Neural networks and their applications

Chris M. Bishop
Neural Computing Research Group, Department of Computer Science and Applied Mathematics, Aston University, Birmingham, B4 7ET, United Kingdom

(Received 16 August 1993; accepted for publication 1 March 1994)

Neural networks provide a range of powerful new techniques for solving problems in pattern recognition, data analysis, and control. They have several notable features including high processing speeds and the ability to learn the solution to a problem from a set of examples. The majority of practical applications of neural networks currently make use of two basic network models. We describe these models in detail and explain the various techniques used to train them. Next we discuss a number of key issues which must be addressed when applying neural networks to practical problems, and highlight several potential pitfalls. Finally, we survey the various classes of problem which may be addressed using neural networks, and we illustrate them with a variety of successful applications drawn from a range of fields. It is intended that this review should be accessible to readers with no previous knowledge of neural networks, and yet also provide new insights for those already making practical use of these techniques.

TABLE OF CONTENTS

I. INTRODUCTION 1803
A. Overview of Neural Networks 1804
B. Biological Neural Networks 1804
C. Artifical Neural Networks 1805
D. A Brief History of Neural Computing 1806
II. MULTIVARIATE NON-LINEAR MAPPINGS 1806
A. Analogy with Polynomial Curve Fitting 1807
B. Error Functions and Network Training 1807
C. Interpolation and Classification 1808
III. THE MULTILAYER PERCEPTRON 1808
A. Architecture of the Multilayer Perceptron 1808
B. Network Training 1811
C. Gradient Descent 1813
D. Alternative Training Algorithms 1814
IV. RADIAL BASIS FUNCTION NETWORKS 1815
A. Structure of the Radial Basis Function Network 1815
B. Choosing the Basis Function Parameters 1816
C. Choosing the Second-layer Weights 1817
V. LEARNING AND GENERALIZATION 1817
A. Interpretation of Network Outputs 1818
B. Generalization 1819
C. Determination of Network Topology 1820
VI. DATA PREPROCESSING 1821
A. The Curse of Dimensionality 1821
B. Linear Rescaling 1822
C. Feature Extraction 1822
D. Prior Knowledge 1823
VII. IMPLEMENTATION OF NEURAL NETWORKS 1823
A. Software Implementation 1823
B. Hardware Implementation 1823
VIII. EXAMPLE APPLICATIONS 1824
A. Interpolation 1824
B. Classification 1826
C. Inverse Problems 1827
D. Control Applications 1829
IX. DISCUSSION 1830
A. Other Network Models 1830
B. Future Developments 1830
APPENDIX: A GUIDE TO THE NEURAL COMPUTING LITERATURE 1830

I. INTRODUCTION

Since the late 1980's there has been a dramatic growth in the level of research activity in neural networks, accompanied by extensive coverage in the popular press. While much of the research effort has been concerned with developing fundamental principles and new algorithms, there has also been an increasing drive towards real-world applications. Indeed, the last few years have seen the subject mature to the point where numerous practical applications are in routine use across a range of fields. It is now clear that neural networks offer a powerful set of tools for solving problems in pattern recognition, data processing, and non-linear control, which can be regarded as complementary to those of more conventional approaches. Scientific instrumentation in particular is one area where there is an increasing need for fast non-linear methods for data processing, and where neural network techniques have much to offer.

Of the many neural network models which have been developed, we shall focus primarily on two, known respectively as the *multilayer perceptron* and the *radial basis function network*. These networks presently form the basis for the

0034-6748/94/65(6)/1803/30/$6.00
© 1994 American Institute of Physics

majority of practical applications, and therefore represent the models which are likely to be of most direct interest to the present audience. They form part of a general class of network models known as *feedforward* networks, which have been the subject of considerable research in recent years. A guide to the neural computing literature, given at the end of this review, should provide the reader with some suggested starting points for learning about other models.

