DIVIDE-AND-CONQUER METHODS

Divide-and-conquer is a general problem-solving strategy. It is used for large and complex problems, where it is difficult to apply a direct strategy. The idea is to break the problem down into smaller independent subproblems, solve the subproblems, and then merge the subsolutions into the solution to the large problem. The easiest case is that in which the problem can be subdivided into subproblems of the same type. As a simplified example, suppose we need to find the global minimum of a given function over a large finite region. Standard available algorithms for finding the minimum value might be stuck in a local minimum, and never be able to escape and look for a better solution. In order to find a better estimate for the global minimum, one can divide the region into small enough subregions. Then the algorithm is used over each of these subregions separately (this can be done in parallel). At the end, a discrete minimum search is used over the results obtained to produce an estimate of the global minimum.

The steps in a basic divide-and-conquer strategy are:

1. Divide the problem into smaller solvable problems of the same type
2. Solve each of the smaller problems separately
3. Merge the solutions obtained into a solution to the original problem

A more general view of the divide-and-conquer concept takes any part of the problem-solving procedure and divides it. In fact, any computer program execution on a parallel machine can be considered to use the same concept. In this case, the process of dividing and conquering is usually done automatically without the user's intervention. A specific branch of parallel processing is the massively parallel processing (MPP) expressed in the form of artificial neural networks (ANN). Here the problem is not divided into smaller problems, but the solving tool is divided into smaller processing units (or rather constructed from such units). The concept, however, is the same—if you cannot solve it with one complicated tool, a number of primitive units may do the job.

Along the same line of treating the solving tool rather than the problem itself, there are cases for which there exists more than one way to solve the problem. It may also be the case that no single clear best way exists. Our previous example of function minimization is appropriate here, too. A global optimization technique like genetic algorithms (GA) is good for finding a crude global minimum. A differential technique like Newton's algorithm is better in accurately finding a local minimum. One can combine the two algorithms by finding a rough estimate for the global minimum first, and then using the differential technique for refining the solution. In this way, the solving process is divided between two "experts." The first expert shines globally but is rough, whereas the second one excels locally but fails on a global scale.

A special case of integrating different kinds of expertise exists when two or more such experts (algorithms) can solve the problem equally well on a broad problem domain, but for a specific case only one of them is the best. If we cannot tell which is the best for each case, we need some general approach for creating a mixture of experts (ME). This mixture should give an integrated opinion, and perform better than any specific expert alone.

Divide-and-conquer methods gain more and more importance as technological development steadily increases the complexity of process plants, vehicles, and other engineered systems. Dealing with this complexity is a difficult problem, as phrased in the principle of incompatibility expressed by Zadeh (1): "As the complexity of a system increases, our ability to make precise and yet significant statements about its behaviour diminishes until a threshold is reached beyond which precision and significance (or relevance) become almost mutually exclusive characteristics." A consequence of this principle is that one should look for methods that use less precise system knowledge in order to gain enough significance of the results (2). This is the trend in intelligent control where fuzzy logic, qualitative modeling, neural networks, expert systems, and probabilistic reasoning are being explored (3,4,5).

It should be noted that divide-and-conquer is a general concept in science and engineering as well as in other areas. ANN's main contribution toward application of that concept is in being an MPP tool. Specific contributions of ANNs are sparser, and will be discussed in the following sections. The general presentation mostly follows that of modeling and control (2). It originally applied to dynamic modeling and control of complex systems; however it applies to other areas as well. In fact, there is no conceptual difference between dynamic and static modeling, and any functional mapping is indeed a modeling of some relation. In that sense, ANNs are a general modeling tool.

MODELING AND CONTROL

A modeling and control problem can be decomposed along several axes:

1. Decomposition into physical components
2. Decomposition based on phenomena
3. Decomposition in terms of mathematical series expansion
4. Decomposition into goals
5. Decomposition into subspaces
6. Decomposition into multiple experts

All these approaches to problem decomposition are practical, but some may not be applicable for certain problems. For example, a theoretical classification problem may not involve physical components at all. There is also some overlapping between the approaches listed, so that choosing between them may be a question of preference. There are also ways to combine these approaches. For example, the local models decomposed using subspaces may have to be represented in terms of equations based on series expansions or phenomena. In cases of dynamic modeling and control, the subspaces translate into operating regimes.

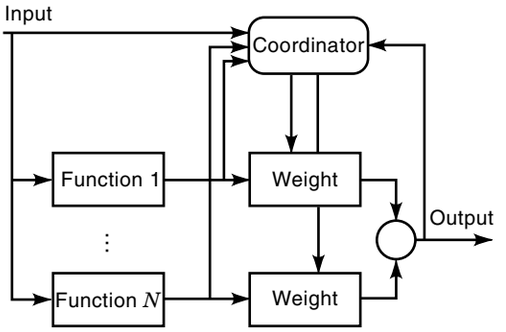

Figure 1. A divide-and-conquer principle schematically realized by weighted coordination of multiple functions. The block on top coordinates the integrated action of the N functions. Each function may vary in the algorithm applied and in its variables and parameters.

Subspace Decomposition

As previously stated, one of the ways to use divide-and-conquer strategies is to partition a complex problem into a number of simpler subproblems that can be solved independently. The individual solutions yield the solution to the original complex problem. One approach to the decomposition of modeling and control problems that has recently attracted significant attention is the subspace decomposition (or operating regime decomposition for dynamic systems) (2).

The main idea in subspace decomposition is to partition the function space of the system, such that one ends with an integration of multiple local models. A central unit coordinates the local parts by selecting a single one, or combining the actions or parameters of a number of local models. Figure 1 shows a scheme of coordination between multiple functions.

The subspaces of the mapping function can often be characterized by different sets of phenomena, which may be simpler to analyze. This simplification may be, for example, due to the fact that linear relations are usually a sufficient approximation looking at a function locally, even though there are complex nonlinearities when viewed globally. Figure 2 illustrates how a function's space is decomposed into a number of possibly overlapping subspaces. These local subspaces have to be combined to yield the global solution to the problem.

