpha_c(t)$ is allowed to increase steadily, but not above 1. Also, in LVQ_PAK, $\alpha_c(t)$ is never allowed to rise above its initial value.

The classification procedure in LVQ2 is similar to LVQ1, except that two codebook vectors m_i and m_j that are the nearest neighbors to x are now updated simultaneously. One of them is chosen to belong to the correct class and the other to a "wrong" class. Also, these two vectors are selected so that x falls into a "window" of values defined around the midplane of m_i and m_j. Thus LVQ2 *differentially* shifts the decision borders towards the Bayes limits. Then the equations for updating the codebook vectors become

$$m_i(t+1) = m_i(t) - \alpha(t)[x - m_i(t)] \text{ and}$$
$$m_j(t+1) = m_j(t) - \alpha(t)[x - m_j(t)]$$

where m_i and m_j are the two closest codebook vectors, x and m_j belong to the same class, and m_i and x belong to different classes.

One could argue that LVQ2 might update the "wrong" class vectors m_i too much so that the m_i vectors do not perform a good job of approximating the class distributions. LVQ3 introduces corrections that take care of this problem. In addition to the conditions mentioned for LVQ2, x should fall into the window defined by the vectors

$$m_k(t) + \varepsilon\alpha(t)[x - m_k(t)] \text{ and}$$
$$m_l(t) + \varepsilon\alpha(t)[x - m_l(t)]$$

where m_k, m_l and x belong to the same class. Typical values of the correction parameter ε, which controls the rate at which changes occur, are from 0.1 to 0.5. The optimal value of ε is found to decrease as the window size increases.

Neuro-fuzzy system

Neural networks and systems based on fuzzy logic have both been the subject of much investigation. Over the past few years, however, researchers have attempted to use synergies between these methods to create hybrid neuro-fuzzy systems. Examples include using neural networks to learn fuzzy membership functions, and using fuzzy logic to adapt the weights of neural networks. The system we describe next is essentially a fuzzy learning system, which can be elegantly represented as a three-layer feedforward neural network.

We have developed a new neuro-fuzzy classification scheme suited for our problem, or for any such problem in which the classes are not mutually exclusive, based on an algorithm proposed by Simpson.[11] Simpson's method uses fuzzy sets to describe pattern classes. Each class is defined by a fuzzy set, which is the fuzzy union of several n-dimensional *hyperboxes*. Each pattern within one of the hyperboxes in a set is a full member of the class defined by that set. (A hyperbox can be

thought of as an n-dimensional generalization of a box or rectangle. We explain the concept in more detail elsewhere.[12]) The hyperbox is completely defined by its min-point and max-point. It has associated with it a fuzzy membership function and an index denoting the class it corresponds to in pattern space. This function helps to view the hyperbox as a fuzzy set and such "hyperbox fuzzy sets" can be aggregated to form a single fuzzy-set class. Without any loss of generality, the pattern space is considered to be the n-dimensional unit hypercube I^n and the membership values are taken to be in the range [0, 1].

Learning in the fuzzy min-max network proceeds by placing and adjusting the hyperboxes in pattern space. The fuzzy set that describes a pattern class is then represented by an aggregate (the fuzzy union) of several fuzzy sets. Learning causes expansion or contraction of the hyperboxes. The learning algorithm operates by selecting an input pattern from the training set and finding a hyperbox for this pattern's class to include it. To control the formation of hyperboxes, the only parameter that must be tuned is the maximum hyperbox size θ beyond which it cannot expand. When this value is set to zero, the algorithm described above reverts to the k-nearest-neighbor classifier algorithm. Recall in the network consists of calculating the fuzzy union of the membership function values produced from each of the fuzzy-set hyperboxes.

Simpson's method assumes that the pattern classes underlying the domain are mutually exclusive and that each pattern belongs to exactly one class. But the pattern classes that characterize problems in many real-world domains are frequently *not* mutually exclusive. For example, some PDEs might have an analytic solution, some might have mixed boundary conditions, but some PDEs can both be analytic *and* have mixed boundary conditions.

The pseudocode for our algorithm[12] is shown in Figure 3.