Much of the research on neural network applications reported in the literature appeals to ad-hoc ideas, or loose analogies to biological systems. Here we shall take a "principled" view of neural networks, based on well established theoretical and statistical foundations. Such an approach frequently leads to considerably improved performance from neural network systems, as well as providing greater insight. A more extensive treatment of neural networks, from this principled perspective, can be found in the book "Neural Networks for Statistical Pattern Recognition."[1]

A. Overview of neural networks

The conventional approach to computing is based on an explicit set of programmed instructions, and dates from the work of Babbage, Turing, and von Neumann. Neural networks represent an alternative computational paradigm in which the solution to a problem is learned from a set of examples. The inspiration for neural networks comes originally from studies of the mechanisms for information processing in biological nervous systems, particularly the human brain. Indeed, much of the current research into neural network algorithms is focused on gaining a deeper understanding of information processing in biological systems. However, the basic concepts can also be understood from a purely abstract approach to information processing.[1,2] For completeness we give a brief overview of biological neural networks later in this section. However, our focus in this review will be primarily on artificial networks for practical applications.

A feedforward neural network can be regarded as a nonlinear mathematical function which transforms a set of input variables into a set of output variables. The precise form of the transformation is governed by a set of parameters called *weights* whose values can be determined on the basis of a set of examples of the required mapping. The process of determining these parameters values is often called *learning* or *training*, and may be a computationally intensive undertaking. Once the weights have been fixed, however, new data can be processed by the network very rapidly. We shall find it convenient at several points in this review to draw an analogy between artificial neural networks and the standard technique of curve fitting using polynomial functions. A polynomial can be regarded as a mapping from a single input variable to a single output variable. The coefficients in the polynomial are analogous to the weights in a neural network, and the determination of these coefficients (by minimizing a sum-of-squares error) corresponds to the process of network training.

As well as offering high processing speed, neural networks have the important capability of learning a general solution to a problem from a set of specific examples. For many applications this circumvents the need to develop a first-principles model of the underlying physical processes, which can often prove difficult or impossible to find.

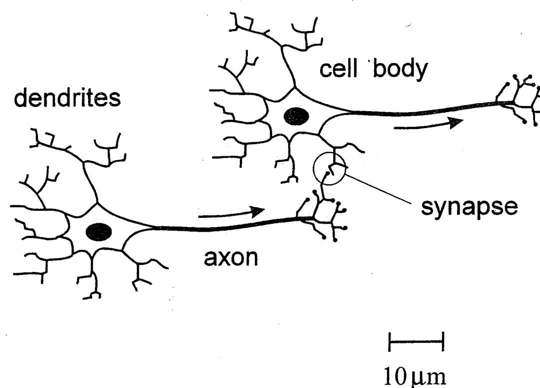

FIG. 1. Schematic illustration of two biological neurons. The dendrites act as inputs, and when a neuron fires an action potential propagates along its axon in the direction shown by the arrow. Interaction between neurons takes place at junctions called synapses.

The principal disadvantages of neural networks stem from the need to provide a suitable set of example data for network training, and the potential problems which can arise if a network is required to extrapolate to new regions of the input space which are significantly different from those corresponding to the training data. In many practical applications these problems will not be relevant, while in other cases various techniques can be used to mitigate their worst effects.[3]

The advantages and limitations of neural networks are often complementary to those of conventional data processing techniques. Broadly speaking, neural networks should be considered as possible candidates to solve those problems which have some, or all, of the following characteristics: (i) there is ample data for network training; (ii) it is difficult to provide a simple first-principles or model-based solution which is adequate; (iii) new data must be processed at high speed, either because a large volume of data must be analyzed, or because of some real-time constraint; (iv) the data processing method needs to be robust to modest levels of noise on the input data.

B. Biological neural networks

The human brain is the most complex structure known, and understanding its operation represents one of the most difficult and exciting challenges faced by science. Biological neural networks provide a driving force behind a great deal of research into artificial network models, which is complementary to the desire to build better pattern recognition and information processing systems. For completeness we give here a simplified outline of biological neural networks.