When dealing with practical aspects of the decomposition process, there are several issues to be considered. First, there is the local versus global dilemma in which one has to decide how much this decomposition has to be refined. On the one extreme there are a few local subspaces, for which the function might still be quite complex, and on the other extreme there are many local subspaces for which the function is simpler. Thus, one has to decide where the golden path is. Second, there is the curse of dimensionality which says that the number of partitions required for a uniform partitioning increases exponentially with the number of variables on which the function depends. This is why a uniform partitioning is not desired and usually not necessary either.

Hard Partitions and Discrete Logic. Hard partitioning of a function's space means that there is no overlap between subspaces, and for each subspace only one model applies. Figure 3 shows schematically how a space is divided. This scheme can be represented by decision trees (6,7,8), discrete logic, expert systems, and hybrid systems (9,10,11,12,13). In the case of an expert system, it is the integrated "firing" of all relevant rules that contributes to a conclusion. An example of an expert system rule would be:

$$\begin{aligned} &\text{IF engine's temperature IS HIGH} \\ &\text{AND engine's pressure IS LOW} \\ &\text{THEN SHUTDOWN} \end{aligned} \tag{1}$$

Another example is a piecewise function approximation. A function representation in this case would be:

$$f(x) \sum_i f_i(x)\mu_i(x) \tag{2}$$

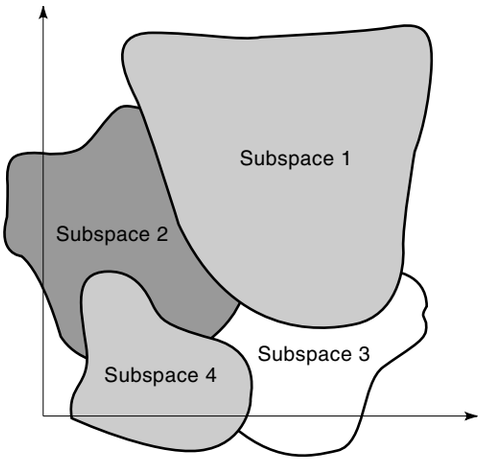

Figure 2. Decomposition of a function's space into a number of overlapping subspaces. Each subspace represents a partial description of the system, and the challenge is to find an appropriate set of such regions and to integrate the different parts.

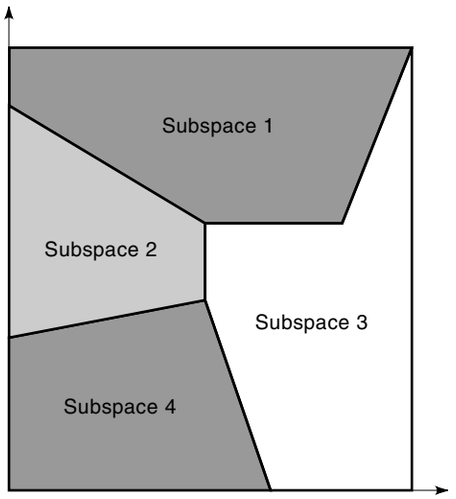

Figure 3. Hard partitioning of a function's space. Subspaces do not overlap. In this case each point belongs to one and only one subspace.

Here f_i is the local function for subspace i, and μ_i is the characteristic function for subspace i:

$$\mu_i(x) = \begin{cases} 1 & \text{for } x \in \text{subspace } i \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Soft Partitions and Fuzzy Logic. Hard partitions and discrete logic are in many cases a crude approximation of the real functional mapping. This is especially true with physical systems. In these cases, a more appropriate way is to move gradually from one subspace to another. Now the function's space is divided into overlapping subspaces, and a smooth transition between them is defined. Appropriate ways to do it are by using fuzzy sets and fuzzy logic (1,12), and interpolation methods (13,14,15). Fuzzy sets are defined by pairs of (member, membership) where each member has a defined membership in the set. Figure 4 shows two fuzzy sets, which define the terms "high" and "low" for some property X. One can see the gradual membership increase of the "high" set members and vice versa for the "low" set. Fuzzy logic enables us to make logical inference based on fuzzy sets, and is actually weighting the various sets, which apply to a given subspace. Now Eq. (2) takes the form:

$$f(x) = \sum_i f_i(x)\rho_i(x) \tag{4}$$

Here ρ_i is a smooth weighting function, and for fuzzy logic inference the following normalization is used (12):

$$\rho_i(x) = \frac{\mu_i(x)}{\sum_j \mu_j(x)} \tag{5}$$

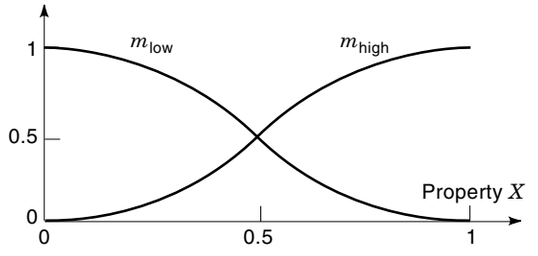

Figure 4. Fuzzy membership functions describing "low" and "high" concepts of property X. The membership function m_{low} defines a "low" value of property X, being high for low values of X, and vice versa for the membership function m_{high}.

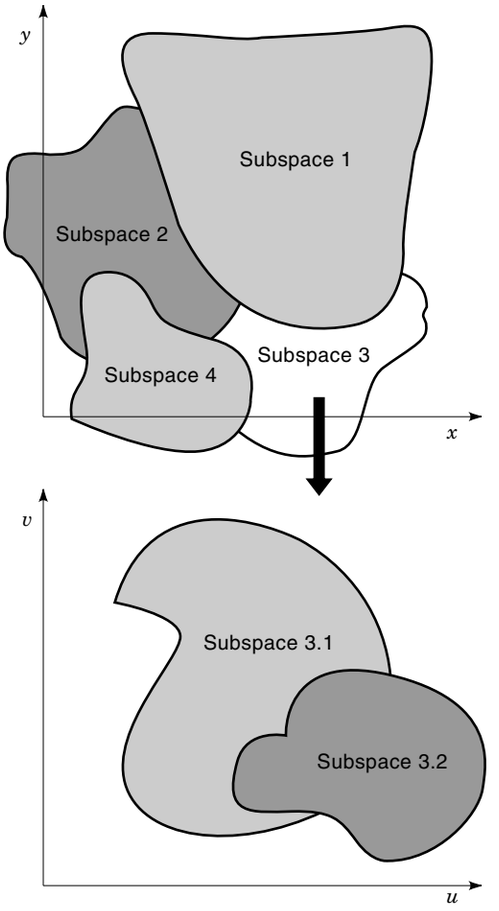

Figure 5. Hierarchical decomposition of a function's space into subspaces. Subspace 3 is further decomposed into subspaces 3.1 and 3.2, thus inducing a two-level hierarchy. Each level uses a different set of variables.