Consider the kth ordered pair $\{A_k, d_k\}$ from the training set. Let the desired output for the kth pattern be [1, 1, 0, 0, ..., 0]. Our algorithm considers this as two ordered pairs containing the same pattern A_k but with two pattern classes as training outputs: d_{k1} = [1, 0, 0, 0, ..., 0] and d_{k2} = [0, 1, 0, 0, ..., 0] respectively. In other words, the pattern is associated with both Class 1 and Class 2. This will cause hyperboxes of both Classes 1 and 2 to completely contain the pattern A_k and thus overlap. We argue, unlike Simpson, that overlap between hyperboxes of different classes should not be eliminated when the problem domain demands it. However, if overlap was due to learning of different patterns of different classes, we eliminate the overlap.[12]

Since each pattern can belong to more than one class, we need to define a new way to interpret the output of the fuzzy min-max neural network. We introduce a parameter δ in the defuzzification step and set to 1 not only the node with the highest output but also the nodes whose outputs fall within δ of the highest output value. This results in more than one output node getting included and consequently, aids in the determination of nonexclusive classes. It also allows us to include "nearby" classes in our decision. Consider the scenario when a pattern gets associated with the wrong class, say Class 1, merely because of its proximity to members of Class 1 that were in the training samples rather than to members of its characteristic class (Class 2). Such a situation can be caused by a larger incidence of the Class 1 patterns in the training set, or due to a nonuniform sampling, since we make no prior assumption on the sampling distribution. In such a case, the δ parameter gives us the ability to make a soft decision by which we can associate a pattern with more than one class.

```
for each set of labeled data i from training set do
   for each class j that pattern i belongs to
        box = identifyexpandablebox();
        if (box = NOTAVAILABLE) addnewbox();
        else expandbox(box);
        flag = checkforoverlap();
        if (flag = true) contracthyperboxes();
```

Figure 3. Pseudocode for hyperbox algorithm.

Classifying PDEs: Results

We ran classification experiments on the Pythia problem domain using the several techniques we have just described. Each method was run numerous times, using a wide range of the parameters that control its operation. We report the results from only the "best" set of parameters.

For each of the techniques, the number of patterns from the test set (that is, all 167 PDEs) that the method classified correctly after its training period was determined as follows. We fixed a threshold for the L_2 error norm (ε) and inferred that patterns leading to error vectors with norms above the threshold were incorrectly classified. (The error vector is defined as the component-by-component difference between the desired output and the actual output.) We have carried out experiments using threshold values of 0.2, 0.1, 0.05, and 0.005 for each of the techniques.

Remember that the classes defined in Pythia are *not* mutually exclusive. Of the methods discussed earlier, only feedforward neural networks, and

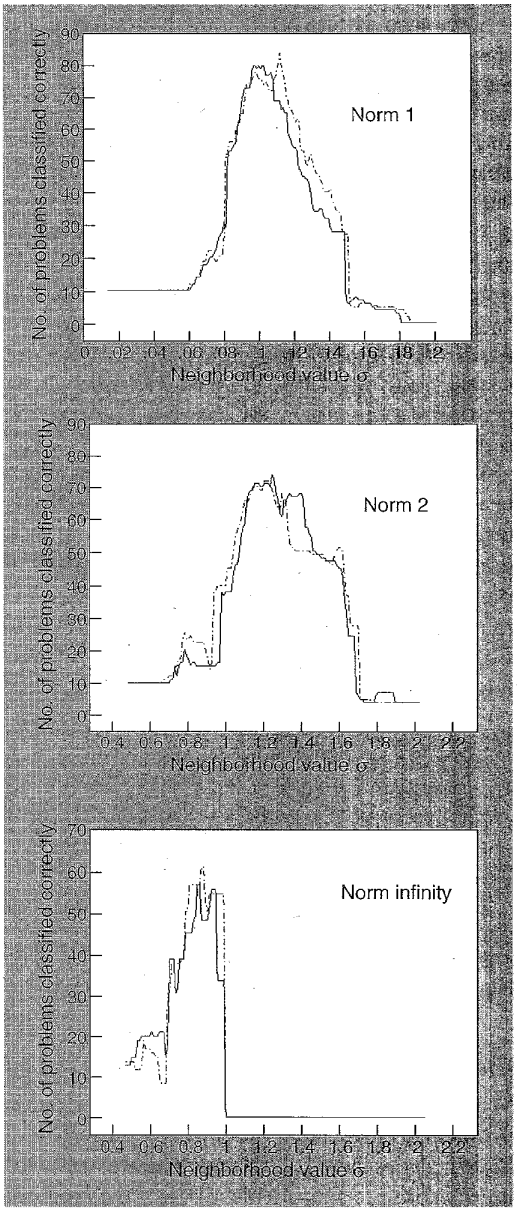

Figure 4. Performance of the traditional (nonneural) method in classifying 167 PDE problems, using norms L_1, L_2, and L_∞, as a function of the "neighborhood value" σ. Solid lines indicate performance after training on the larger training set, dashed lines the smaller.