The human brain contains around 10^{11} electrically active cells called *neurons*. These exist in a large variety of different forms, although most have the common features indicated in Fig. 1. The branching tree of *dendrites* provides a set of inputs to the neuron, while the *axon* acts as an output. Communication between neurons takes place at junctions called *synapses*. Each neuron typically makes connec-

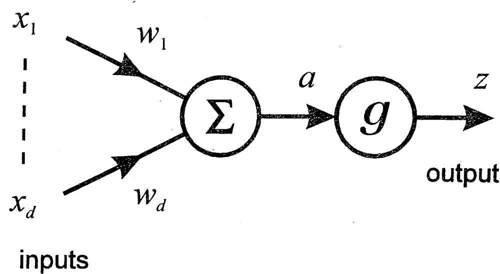

FIG. 2. The McCulloch-Pitts model of a single neuron forms a weighted sum of the inputs $x_1,...,x_d$ given by $a=\Sigma_i w_i x_i$ and then transforms this sum using a non-linear activation function $g(\ )$ to give a final output $z=g(a)$.

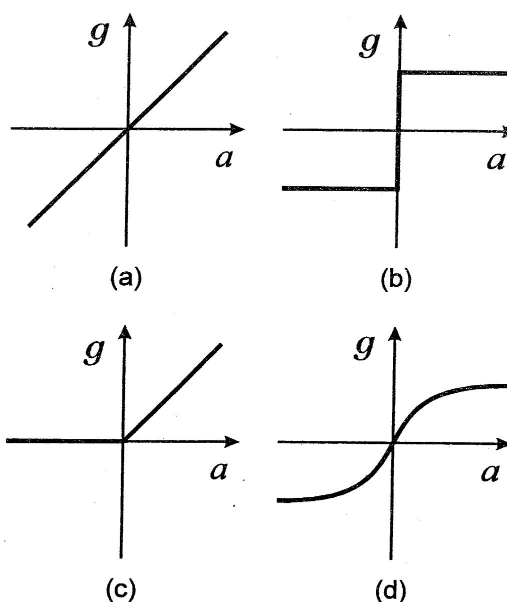

FIG. 3. A selection of typical activation functions: (a) linear, (b) threshold, (c) threshold linear, (d) sigmoidal. The multilayer perceptron network makes use of sigmoidal units to give network mapping functions which are both non-linear and differentiable.

tions to many thousands of other neurons, so that the total number of synapses in the brain exceeds 10^{14}. Although each neuron is a relatively slow information processing system (operating on an effective time scale of around 1 ms) the massive parallelism of information processing at many synapses simultaneously leads to an effective processing power which greatly exceeds that of present day supercomputers. It also leads to a high degree of fault tolerance, with many neurons dying each day with little adverse effect on performance.

Many neurons act in an all-or-nothing manner, and when they "fire" they send an electrical impulse (called an action potential) which propagates from the cell body along the axon. When this signal reaches a synapse it triggers the release of chemical neuro-transmitters which cross the synaptic junction to the next neuron. Depending on the type of synapse, this can either increase (excitatory synapse) or decrease (inhibitory synapse) the probability of the subsequent neuron firing. Each synapse has an associated strength (or weight) which determines the magnitude of the effect of an impulse on the post-synaptic neuron. Each neuron thereby computes a weighted sum of the inputs from other neurons, and, if this total stimulation exceeds some threshold, the neuron fires. As we shall see later, networks of such neurons have very general information processing capabilities.

A key property of both real and artificial neural systems is their ability to modify their responses as a result of exposure to external signals. This is generally referred to as learning, and occurs primarily through changes in the strengths of the synapses.

The above, grossly simplified, picture of biological neural systems provides a convenient starting point for a discussion of artificial network models. Unfortunately, lack of space prevents a more comprehensive overview, and the interested reader is referred to Refs. 4–6 for more information.

C. Artificial neural networks

A simple mathematical model of a single neuron was introduced in a seminal paper by McCulloch and Pitts in 1943,[7] and takes the form indicated in Fig. 2. It can be regarded as a non-linear function which transforms a set of input variables x_i, $(i=1,...,d)$ into an output variable z. Note that from now on we shall refer to an artificial model of a neuron as a *processing unit*, or simply *unit*, to distinguish it from its biological counterpart.