Hierarchical Structures

A different way to divide-and-conquer complex systems is by using hierarchical structures. This technique deals with high dimensional problems by multiresolution representations. The problem is first divided into subspaces over the most important variables' space. Then each subspace is further divided along less important variable axes. Figure 5 illustrates such a structure. This multilevel structure enables us to concentrate first on the most important aspects of the problem, and then resolve the fine details on lower levels. In fact, the paradigm of a multilayer perceptron (MLP) ANN consists of a hierarchical structure. In a classical three-layer, the first layer is the input layer, the second hidden one is a feature extraction layer, and the third one is the decision-making layer. When more than one hidden layer exists, each one builds on finer feature extraction of the previous layer (16).

Multiple Experts

Any problem's solution can be considered as a single expert's advice. Usually there are several ways to solve the problem, and thus there are multiple experts. The question arises then, how we can deal with this multisolution situation. In some cases, one specific solution may be good enough so that we do not really care about any other solutions. In other cases, there may be a clear winner among the possible solutions, and again the decision is obvious as to which one to choose. There are, however, cases for which no one solution can be consid-

ered the best overall possible solution. In such cases one can combine the various solutions. As an example, in most real-life classification problems, there is no one optimal classifier. One has to combine multiple suboptimal and complementary classifiers to yield a better performance than any single one (17). What we actually do is divide the decision burden between the experts, and conquer the problem by integrating the experts' opinions.

Experts as Ensembles of ANNs. An example of using this multiple experts approach is the integration of ensembles of ANNs (18), shown schematically in Figure 6. In this scheme an ensemble consists of a set of ANNs, all of which have the same architecture and data representation. They differ in their training sets, which are *bootstrap* replicas (19) of the same original set. Each ensemble serves as an expert, whose opinion is defined as the average of its members. Suppose there are M such experts, each one consisting of L ANNs. Then the final output for input vector $\boldsymbol{x}_i$ is:

$$O(\boldsymbol{x}_i) = \sum_{m=1}^{M} \alpha_m(\boldsymbol{x}_i) \left[\frac{1}{L} \sum_{l=1}^{L} O_{lm}(\boldsymbol{x}_i) \right] \tag{6}$$

Here O_{lm} is the output of the lth ANN from the mth expert, and the α_m coefficient depends on the input vector. The α_m accounts for the confidence in the mth expert's opinion. This in turn can be estimated from the variance in results of all its L ANNs, and thus it depends on the specific input. The lower the variance is, the more agreement there is and thus the higher our confidence is (20,21). One can define α_m to be inversely proportional to the variance, or choose $\alpha_m = 1$ for the expert whose variance is minimal, and zero otherwise. Shimshoni and Intrator (18) showed that this approach yielded better integrated results than the best individual ANN or best expert did.

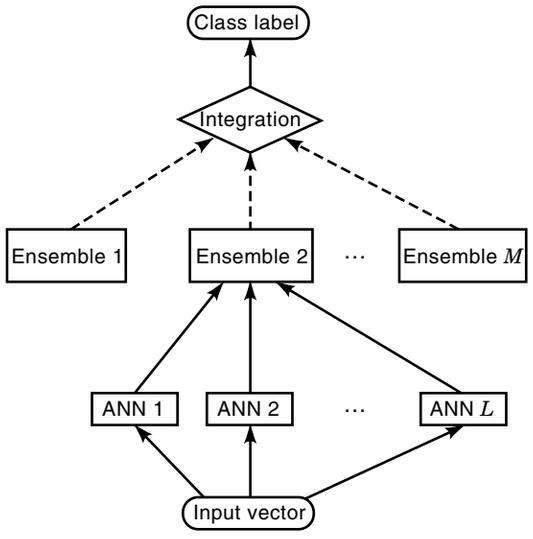

Figure 6. Integrating ensembles of artificial neural networks (ANNs). Each ensemble consists of several ANNs that share the same data representation and architecture, but are trained on different input sets. The top block integrates the ensembles' outputs into a class label.

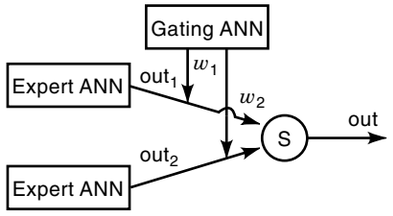

Figure 7. Adaptive mixture of expert ANNs using a gating ANN. All expert ANNs are of the same type, receive the same input and have the same number of outputs. The gating ANN is also of the same type and typically receives the same input as the expert ANNs.

Adaptive Mixtures of Local Experts. A different approach to combine several experts was suggested by Jacobs et al. (22). They realized that training an ANN on a global space with local complexities may cause slow learning and poor generalization. This is a practical issue, since we already know that an ANN can approximate any continuous mapping to a desired precision, given enough degrees of freedom (23,24). The system proposed consisted of several different expert networks, plus a gating network whose function is to decide which of the experts will be used for any given case. The idea was that after the gating network decides which expert(s) will take the responsibility, any error in the output will reflect only on those experts' weights (and the gating network). Figure 7 shows such a system of two experts and a gating network. The gating ANN and the experts' ANNs receive the same input vectors. This gating network makes a stochastic decision about which single expert to use for each case. A simple error function for such a system would be:

$$E = \sum_i w_i \|\boldsymbol{d} - \boldsymbol{out}_i\|^2 \tag{7}$$

Where $\boldsymbol{out}_i$ is the output vector of expert i, $\boldsymbol{d}$ is the desired output, and w_i is the relative contribution of expert i. This contribution is given by:

$$w_i = \frac{e^{s_i}}{\sum_j e^{s_j}} \tag{8}$$

Where s_i is the total weighted input received by output unit i of the gating network. The output vector of the mixture is given by the gating ANN weighting of the individual experts:

$$\boldsymbol{out} = \sum_i w_i \boldsymbol{out}_i \tag{9}$$

With this system, one expert's decision on a given case is not directly influenced by the weights of the other experts. It is indirectly influenced through changes in the relative contributions w_i. The result of the competition between the experts is that the architecture adaptively splits the input space into regions, and learns separate mappings within each region by a separate expert.