Simpson's algorithm as modified by us, inherently cater to mutually nonexclusive classes. The other paradigms require the user to associate a single class with the problem characteristic vector at the time of training. Hence, in these paradigms, we transform the inputs as described earlier for our neuro-fuzzy method.

Traditional method

We showed before that the traditional method relies on the definition of an appropriate distance measure to quantify the distance of a problem p from a class C. We have used three different norms $\|\cdot\|$, namely the norms $\|\cdot\|_1$, $\|\cdot\|_2$, and $\|\cdot\|_\infty$. Each of these norms was used in conjunction with both the larger and smaller training sets. The neighborhood value σ was varied within an appropriate range since each norm has a different interpretation of the distance, and a single range of σ was not suitable for representing different metrics. For the norm $\|\cdot\|_1$, the range was [0.01, 0.2]. For the norms $\|\cdot\|_2$ and $\|\cdot\|_\infty$, the range was [0.5, 2]. Figure 4 displays the number of patterns classified correctly as a function of σ for each of the above three norms. Contrary to expectations, we observed that varying the L_2 threshold (ε) did not lead to a perceptible improvement or decline in the performance of the paradigm.

Of the three norms, $\|\cdot\|_1$ provides the best performance, though it seldom provides an absolute accuracy above 50 percent. Also, surprisingly, in some cases training with the smaller set leads to slightly better performance than the larger. A possible explanation is that the smaller set was more representative of the data in terms of the structure that the traditional method imposes on the distribution of the patterns. The larger training set, on the other hand, may have offered more inconsistencies.

Feedforward neural networks

The feedforward neural network was trained using all the aforementioned algorithms, with five choices of the control parameters. The choice leading to the best performance was considered for performance evaluation. All the simulations were performed using the Stuttgart Neural Network Simulator.[7]

The only "free" parameter in the simple backpropagation paradigm was the learning rate η, which was varied in the range [0.1, 0.9]. The best performance, in terms of classification accuracy, was achieved at $\eta = 0.9$. Increasing η also led to a decrease in convergence time.

In the enhanced variant of backpropagation with momentum, the important parameters are the learning rate η, the momentum coefficient α, and the flat-spot elimination constant λ. The value of η was kept low (0.2), because of the overpowering effect of the high momentum term which was found to be optimal at the values 0.7, 0.8, 0.9. The ideal value of the flat-spot elimination constant was found to be 0.05. The best performance was achieved at $(\eta, \alpha, \lambda) = (0.2, 0.8, 0.05)$.

QuickProp also assumed a low learning rate η. Also, the maximum growth parameter μ and the weight decay term ν strongly influence the performance of QuickProp. The ideal value of μ was in the range [1.75, 2] and that for ν was either 0.0001 or 0.0002. QuickProp had a very fast convergence rate; even though it got into lots of local minima problems, it was always able to come out

of them. Also, the maximum weight changes took place in the first 100 to 200 iterations and the subsequent iterations only served to fine-tune the error attained in these initial iterations. The best performance was achieved at values of 0.2 for η, 1.75 for μ, and 0.0001 for ν.

Of all the supervised paradigms for feedforward neural networks studied in this article, RProp provided the best performance for the same number of training iterations. We chose a fixed value of Δ_0 because the algorithm refines it iteratively and we set an upper bound 25 on the weight changes Δ_{max}. Even though some local minima problems were observed at high values of Δ_{max}, an extremely fast convergence rate served to make the network settle to a comfortable error level in about 100 iterations. The best performance was achieved at $(\Delta_0, \Delta_{max}) = (0.1, 25)$.