In the McCulloch-Pitts model, the signal x_i at input i is first multiplied by a parameter w_i known as a *weight* (which is analogous to the synaptic strength in a biological network) and is then added to all the other weighted input signals to give a total input to the unit of the form

$$a=\sum_{i=1}^{d} w_i x_i + w_0 , \tag{1}$$

where the offset parameter w_0 is called a *bias* (and corresponds to the firing threshold in a biological neuron). Formally, the bias can be regarded as a special case of a weight from an extra input whose value x_0 is permanently set to $+1$. Thus we can write Eq. (1) in the form

$$a=\sum_{i=0}^{d} w_i x_i , \tag{2}$$

where $x_0=1$. Note that the weights (and the bias) can be of either sign, corresponding to excitatory or inhibitory synapses. The output z of the unit (which may loosely be regarded as analogous to the average firing rate of a neuron) is then given by operating on a with a non-linear *activation function* $g()$ so that

$$z=g(a). \tag{3}$$

Some possible forms for the function $g(\ )$ are shown in Fig. 3. The original McCulloch-Pitts model used the threshold function shown in Fig. 3(b). Most networks of practical interest make use of sigmoidal (meaning S-shaped) activation functions of the kind shown in Fig. 3(d).

As we shall see, this simple model of the neuron forms the basic mathematical element in many artificial neural network models. By linking together many such simple processing elements it is possible to construct a very general class of non-linear mappings, which can be applied to a wide range of practical problems. Adaptation of the weight values, according to an appropriate *training algorithm*, can allow networks to learn in response to external data.

Although we have introduced this mathematical model of the neuron as a representation of the behavior of biological neurons, precisely the same ideas also arise when we consider optimal approaches to the solution of problems in statistical pattern recognition. In this context, expressions such as Eqs. (2) and (3) are known as *linear discriminants*.

D. A brief history of neural computing

The origins of neural networks, or neural computing (sometimes also called neurocomputing or connectionism), lie in the 1940's with the paper of McCulloch and Pitts[7] discussed above. They showed that networks of model neurons are capable of universal computation, in other words they can in principle emulate any general-purpose computing machine.

The next major step was the publication in 1949 of the book *The Organization of Behaviour* by Hebb,[8] in which he proposed a specific mechanism for *learning* in biological neural networks. He suggested that learning occurs through modifications to the strengths of the synaptic interconnections between neurons, such that if two neurons tend to fire together then the synapse between them should be strengthened. This learning rule can be made quantitative, and forms the basis for learning in some simple neural network models (which will not be considered in this review).

During the late 1950's the first hardware neural network system was developed by Rosenblatt.[9,10] Known as the *perceptron*, this was based on McCulloch-Pitts neuron models of the form given in Eqs. (2) and (3). It had an array of photoreceptors which acted as external inputs, and used banks of motor-driven potentiometers to provide adaptive synaptic connections which could retain a learned setting. Adjustments to the potentiometers were made using the *perceptron learning algorithm*.[10] In many circumstances the perceptron could learn to distinguish between characters or shapes presented to the inputs as pixellated images. Rosenblatt also demonstrated theoretically the remarkable result that, if a given problem was soluble in principle by a perceptron, then the perceptron learning algorithm was guaranteed to find the solution in a finite number of steps. Similar networks were also studied by Widrow, who developed the ADALINE (ADAptive LINear Element) network[11] and a corresponding training procedure called the Widrow-Hoff learning rule.[12] These network models are reviewed in Ref. 13. The underlying algorithm is still in routine use for echo cancellation on long distance telephone cables.

The 1960's saw a great deal of research activity in neural networks, much of it characterized by a lack of rigor, sometimes bordering on alchemy, as well as excessive claims for the capability and near-term potential of the technology. Despite initial successes, however, momentum in the field began to diminish towards the end of the 1960's as a number of difficult problems emerged which could not be solved by the algorithms then available. In addition, neural computing suffered fierce criticism from proponents of the field of Artificial Intelligence (which tries to formulate solutions to pattern recognition and similar problems in terms of explicit sets of rules), centering around the book *Perceptrons*[14] by Minsky and Papert. Their criticism focused on a class of problems called *linearly non-separable* which could not be solved by networks such as the perceptron and ADALINE. The field of neural computing fell into disfavor during the 1970's, with only a handful of researchers remaining active.