Hierarchies of Adaptive Experts. In the same spirit of divide-and-conquer, we can now generalize the method of the previous section into a hierarchical structure. Each expert previously responsible for a defined region in the input space can now be divided into subexperts, each of which is responsible for a subregion. In this way we can recursively define hierarchies of adaptive experts. Jordan and Jacobs (25) show the way to build such a system, depicted schematically in Fig. 8. Now there are clusters of experts, which combine together to the final integrated solution. The output of cluster i is given by:

$$\boldsymbol{out}_i = \sum_j w_{ji} \boldsymbol{out}_{ij} \tag{10}$$

The output of the whole system is given by:

$$\boldsymbol{out} = \sum_i w_i \boldsymbol{out}_i \tag{11}$$

The equations for the gating ANNs' weights are similar to Eq. (8). Here again, the gating ANNs serve as classifiers that partition the input space. The nested structure of the system induces nested partitioning of the input space.

Decomposition in Terms of Mathematical Series Expansion

Divide-and-conquer methods using mathematical series expansion have been used for quite some time for interpolation and function estimation. See, for example, Powel (26) and Micchelli (27). The special form that ANNs had to offer was the adaptive tuning of the free parameters involved. Several ANNs paradigms use this type of expansion, and a few are described here.

Radial Basis Function Neural Networks. Moody and Darken (28) used radial basis function (RBF) ANNs, and showed how to train them. Suppose we want to approximate a continuous or piecewise continuous real-valued function f from R^n to R^m.

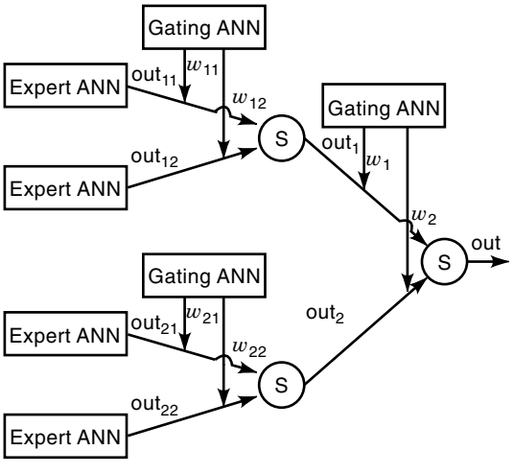

Figure 8. Hierarchical adaptive mixture of expert ANNs. Two hierarchical levels are shown. All expert ANNs and gating ANNs have the same input vector.

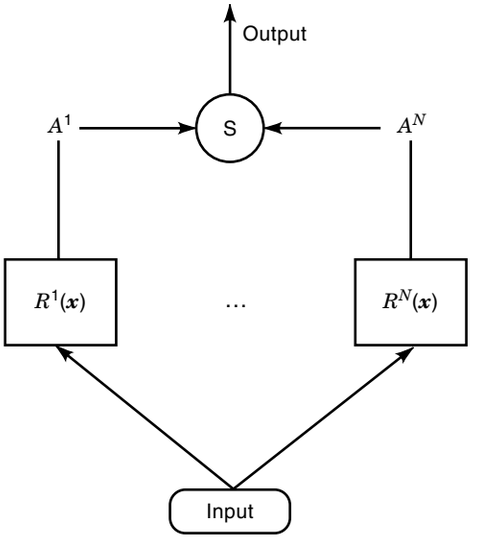

Figure 9. A general scheme of a radial basis function (RBF) network. The network consists of N unit response functions, each of which is locally tuned and has diminishing activation values outside its region of influence.

Then an RBF network is constructed from a linear combination of locally tuned functions as follows:

$$f(\boldsymbol{x}) = \sum_\alpha A^\alpha R^\alpha(\boldsymbol{x}) \tag{12}$$

Here A^α is the weight associated with the ith basis function, and the radial basis functions are given by:

$$R^\alpha(\boldsymbol{x}) \equiv R\left(\frac{\|\boldsymbol{x} - \boldsymbol{x}^\alpha\|}{\sigma^\alpha}\right) \tag{13}$$

As can be seen R^α is a radially symmetric function with a single maximum at the origin, and it drops to zero as the radius increases. The center of the function is at x^α, and its width is σ^α. Figure 9 shows schematically the structure of an RBF network.

A common choice for the basis functions is of the Gaussian form:

$$R^\alpha(\boldsymbol{x}) = \exp\left[-\frac{\|\boldsymbol{x} - \boldsymbol{x}^\alpha\|^2}{(\sigma^\alpha)^2}\right] \tag{14}$$

It should be noted that these basis functions are not orthonormal, not uniformly distributed, and do not have uniform width. The locality of the basis functions contributes to the efficiency of calculating activations, since only functions that are very near to a given input vector will have significant response. Thus only these functions have to be evaluated and trained.

The error measure used for supervised training can be written in the form:

$$E = \frac{1}{2}\sum_i \left(d(\boldsymbol{x}_i) - f(\boldsymbol{x}_i)\right) \tag{15}$$

Where $\boldsymbol{x}_i$ is the ith training vector, $d(\boldsymbol{x}_i)$ is the desired output and $f(\boldsymbol{x}_i)$ is the ANN's output. Moody and Darken (26) first used a conjugate gradient optimization procedure to find all tunable parameters of the network, namely: $\{\boldsymbol{x}^\alpha, \sigma^\alpha, A^\alpha\}$. It turned out, however, that the widths σ^α were not restricted to small values and thus lost the locality of the basis functions. Moreover, some basis functions were located far from the data region, and the convergence was slow.