Figure 5 describes the behavior of the four methods for specific values of the L_2 error norm threshold. Observe that as the threshold value is decreased, the performance of backpropagation, enhanced backpropagation, and QuickProp methods decline while that of RProp is consistently high. RProp manages to correctly classify 160 of the 167 patterns. The accuracy of backpropagation, enhanced backpropagation, QuickProp, and RProp respectively for different values of the L_2 error threshold ε are:

(1) $\varepsilon = 0.005$: (47.3, 72.45, 74.25, and 95.83 %)
(2) $\varepsilon = 0.05$: (90.41, 93.41, 94.61, and 95.83 %)
(3) $\varepsilon = 0.1$: (92.81, 94.01, 94.61, and 95.83 %)
(4) $\varepsilon = 0.2$: (92.81, 94.01, 94.61, and 95.83 %)

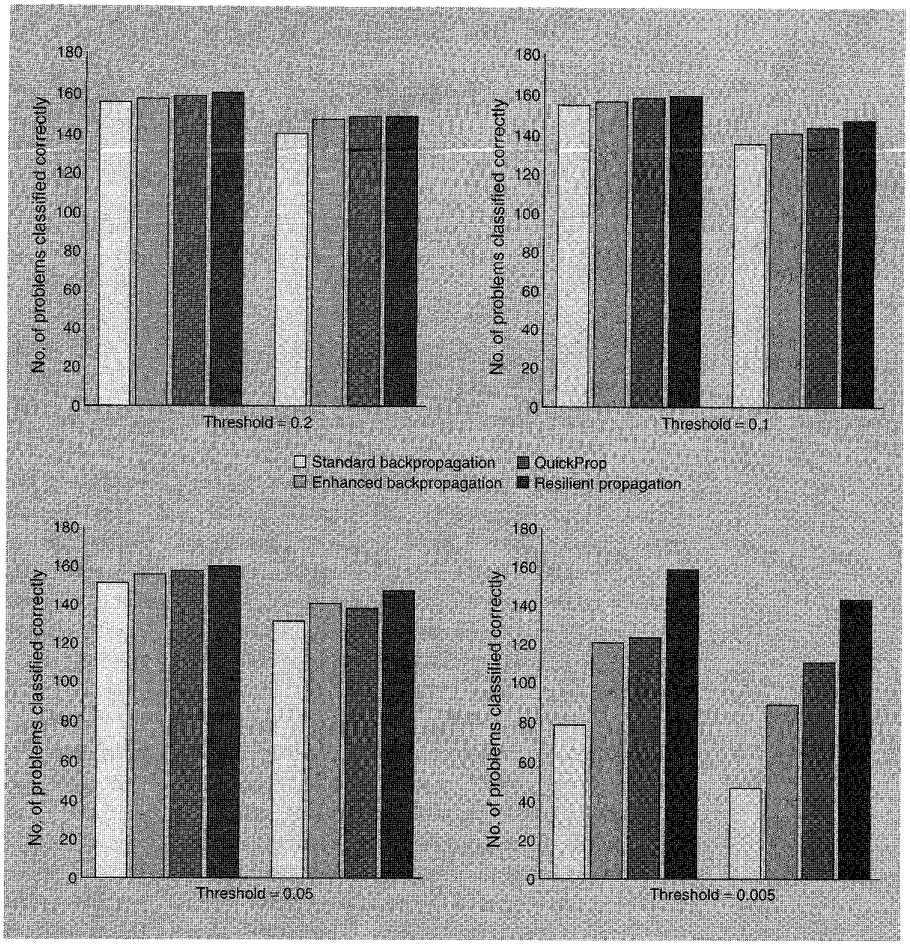

Figure 5. The performance of the feedforward neural network for various training schemes and four threshold values. Each network was trained for 2,000 iterations and then its "learning" was evaluated by testing to see how well it classified the 167 PDE problems. For each threshold value, results are shown for the larger training set (left four bars) and smaller training set (right four bars).

Learning vector quantization

Since the LVQ algorithms and the fuzzy min-max neural network work for labeled data of the form $\{A_h, d_h\}$, there is an implicit assumption that each pattern should belong to at least one class. However, in the problem domain, we may come across instances of PDEs that do not belong to any of the above defined classes. To circumvent this difficulty, we define a sixth class as follows:

(6) *Special:* problems whose solutions do not fall into any of the above-defined Classes 1 through 5. This artifice is employed in both the LVQ algorithms and the fuzzy min-max neural network.