A dramatic resurgence of interest in neural networks began in the early 1980's and was driven in large part by the work of the physicist Hopfield,[15,16] who demonstrated a close link between a class of neural network models and certain physical systems known as spin glasses. A second major development was the discovery of learning algorithms, based on *error backpropagation*[17] (to be discussed at length in Sec. III), which overcame the principal limitations of earlier neural networks such as the simple perceptron. During this period, many researchers developed an interest in neural computing through the books *Parallel Distributed Processing* by Rumelhart *et al.*[6,18,19] An additional important factor was the widespread availability by the 1980's of cheap powerful computers which had not been available 20 years earlier. The combination of these factors, coupled with the failure of Artificial Intelligence to live up to many of its expectations, led to an explosion of interest in neural computing. The early 1990's has been characterized by a consolidation of the theoretical foundations of the subject, as well as the emergence of widespread successful applications. Neural networks can even be found now in consumer electronics and domestic appliances, for applications varying from sophisticated autoexposure on video cameras to "intelligent" washing machines.

Many of the historically important papers from the field of neural networks have been collected together and reprinted in two volumes in Refs. 20 and 21.

II. MULTIVARIATE NON-LINEAR MAPPINGS

In this review we shall restrict our attention primarily to *feedforward* networks, which can be regarded as general purpose non-linear functions for performing mappings between two sets of variables. As we indicated earlier, such networks form the basis for most present day applications. In addition, a sound understanding of such networks provides a good basis for the study of more complex network architectures. Figure 4 shows a schematic illustration of a non-linear function which takes d independent variables $x_1,...,x_d$ and maps them onto c dependent variables $y_1,...,y_c$. In the terminology of neural computing, the x's are called *input* variables and the y's are called *output* variables. As we shall see, a wide range of practical applications can be cast in this framework.

As a specific example, consider the problem of analyzing a Doppler-broadened spectral line. The x's might represent the observed amplitudes of the spectrum at various wavelengths, and the y's might represent the amplitude,

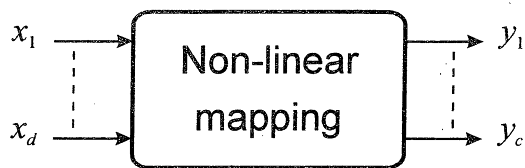

FIG. 4. Schematic illustration of a general non-linear functional mapping from a set of input variables $x_1,...,x_d$ to a set of output variables $y_1,...,y_c$. Each of the y_k can be an arbitrary non-linear function of the inputs.

width, and central wavelength of the spectral line. A suitably trained neural network can then provide a direct mapping from the observed data onto the required spectral line parameters. Practical applications of neural networks to spectral analysis problems of this kind can be found in Refs. 22 and 23, and will be discussed further in Sec. VIII.

It is sometimes convenient to gather the input and output variables together to form input and output vectors which we shall denote by $\mathbf{x}\equiv(x_1,...,x_d)$ and $\mathbf{y}\equiv(y_1,...,y_c)$. The precise form of the function which maps $\mathbf{x}$ to $\mathbf{y}$ is determined both by the internal structure (i.e., the topology and choice of activation functions) of the neural network, and by the values of a set of weight parameters $w_1,...w_{\mathcal{N}}$. Again, the weights (and biases) can conveniently be grouped together to form a *weight vector* $\mathbf{w}\equiv(w_1,...w_{\mathcal{N}})$. We can then write the network mapping in the form $\mathbf{y}=\mathbf{y}(\mathbf{x};\mathbf{w})$, which denotes that $\mathbf{y}$ is a function of $\mathbf{x}$ which is parameterized by $\mathbf{w}$.