A better approach used was a three-step hybrid learning procedure. The first step was a standard k-means clustering algorithm (29,30) for choosing $\boldsymbol{x}^\alpha$ values. This technique finds a local minimum of the total squared Euclidean distances E between the training vectors $\boldsymbol{x}_i$ and the nearest of the centers $\boldsymbol{x}^\alpha$:

$$E = \sum_{i,k} M_{i\alpha}(\boldsymbol{x}_i - \boldsymbol{x}^\alpha)^2 \tag{16}$$

The $M_{i\alpha}$ matrix is the cluster membership function consisting of 0s and 1s, which identifies the basis function to which a training vector belongs.

The second step was using "P nearest neighbor" heuristics to find a set of widths, such that the basis functions form a smooth and contiguous interpolation over the space they cover. A simple example of such heuristic would be a uniform width σ, which equals the average Euclidean distance between each basis function's center and its nearest neighbor.

The third step was to optimize the set of A^α coefficients in Eq. (12) such that the error in Eq. (15) is minimized. Since the centers and widths of the basis functions are already fixed, the optimization process is much faster than before, and can be obtained using the linear least squares (LS) method. Moreover, basis functions are now located within the data space, and the overall representation has smoother transition between function centers.

General Regression Neural Networks. A general regression neural network (GRNN) is a one-pass learning algorithm, which provides estimates of continuous variables and converges to the underlying regression surface (31). In this approach, one does not have to assume a specific functional form for the regression, but rather estimate the underlying density adaptively from the data points available.

Suppose $f(\boldsymbol{x}, y)$ is the joint continuous probability density function (PDF) of a vector random variable $\boldsymbol{x}$ and a scalar random variable y. If $\boldsymbol{X}$ is a particular value of $\boldsymbol{x}$, then the conditional mean of y given $\boldsymbol{X}$ is given by:

$$\hat{Y}(\boldsymbol{X}) = \frac{\int_{-\infty}^{\infty} y f(\boldsymbol{X}, y)\, dy}{\int_{-\infty}^{\infty} f(\boldsymbol{X}, y)\, dy} \tag{17}$$

To estimate $f(\boldsymbol{x}, y)$ one can use the estimators proposed by Parzen (32) for the one-dimensional case and by Cacoullos (33) for the multidimensional case. If n sample points $(\boldsymbol{X}_i, Y_i)$ are available based on random variables $\boldsymbol{x}$ and y, and p is the dimension of $\boldsymbol{x}$, then the probability estimator is given by:

$$\hat{f}(\boldsymbol{X}, Y) = \frac{1}{(2\pi)^{(p+1)/2}\sigma^{(p+1)}} \cdot \frac{1}{n}\sum_{i=1}^{n} \exp\left[-\frac{\|\boldsymbol{X} - \boldsymbol{X}_i\|^2 + |Y - Y_i|^2}{2\sigma^2}\right] \tag{18}$$

Substituting this expression into Eq. (17) and performing the integrations, results in:

$$\hat{Y}(\boldsymbol{X}) = \frac{\sum_{i=1}^{N} Y_i \exp\left(-\frac{D_i^2}{2\sigma^2}\right)}{\sum_{i=1}^{n} \exp\left(-\frac{D_i^2}{2\sigma^2}\right)} \tag{19}$$

Where:

$$D_i = \|\boldsymbol{X} - \boldsymbol{X}_i\| \tag{20}$$

Density estimators of the form in Eq. (18) have the property of being consistent estimators, which means that they converge asymptotically to the underlying PDF $f(\boldsymbol{x}, y)$, at all points $(\boldsymbol{x}, y)$ at which the density function is continuous, provided that $\sigma = \sigma(n)$ is a decreasing function of n such that:

$$\sigma(n) \xrightarrow{n\to\infty} 0 \tag{21}$$

and

$$n\sigma^P(n) \xrightarrow{n\to\infty} \infty \tag{22}$$

The density estimated by $\hat{Y}(\boldsymbol{X})$ is actually a weighted average of the Y_i values. As the σ parameter gets larger, this density becomes smoother, and in the limit becomes a multivariate Gaussian with covariance $\sigma^2 I$. This parameter has to be optimized, and for regression estimation the optimization criterion can be defined by the mean squared error (MSE):

$$\text{MSE} = \sum_{i=1}^{n} |Y_i - \hat{Y}(\boldsymbol{X}_i)|^2 \tag{23}$$

In case the dependent variable is a vector $\boldsymbol{Y}$, each of its components is calculated using Eq. (13).

A preprocessing step is usually required which scales all input variables to the same range, such that the kernel used for the estimation has the same width in all dimensions. The width can be found by minimizing the mean squared error in Eq. (23) using the *holdout* method. This method consists of removing one sample at a time, constructing the network based on all the rest, estimate the Y for the removed sample and thus constructing the MSE.

In case the number of samples is large, such that assigning a neuron to each one is not practical, one can use clustering. Each cluster is then represented by one neuron positioned at the cluster's center. This clustering can be done by various methods like learning vector quantization (34), K-means averaging (35), adaptive K-means (28), one-pass K-means or restricted Coulomb energy (RCE) (35). If N_i denotes the number of samples assigned to the ith cluster, then Eq. (19) can be written as:

$$\hat{Y}(\boldsymbol{X}) = \frac{\sum_{i=1}^{n} A_i \exp\left(-\frac{D_i^2}{2\sigma^2}\right)}{\sum_{i=1}^{n} B_i \exp\left(-\frac{D_i^2}{2\sigma^2}\right)} \tag{24}$$

Where A_i is the sum of the Y values and B_i is the number of samples assigned to the ith cluster.

Figure 10 shows a scheme of GRNN for a scalar output, where $\hat{Y}f(\boldsymbol{X})K$ represents the denominator in Eq. (24), and $f(\boldsymbol{X})K$ is the numerator. The main advantages of this scheme are fast learning and convergence to the optimal regression surface as the number of samples increases. The disadvantage is the computation time needed to estimate a new output vector.