The LVQ algorithms were trained as follows. Fifty codebook vectors were chosen so that their numbers in the respective classes were proportional to their a priori probabilities. Then the algorithms were trained for 2,000 iterations using both the larger and the smaller training sets. This number of iterations is adequate for convergence for both sets. We present here only the results with the larger training set.

The important free parameter in LVQ1 was the learning rate. This was varied from 0.1 to 1 in steps of 0.01. The accuracy achieved is plotted against the learning rate in Figure 6a.

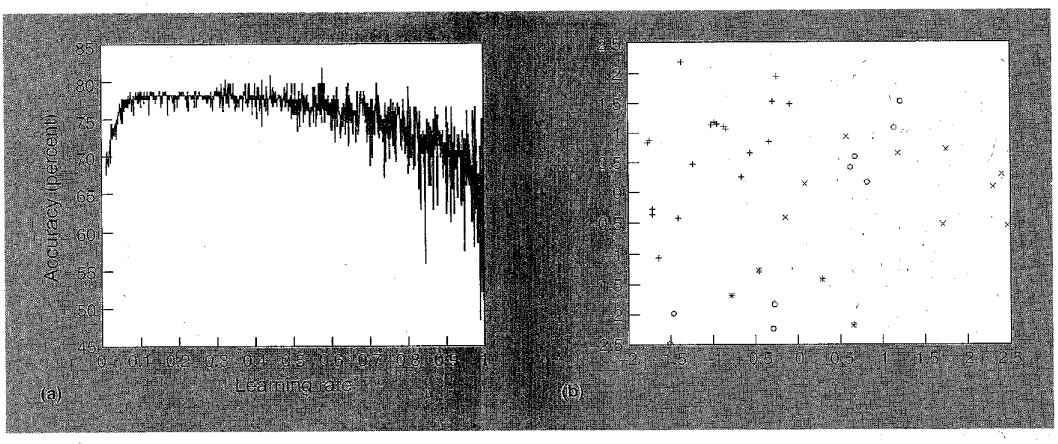

Figure 6. (a) Performance of the LVQ1 algorithm. (b) Clustering of the PDE problem classes. Each symbol represents a class of PDE problems.

The highest accuracy (77.06 percent) was attained at a learning rate of 0.05 (this was for an L_2 threshold of 0.005). LVQ1 is used to provide an "initial" solution and other LVQ algorithms can be used to improve the learning done by LVQ1. We adopted this strategy for our experiment.

OLVQ1 was subsequently trained for 200 iterations and the accuracy obtained was 80 percent (L_2 threshold value = 0.005). Thus OLVQ1 substantially fine-tunes the initial solution provided by LVQ1. In Figure 6b, we map the 32-dimensional data space of the codebook vectors onto the two-dimensional plane using Sammon's mapping. This two-dimensional mapping approximates to the Euclidean distances of the data space and thus visualizes the clustering of data. The LVQ2 algorithm depends on the window width parameter, that is, the relative width of the window into which the training data must fall. We varied the window width parameter from 0.1 to 0.5 and also the learning rate as mentioned in the LVQ1 experiment. The optimal performance was achieved at a window width of 0.3 and a learning rate of 0.2. However the accuracy for this implementation (for an L_2 error threshold of 0.005) was only 79.79 percent, a bit lower than that achieved by the OLVQ algorithm.

The LVQ3 algorithm can be used for an additional fine-tuning stage in learning. The relative learning rate parameter ε is used (multiplied by the parameter α), when both the nearest codebook vectors belong to the same class. Again, as in the LVQ2 experiment, the relative window width parameter determines the "box" into which the training data must fall. Again, a window size of 0.3 was used and the relative learning rate parameter ε was set at 0.1. We observed that though LVQ3 improves the initial codebook, it does not give results better than the OLVQ algorithm. For example, the maximum accuracy attained by LVQ3 was 79.26 percent (for an L_2 error threshold value of 0.005).

Fuzzy min-max neural networks

For the neuro-fuzzy method, we conducted experiments on the effect of the maximum hyperbox size θ, the effect of the parameter δ used in the defuzzification step, and on-line adaptation. Again, we present the results only with the larger training set.