In this review we shall consider two of the principal neural network architectures. The first is called the *multilayer perceptron* (MLP) and is currently the most widely used neural network model for practical applications. The second model is known as the *radial basis function* (RBF) network, which has also been used successfully in a variety of applications, and which has a number of advantages, as well as limitations, compared with the MLP. Although this by no means exhausts the range of possible models (which now number many hundreds) these two models together provide the most useful tools for many applications. In Sec. IX we shall give an overview of some of the other major models which have been developed and indicate their potential uses. Some of these models do more than provide static non-linear mappings, as the networks themselves have dynamical properties.

A. Analogy with polynomial curve fitting

We shall find it convenient at several points in this review to draw an analogy between the training of neural networks and the problem of curve fitting using simple polynomials. Consider for instance the mth order polynomial given by

$$y=w_m x^m+...+w_1x+w_0=\sum_{j=0}^{m} w_j x^j. \tag{4}$$

This can be regarded as a non-linear mapping which takes x as an input variable and produces y as an output variable. The precise form of the function $y(x)$ is determined by the values of the parameters $w_0,...w_m$, which are analogous to the weights in a neural network [strictly, w_0 is analogous to a bias parameter, as in Eq. (1)]. Note that the polynomial can be written as a functional mapping in the form $y=y(x;\mathbf{w})$ as was done for more general non-linear mappings above.

There are two important ways in which neural networks differ from such simple polynomials. First, a neural network can have many input variables x_i and many output variables y_k, as compared with the one input variable and one output variable of the polynomial. Second, a neural network can approximate a very large class of functions very efficiently. In fact, a sufficiently large network can approximate any continuous function, for a finite range of values of the inputs, to arbitrary accuracy.[24–29] Thus, neural networks provide a general purpose set of mathematical functions for representing non-linear transformations between sets of variables. Note that, although in principle multi-variate polynomials would satisfy the same property, they would require extremely (exponentially) large numbers of adjustable coefficients. In practice, neural networks can achieve similar results using far fewer parameters, and so offer a practical approach to the representation of general non-linear mappings in many variables.

B. Error functions and network training

The problem of determining the values for the weights in a neural network is called *training* and is most easily introduced using our analogy of fitting a polynomial curve through a set of n data points. We shall label a particular data point with the index $q=1,...,n$. Each data point consists of a value of x, denoted by x^q, and a corresponding desired value for the output y, which we shall denote by t^q. These desired output values are called *target* values in the neural network context. (Note that data points are sometimes also referred to as *patterns*.) In order to find suitable values for the coefficients in the polynomial, it is convenient to consider the error between the desired output value t^q, for a particular input x^q, and the corresponding value predicted by the polynomial function given by $y(x^q;\mathbf{w})$. Standard curve fitting procedures involve minimizing the square of this error, summed over all data points, given by

$$E=\frac{1}{2}\sum_{q=1}^{n}\{y(x^q;\ \mathbf{w})-t^q\}^2. \tag{5}$$

We can regard E as being a function of $\mathbf{w}$, and so the curve can be fitted to the data by choosing a value for $\mathbf{w}$ which minimizes E. Note that the polynomial (4) is a linear function of the parameters $\mathbf{w}$ and so Eq. (5) is a quadratic function of $\mathbf{w}$. This means that the minimum of E can be found in terms of the solution of a set of linear algebraic equations.

It should be noted that the standard sum-of-squares error, introduced here from a heuristic viewpoint, can be derived from the principle of maximum likelihood on the assumption that the noise on the target data has a Gaussian distribution.[1,2] Even when this assumption is not satisfied, however, the sum-of-squares error function remains of great practical importance. We shall discuss some of its properties in later sections.

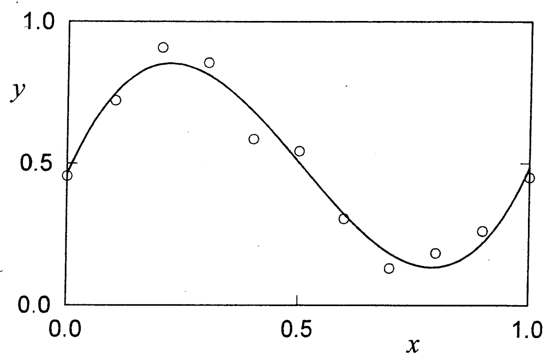

FIG. 5. An example of curve fitting using a polynomial function. Here 11 data points have been generated by sampling the function $\sin(2\pi x)$ at equal intervals of x and then adding zero mean Gaussian noise with variance of 0.05. The solid curve shows a cubic polynomial fitted by minimizing a sum-of-squares error. (From Ref. 1.)