Probabilistic Neural Networks. A probabilistic neural network (PNN) (36) is a classification network, similar in structure to the GRNN network. It demonstrates the same divide-and-conquer idea of decomposition in terms of mathematical series expansion, but this time by estimating the likelihood of an input vector to belong to a given class. This is done using the same distribution functions as GRNNs, taking into account any known a priori probabilities. The PNN is essentially an ANNs implementation of a Bayesian classifier. The principle of a Bayesian classifier is that an input vector belongs to the category for which its PDF is highest. This PDF is estimated using Parzen (32) estimator combined with the a priori probabilities.

Suppose we have an input vector space, where each vector $\boldsymbol{x}$ belongs to a given class. The list of possible classes consists of the K classes $C_1, C_2, \ldots, C_K$. The a priori probability for vector $\boldsymbol{X}$ to be in class k is p_k, and the PDF of the kth class is $f_k(\boldsymbol{X})$. Then the Bayes decision rule chooses the class to which $\boldsymbol{x}$ belongs as the one having maximum value out of the following list: $p_1 \cdot f_1(\boldsymbol{X}), p_2 \cdot f_2(\boldsymbol{X}), \ldots, p_K \cdot f_K(\boldsymbol{X})$. This rule provides an optimum classification in terms of minimizing the expected risk. The main difficulty here is the estimation of the PDFs at given input vectors, weighting them and comparing them. This is done here using the same Parzen's estimators we used for the GRNN. The expression for such an estimator would be:

$$f_k(\boldsymbol{X}) = \frac{1}{(2\pi)^{(p+1)/2}\sigma^{(p+1)}} \cdot \frac{1}{N_k} \sum_{i=1}^{N_k} \exp\left[-\frac{\|\boldsymbol{X} - \boldsymbol{X}_{ki}\|^2}{2\sigma^2}\right] \quad (25)$$

Where p is the dimension of $\boldsymbol{X}$, N_k is the number of samples belonging to class k, and $\boldsymbol{X}_{ki}$ is the ith sample in class k. The smoothing parameter depends on N_k such that analogous relations to Eqs. (21) and (22) have to be fulfilled. One possible way to define it is by:

$$\sigma = \sigma(N_k) = a \cdot N_k^{-b} \quad (26)$$

With b being a constant between 0 and 1.

All input vectors are required to be normalized such that:

$$\|\boldsymbol{X}\| = 1.0 \quad (27)$$

Expanding the term in the exponent of Eq. (25) we get:

$$\|\boldsymbol{X} - \boldsymbol{X}_{ki}\|^2 = \|\boldsymbol{X}\|^2 + \|\boldsymbol{X}_{ki}\|^2 + 2\boldsymbol{X}^T \cdot \boldsymbol{X}_{ki} = 2(1.0 + \boldsymbol{X}^T \cdot \boldsymbol{X}_{ki}) \quad (28)$$

Thus, if a neuron has its weights set to the components of a sample vector $\boldsymbol{X}_{ki}$, then the standard summation procedure yields the expression in Eq. (28). Setting the neuron's transfer function to:

$$F(z) = \exp\left(\frac{z-1}{\sigma^2}\right) \quad (29)$$

Yields an output of the form shown in Eq. (25) per sample with $z = \boldsymbol{X}^T \cdot \boldsymbol{X}_{ki}$. What is left is to sum up all these neurons outputs to yield the PDF estimator of Eq. (25). Figure 11 illustrates a paradigm implementing the PNN. It shows the pattern neurons grouped into classes performing Eq. (29), and then the summation units, one per class, performing Eq. (25).

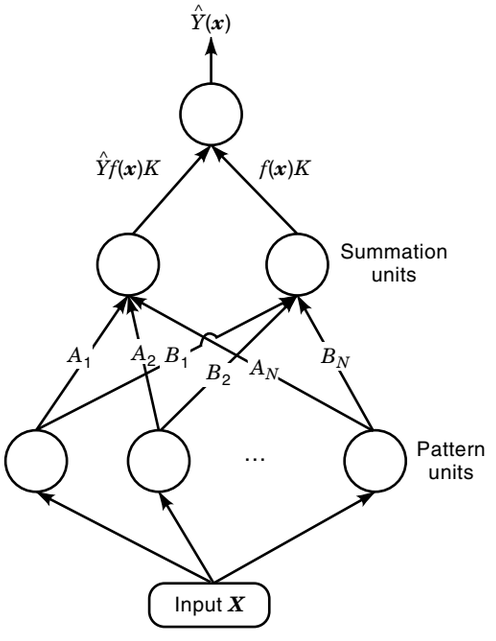

Figure 10. A general scheme of a general regression neural network (GRNN). The summation units perform a dot product between a weight vector and a vector composed of the signals from the pattern units.

The Problem of "Don't Know" Patterns. The PNN paradigm is a localized network. It divides the input space into local influence regions whose level of importance relates directly to the input data density. This gives PNN an important advantage compared to the well known MLP networks. Classification by MLP network is done by building $n - 1$ dimensional boundaries in the n-dimensional data space. A new pattern is classified based on its location relative to the boundaries. Therefore the MLP might give a classification answer with high confidence (e.g., output neuron value equals 1 for that class), for a new type of data on which it has never been trained. This happens when the classification boundaries define a class region that includes some subspace having no training data at all. A new type of data may have its patterns located in such a subspace, thus causing the network to misclassify it as one of its learned types. This may not be important for some applications, but for others like nuclear power plants transient diagnostics, it is crucial (37,38).

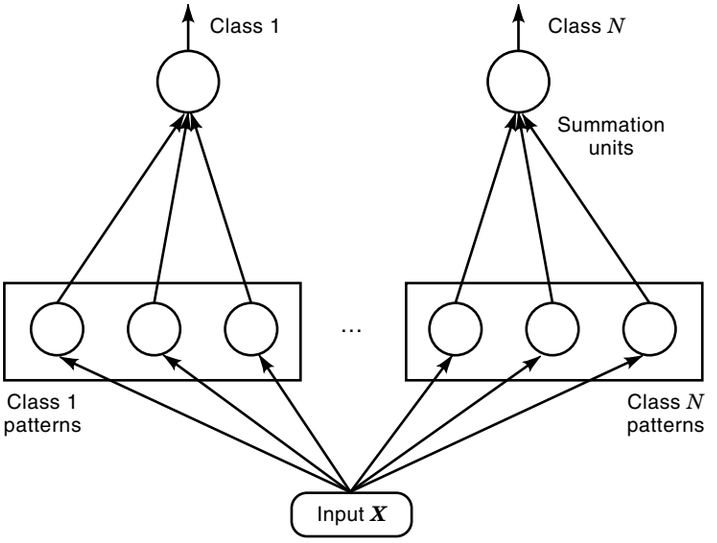

Figure 11. A general scheme of a probabilistic neural network (PNN). The summation units sum up the contribution of all unit functions for each class separately.