Effect of maximum hyperbox size. In this set of experiments, the maximum hyperbox size was varied continuously and its effect on other variables was studied. In particular, we observed that when θ was increased, fewer hyperboxes needed to be formed, that is, when θ tends to 1, the number of hyperboxes formed is six, the number of classes in the domain. Also performance on the training set and the test set steadily improved as θ was decreased (Figure 7a). Performance on the training set was, as expected, better than that on the test set. Optimal error was achieved when θ equaled 0.00125. When θ was more than 0.00125, the error increased on both the sets and when θ was less than 0.00125, the network *overfit* the training data so that its performance on the test set started to decline. The number of hyperboxes formed for this optimal value of θ was 62, approximately double the dimension of the pattern space.

Effect of δ. In this experiment, we set $\theta = 0.00125$ (the optimal value) and we varied δ by assigning to it the values 0.01, 0.02, 0.05, and 0.09. When δ was increased, more output nodes tended to get included in the "reading-off" stage so that the overall error increased. Figure 7b shows a scatter plot of the results for $\delta = 0.01$.

On-line adaptation. The last series of experiments was conducted to test the fuzzy min-max neural network for its on-line adaptation. That is, each pattern was incrementally presented to the network and the error on both sets was recorded at each stage. The number of hyperboxes formed

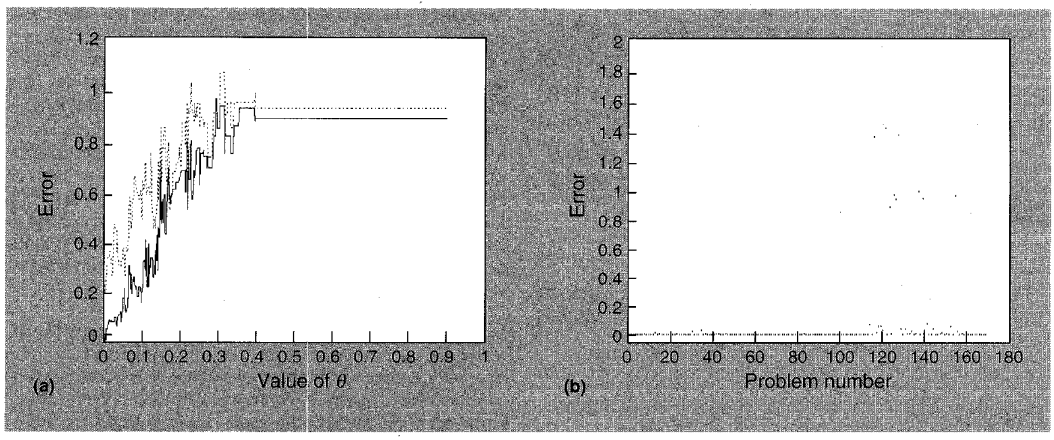

Figure 7.
(a) Effect of maximum hyperbox size θ on performance of the fuzzy min-max neural network. The solid line indicates the error on the training set while the dashed line indicates the error on the test set.
(b) Scatter plot of results for optimum θ and $\delta = 0.01$.

slowly increased from 1 to the optimal number 62 (as seen from the experiment on the effect of θ). Also, performance on both sets steadily improved to the values obtained in that experiment.

The accuracy obtained by the fuzzy min-max neural network was 95.21 percent. Varying the L_2 error threshold value ε did not alter the accuracy.

Discussion

The best mean and median values for errors from all the paradigms discussed in this article are summarized in Table 1. In each row of Table 1, the best possible method and the optimal combination of the parameters was used for the comparison.

The naive technique, which represents classes by the centroid of the known samples, performed very poorly (with an accuracy of 47.9 percent).

Feedforward neural networks, in general, performed quite well, with more complicated training schemes such as enhanced backpropagation, Quick Propagation, and Resilient Propagation clearly winning out over plain error backpropagation. For higher L_2 error threshold values (say 0.2), all these learning techniques gave values close to each other (92.81, 94.01, 94.61, and 95.83 percent respectively). However, when the L_2 error threshold levels were lowered (to 0.005), RProp was clearly better than all the other methods (with an accuracy of 95.83 percent over 47.3, 72.45, and 74.25 percent for plain backpropagation, enhanced backpropagation and Quick Propagation). The same observations can be made by looking at the mean and median of the error values in the table. While the mean for RProp (0.0446) is slightly lower than that of others, the median is significantly lower (0.000001). This means that RProp classifies most patterns correctly with almost zero error, but has a few outliers. The other methods have the errors spread more "evenly," which leads to a degradation in their performance as compared to RProp.