Figure 5 shows an example of a set of data points together with a cubic polynomial [Eq. (4) with $m=3$] which has been fitted to the data by minimizing the sum-of-squares error. We see that the minimum-error curve successfully captures the underlying trend in the data.

The training of a neural network proceeds in an analogous manner. A suitable error function is defined with respect to a set of data points, and the parameters (weights) are chosen to minimize the error. We shall see later that neural network functions depend non-linearly on their weights and so the minimization of the corresponding error function is substantially more difficult than in the case of polynomials, and generally requires the use of iterative non-linear optimization algorithms.

In the case of a neural network, each input vector $\mathbf{x}^q=(x_1^q,\ldots,x_d^q)$ from the data set has a corresponding target vector $\mathbf{t}^q$. The error for output k when the network is presented with pattern q is given by $y_k(\mathbf{x}^q;\mathbf{w})-t_k^q$. The total error for the whole pattern set can then be defined as the squares of the individual errors summed over all output units and over all patterns. This gives an error function, for use in neural network training, of the form

$$E=\frac{1}{2}\sum_{q=1}^{n}\sum_{k=1}^{c}\{y_k(\mathbf{x}^q;\ \mathbf{w})-t_k^q\}^2. \tag{6}$$

While the sum-of-squares error is the most commonly used form of error function, it should be noted that there exist other error measures which may be more appropriate in particular circumstances. (A lengthy discussion of error functions and their properties can be found in Ref. 1.)

C. Interpolation and classification

In polynomial curve fitting the goal is generally to find a smooth representation of the underlying trends in a set of data. We shall refer to this process as interpolation. Typically the data will be noisy and so we are looking for a function which passes close to the data but which does not necessarily pass exactly through each data point. Note that this differs from the problem of strict interpolation in which the aim is to find a function which fits the data exactly. Neural networks can similarly be applied to interpolation problems in which there may be several input and several output variables. The spectral line analysis application mentioned earlier is an example of an interpolation problem, and we shall consider other examples of this type in Sec. VIII.

A second major class of applications for which neural networks may be used are *classification* problems in which the goal is to assign input vectors correctly to one of a number of possible *classes* or categories. One example of a classification problem, which will be discussed in more detail in Sec. VIII, concerns the monitoring of oil flow along a pipe containing a mixture of oil, water, and gas. The inputs to the network consist of measurements from a number of gamma-ray based diagnostics, and the outputs indicate which of a number of possible geometrical flow configurations (stratified, annular, homogeneous, etc.) is present in the pipe.

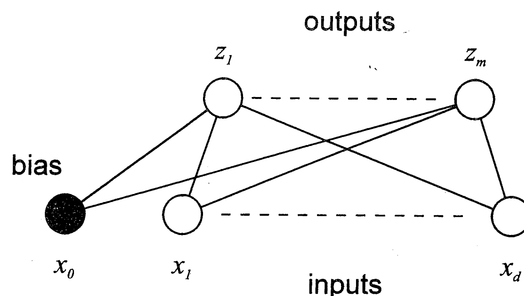

FIG. 6. A single-layer network having d inputs $x_1,\ldots,x_d$ and m outputs $z_1,\ldots,z_m$. Each line in the diagram corresponds to one of the weight parameters in the network function. The biases are shown as weights from an extra input unit (denoted by the solid black circle) whose activation is permanently set to $x_0=1$. (From Ref. 1.)

III. THE MULTILAYER PERCEPTRON

So far we have described feedforward neural networks in terms of non-linear mappings between multi-dimensional spaces. We now introduce one explicit form for the mapping known as the multilayer perceptron network. This class of networks has been used as the basis for the majority of practical applications of neural networks to date.