A demonstration of this drawback of MLPs is shown in Figure 12. A simple 2-D classification problem was chosen here, where the training set consists of just two patterns corresponding to two different classes encoded by 0 and 1. These two patterns are located at (0.25, 0.25) and (0.75, 0.25) on the 2-D square $[0\text{-}1]^2$. A backpropagation MLP was trained on this set, and then its response was recalled on an equidistance grid of points as a test set. This network had two input neurons, one hidden neuron and one output neuron, and it used the logistic activation transfer function. The recall results are shown as a contour plot in the figure. One can clearly see that this MLP responded with "high confidence" (output equals 0 or 1) for the category of very far points on which it has never been trained. This behavior is typical for MLPs in general. They are only trained to reduce the output error on the training set, and thus their response for far points can literally be anything within their output range. The problem stems from the fact that MLPs have no way of giving a reliable answer of "don't know," unless it has been trained on "don't know" patterns.

A possible remedy to this drawback is to include "don't know" patterns in the training set. Figure 13 shows the results of the previous MLP, now trained on an enhanced training set. This training set includes the original two points encoded as 0 and 1, and a grid of other "don't know" points encoded as 0.5. Here far patterns are classified correctly because we included them in the training set. While this solution may work for a 2-D problem, it is prohibitive for a real multi-D problem because of the curse of dimensionality. This means that most of the multi-D space is

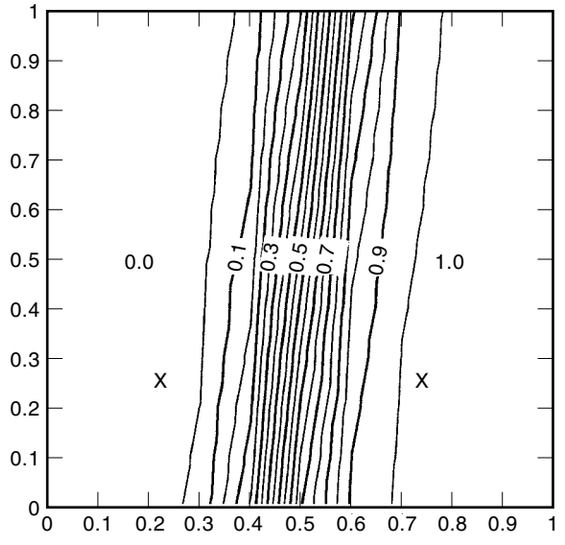

Figure 12. Backpropagation network's output without "don't know" patterns. The two-point training set is shown as X's.

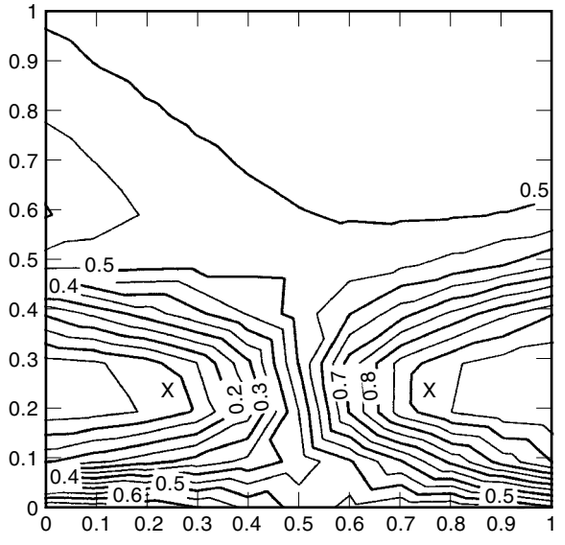

Figure 13. Backpropagation network's contours output with "don't know" patterns. The original two-point training set is shown as X's.

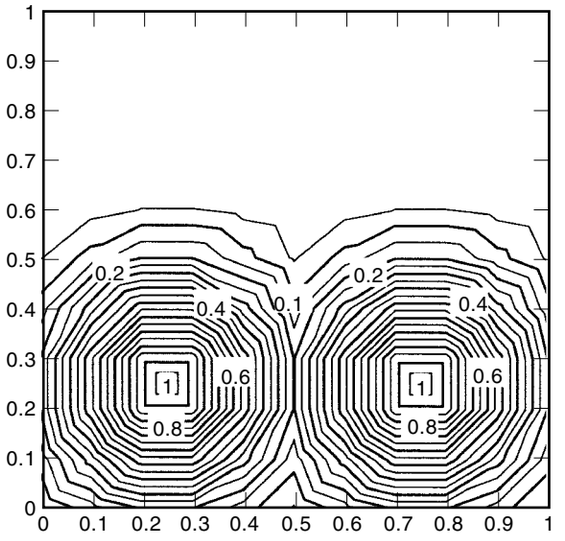

Figure 14. The PNN normalized PDF contours for the two-points training set.

vacant and thus too many "don't know" patterns would be needed.

Networks like the PNN can solve this problem due to their localized nature. Figure 14 demonstrates applying a PNN network to the simple 2-D problem. The network has two input neurons, two hidden neurons corresponding to our two known vectors, and two output neurons. The figure shows the probability density of each of the two classes as computed by the network. One can see how the PDFs of the two classes decay as the distance from the training set increases. In particular, the upper left and right part of the square can be clearly classified as "don't know" as opposed to the MLP response in Figure 12.