The variants of the LVQ method (LVQ1, OLVQ1, LVQ2, and LVQ3) that we tried performed about average. While they were better than the naive classifier (with an accuracy of 80 percent for OLVQ1), their performance was only in the 75- to 80-percent range (for an L_2 error threshold value of 0.005). Increasing the L_2 error threshold did not improve accuracy.

Finally, our neuro-fuzzy method, which is a variant of that proposed by Simpson,[11] performed quite well. In fact, it performed almost as well as RProp both in terms of percentage accuracy (95.20 percent), mean error (0.05534), and median error (0.000001). As with RProp, increasing the L_2 error threshold did not significantly alter the performance. Considering that unlike RProp, our method allows on-line adaptation (that is, new data do not require retraining on the old data), it is clearly superior in this context. This is because in the Pythia environment, we expect the system to constantly update its database with the new problems it has seen.

THE PYTHIA INTELLIGENT ASSISTANT, OR systems like it, is an important component of the problem-solving environments being developed to help computational scientists and engineers do their research. Further work on several neural aspects of Pythia is in progress. We are developing a method to directly map the original problem—actually selecting a solution method for a PDE given the user's time, grid, and error criteria—to a

Table 1. Comparison of the four methods in classifying the 167 PDE problems (error statistics).

Method	Description	Mean	Median
Traditional	Norm 1	0.618657	1.00000
Feedforward	RProp	0.044661	0.000001
LVQ	OLVQ1	0.155557	0.006125
Fuzzy min-max	—	0.055345	0.000001

neural network. Also, we are investigating extensions to predict when the problem would require a parallel machine, which machine, and what its configuration should be. Work is also in progress on improving our neuro-fuzzy scheme to use nonisothetic hyperboxes, and on exploiting the SIMD parallelism inherent in the neural network. ◆

Acknowledgments

This work was supported in part by NSF awards ASC 9404859 and CCR 9202536, AFOSR award F49620-92-J-0069 and ARPA ARO award DAAH04-94-G-0010.

References

1. D.P. O'Leary, "Parallel Computing: Emerging from a Time Warp," *IEEE Computational Science & Engineering*, Vol. 1, No. 4, Winter 1994, p. 1.
2. E. Gallopoulos, E. Houstis, and J.R. Rice, "Computer as Thinker/Doer: Problem-Solving Environments for Computational Science," *IEEE Computational Science & Engineering*, Vol. 1, No. 2, Summer 1994, pp. 11–23.
3. E.N. Houstis et al., "//ELLPACK: A Numerical Simulation Programming Environment for Parallel MIMD Machines," *Proc. 1990 ACM Int'l Conf. on Supercomputing* [Amsterdam], ACM Press, New York, 1990, pp. 96–107.
4. J.R. Rice, E.N. Houstis, and W.R. Dyksen, "A Population of Linear, Second Order, Elliptic Partial Differential Equations on Rectangular Domains, Part I," *Math. of Computation*, Vol. 36, 1981, pp. 475–484.
5. A. Joshi, S. Weerawarana, and E.N. Houstis, "The Use of Neural Networks to Support "Intelligent" Scientific Computing," *Proc. IEEE Int'l Conf. Neural Networks (ICNN '94)*, IEEE, Piscataway, N.J., 1994, Vol. 4, pp. 411–416.
6. D.E. Rumelhart, G.E. Hinton, and R.J. Williams, "Learning Internal Representations by Error Propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1, D.E. Rumelhart and J.L. McClelland, eds., MIT Press, Cambridge, Mass., 1986, pp. 318–364.
7. A. Zell et al., "SNNS: Stuttgart Neural Network Simulator," Tech. Report 3/93, Inst. for Parallel and Distributed High Performance Systems, Univ. of Stuttgart, Germany, 1993. Report and software available on World Wide Web at http://vasarely.informatik.uni-stuttgart.de/snns/snns.html.
8. S.E. Fahlman, "Faster-Learning Variations on Backpropagation: An Empirical Study," in *Proc. 1988 Connectionist Models Summer School*, T.J. Sejnowski, G.E. Hinton, and D.S. Touretzky, eds., Morgan Kaufmann, San Francisco, 1989, pp. 38–51.
9. H. Braun and M. Riedmiller, "Direct Adaptive Method for Faster Backpropagation Learning: The RProp Algorithm," *Proc. IEEE Int'l Conf. Neural Networks (ICNN '93)*, IEEE, Piscataway, N.J., 1993, pp. 586–591.
10. T. Kohonen et al., "LVQ-PAK: Learning Vector Quantization Program Package," tech. report, Laboratory of Computer and Information Science, Rakentajanaukio 2 C, SF-02150, Espoo, Finland, 1992. Software available on World Wide Web at http://nucleus.hut.fi/~hynde/lvq/.
11. P.K. Simpson, "Fuzzy Min-Max Neural Networks—Part 1: Classification," *IEEE Trans. Neural Networks*, Vol. 3, No. 5, 1992, pp. 776–786.
12. N. Ramakrishnan et al., "Neuro-Fuzzy Systems for Intelligent Scientific Computing," *Proc. Artificial Neural Networks in Engineering (ANNIE) '95*, ASME Press, New York, 1995, pp. 279–284.