BIBLIOGRAPHY

1. L. A. Zadeh, Outline of a new approach to the analysis of complex systems and decision processes, *IEEE Trans. Syst. Man Cybern.,* **3**: 28–44, 1973.
2. R. Murray-Smith and T. A. Johansen (eds.), *Multiple Model Approaches to Modelling and Control,* London: Taylor & Francis, 1997.
3. K. J. Åström and T. J. McAvoy, Intelligent control, *J. Process. Control,* **2**: 115–125, 1992.
4. P. J. Antsaklis, K. M. Passino, and S. J. Wang, An introduction to autonomous control systems, *IEEE Control Syst. Mag.,* **11** (4): 5–13, 1991.
5. K. J. Åström, J. J. Anton, and K. E. Årzèn, Expert control, *Automatica,* **22**: 277–286, 1986.
6. L. Breiman et al., *Classification and Regression Trees,* Monterey, CA: Wadsworths & Brooks, 1984.
7. J. E. Strömberg, F. Gustaffson, and L. Ljung, Trees as black-box model structures for dynamical systems, *Proc. Eur. Control Conf.,* Grenoble, France, 1991, pp. 1175–1180.
8. T. D. Sanger, A tree-structured algorithm for reducing computation in networks with separable basis functions, *Neural Comput.,* **3** (1): 67–78, 1991.
9. P. I. Barton and C. C. Pantelides, Modeling of combined discrete/continuous processes, *Amer. Inst. Chem. Eng. J.,* **40**: 966–979, 1994.
10. W. J. Bencze and G. F. Franklin, A separation principle for hybrid control system design, *IEEE Control Syst. Mag.,* **15** (2): 80–85, 1995.
11. K. E. Simonyi, N. K. Loh, and R. E. Haskell, An application of expert hierarchical control to piecewise linear systems, *Proc. 28th IEEE Conf. Decision Control,* Tampa, FL, 1989, pp. 822–827.
12. T. Takagi and M. Sugeno, Fuzzy identification of systems and its applications for modeling and control, *IEEE Trans. Syst. Man Cybern.,* **15**: 116–132, 1985.
13. K. Stokbro, J. A. Hertz, and D. K. Umberger, Exploiting neurons with localized receptive fields to learn chaos, *J. Complex Syst.,* **4**: 603, 1990
14. R. D. Jones et al., *Nonlinear adaptive networks: A little theory, a few applications,* Technical Report 91-273, Los Alamos, NM: Los Alamos National Lab., 1991.
15. T. A. Johansen and B. A. Foss, Constructing NARMAX models using ARMAX models, *Int. J. Control,* **58**: 1125–1153, 1993.
16. J. Hertz, A. Krogh, and R. G. Palmer, *Introduction to the Theory of Neural Networks,* Reading, MA: Addison-Wesley, 1991.
17. D. H. Wolpert, Stacked generalization, *Neural Net.,* **5** (2): 241–259, 1992.
18. Y. Shimshoni and N. Intrator, Classification of seismic signals by integrating ensembles of neural networks, *IEEE Trans. Signal Process.,* **46**: 1194–1201, 1998.
19. L. Breiman, Bagging predictors, Technical report. Berkeley, CA: University of California, 1994.
20. L. K. Hansen and P. Salamon, Neural networks ensembles, *IEEE Trans. Pattern Anal. Mach. Intell.,* **12**: 993–1001, 1990.
21. A. Krogh and J. Vedelsby, Neural networks ensembles, cross validation and active learning, *Advances Neural Inf. Process. Syst.,* **7**: 1995.
22. R. A. Jacobs et al., Adaptive mixtures of local experts, *Neural Comput.,* **3**: 79–87, 1991.
23. G. Cybenko, Approximations by superpositions of a sigmoidal function, *Math. Control Signals Syst.,* **2**: 4, 303–314, 1989.
24. K. Funahashi, On the approximate realization of continuous mappings by neural networks, *Neural Netw.,* **2**: 183–192, 1989.
25. M. I. Jordan and R. A. Jacobs, Hierarchical mixtures of experts and the EM algorithm, *Neural Computation,* **6** (2): 181–214, 1994.
26. M. J. D. Powell, Radial basis functions for multivariable interpolation: A review, *IMA Conf. Algorithms Approximation Functions DATA,* RMCS Shrivenham, 1985.
27. A. C. Micchelli, Interpolation of scattered data: distance matrices and conditionally positive definite functions, *Constructive Approximation,* **2**: 11–22, 1986.
28. J. Moody and C. J. Darken, Fast learning in networks of locally-tuned processing units, *Neural Comput.,* **1**: 281–294, 1989.
29. S. P. Lloyd, Least squares quantization in PCM, *IEEE Trans. Inf. Theory,* **IT-28**: 129–137, 1982.
30. J. MacQueen, Some methods for classification and analysis of multi-variate observations, in L. M. LeCam and J. Neyman (eds.), *Proc. 5th Berkeley Symp. Math. Probability,* Berkeley, CA: Univ. California Press, 1967, p. 281.
31. D. F. Specht, A general regression neural network, *IEEE Trans. Neural Netw.,* **2**: 568–576, 1991.
32. E. Parzen, On estimation of a probability density function and mode, *Ann. Math. Stat.,* **33**: 1065–1076, 1962.

33. T. Cacoullos, Estimation of a multivariate density, *Ann. Inst. Stat. Math.,* Tokyo, **18** (2): 179–189, 1966.
34. P. Burrascano, Learning vector quantization for the probabilistic neural network, *IEEE Trans. Neural Netw.,* **2**: 458–461, 1991.
35. J. T. Tou and R. C. Gonzalez, *Pattern Recognition Principles,* Reading, MA: Addison-Wesley, 1974.
36. D. F. Specht, Probabilistic neural networks, *Neural Netw.,* **3**: 109–118, 1990.
37. Y. Bartal, J. Lin, and R. E. Uhrig, Nuclear power plants transient diagnostics using LVQ, or some networks don't know that they don't know, *IEEE Int. Conf. Neural Netw.,* Orlando, FL, 1994.
38. Y. Bartal, J. Lin, and R. E. Uhrig, Nuclear power plant transient diagnostics using artificial neural networks that allow "don't know" classifications, *Nuclear Tech.,* **110**: 436–449, 1995.

YAIR BARTAL
Soreq Nuclear Research Center

DIVIDE AND RULE. See DIVIDE-AND-CONQUER METHODS.

DLTS. See DEEP-LEVEL TRANSIENT SPECTROSCOPY.