Note: A guide to further reading on the subject of this article is on the World Wide Web at http://www.computer.org/pubs/cs&e/cs&e.htm.

Anupam Joshi is a visiting assistant professor of computer sciences at Purdue University. He received his B.Tech degree in electrical engineering from the Indian Institute of Technology, Delhi, in 1989, and his PhD in computer science from Purdue in 1993. His research interests span the broad area of artificial and computational intelligence. He has worked on neuro-fuzzy techniques, multiagent systems, and computer vision, and has done work in using AI/CI techniques to help create problem-solving environments for scientific computing. His other interests include mobile computing and computer-mediated learning. He is a member of IEEE, the IEEE Computer Society, ACM, and the computer science honorary society Upsilon Pi Epsilon.

Sanjiva Weerawarana is a visiting assistant professor of computer sciences at Purdue University. His research is centered around designing and building problem-solving environments for scientific computing with concentration on partial differential equation–based applications. He contributed to the design and implementation of the parallel Ellpack and PDELab PSEs and is involved with the SoftLab and SciencePad projects as well. His current research is focused on building software infrastructure for developing PSEs. Weerawarana received his BS and MS in applied mathematics/computer science from Kent State University and his PhD (1994) in computer science from Purdue. He is a member of the ACM and UPE.

Narendran Ramakrishnan is pursuing his PhD in computer sciences at Purdue University. He received an ME in computer science and engineering from the Anna University, Madras, India, and has worked in the areas of computational models for pattern recognition and prediction. His current research addresses the role of intelligence in problem-solving environments for scientific computing. He is a member of the IEEE Computer Society and ACM SIGART.

Elias N. Houstis is professor of computer science and director of the computational science and engineering program at Purdue University, where he received his PhD in 1974. His research interests include parallel computing, neural computing, and computational intelligence for scientific applications. He is currently working on the design of a problem-solving environment called PDELab for applications modeled by partial differential equations and implemented on a parallel virtual machine environment. He is a member of ACM and the International Federation for Information Processing (IFIP) Working Group 2.5 (Numerical Software).

John R. Rice is W. Brooks Fortune Professor of Computer Sciences at Purdue University, where he is establishing a graduate degree program in computational science and engineering. He received his PhD in mathematics from Caltech in 1959, joined the Purdue faculty in 1964, and has been head of the Department of Computer Sciences there since 1983. The author of several books on approximation theory, numerical analysis, computer science, and mathematical and scientific software, Rice founded the *ACM Transactions on Mathematical Software* in 1975 and remained as its editor until 1993. He is a member of the National Academy of Engineering, the IEEE Computer Society, ACM, IMACS, and SIAM. He serves as *IEEE CS&E*'s area editor for problem-solving environments.

Readers may contact the authors in care of Anupam Joshi, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, USA. E-mail addresses of the respective authors are {joshi,saw,ramakris,enh,jrr}@cs.purdue.edu. Their World Wide Web pages are at http://www.cs.purdue.edu/people/{joshi,saw,ramakris,enh,jrr}.