A. Architecture of the multilayer perceptron

In Sec. I we introduced the concept of a single processing unit described by Eqs. (2) and (3). If we consider a set of m such units, all with common inputs, then we arrive at a neural network having a single layer of adaptive parameters (weights) as illustrated in Fig. 6. The output variables are denoted by z_j and are given by

$$z_j=g\left(\sum_{i=0}^{d} w_{ji}x_i\right), \tag{7}$$

where w_{ji} is the weight from input i to unit j, and $g(\ )$ is an activation function as discussed previously. Again we have included bias parameters as special cases of weights from an extra input $x_0=1$.

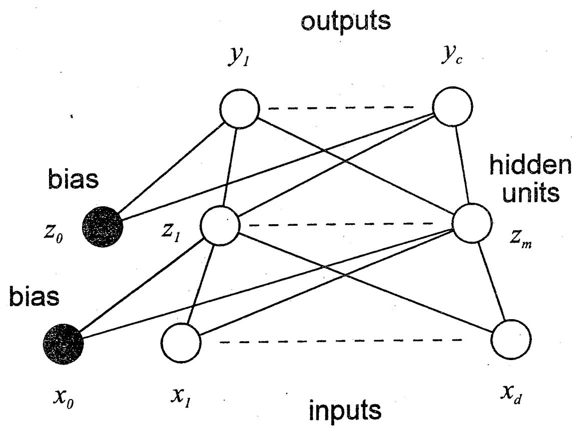

FIG. 7. A multilayer perceptron neural network having two layers of weights. Such networks are capable of approximating any continuous non-linear function to arbitrary accuracy provided the number m of hidden units is sufficiently large. (From Ref. 1.)

Note that Fig. 6 can be regarded as a diagramatic short-hand for function (7), with each element of the diagram corresponding to one of the components of the function. Each circle at the bottom of Fig. 6 represents one of the inputs x_i, each circle at the top represents one of the outputs z_j, and the lines connecting the circles represent the corresponding weights w_{ji}. The extra input $x_0=1$ is shown by the solid black circle, and the lines connecting this unit to the output represent the bias parameters w_{j0}. Single-layer networks such as these were studied extensively in the 1960's. They generally used activation functions $g(\ )$ given by the step function in Fig. 3(b) and were known as *perceptrons*, and were trained using the *perceptron learning algorithm* discussed earlier. Such networks have very limited computational capabilities. In fact, if the linear activation function of Fig. 3(a) is chosen, then the network reduces to simple matrix multiplication. While single-layer networks do have some practical significance, a much more powerful class of networks is obtained if we consider networks having several successive layers of processing units. Such networks were not considered extensively in the 1960's due to the difficulty of finding a suitable training algorithm (the perceptron algorithm only works for single-layer networks). The solution to the problem of training networks having several layers is to replace the step activation functions of Fig. 3(b) with differentiable sigmoidal activation functions of the form shown in Fig. 3(d). This allows techniques of differential calculus to be applied in order to find a suitable training algorithm. Such networks are known as *multilayer perceptrons*.

Figure 7 shows a network with two successive layers of units, and thus two layers of weights. Units in the middle layer are known as *hidden units* since their activation values are not directly accessible from outside the network. The activation of these units is again given by Eq. (7) as in the case of the single-layer network. The outputs of the network are obtained by acting on the z's with a second transformation, corresponding to a second layer of units, to give

$$y_k=\tilde{g}\left(\sum_{j=0}^{m} \tilde{w}_{kj} z_j\right), \tag{8}$$

where $\tilde{w}_{kj}$ denotes a weight in the second layer connecting hidden unit j to output unit k. Note that we have introduced an extra hidden unit with activation $z_0=1$ to provide a bias for the output units. The bias terms (for both the hidden and output units) play an important role in ensuring that the network can represent general non-linear mappings. We can combine Eqs. (7) and (8) to give the complete expression for the transformation represented by the network in the form

$$y_k=\tilde{g}\left(\sum_{j=0}^{m} \tilde{w}_{kj} g\left(\sum_{i=0}^{d} w_{ji} x_i\right)\right). \tag{9}$$

Again, each of the components of Eq. (9) corresponds to an element of the diagram in Fig. 7. Note that the activation function $\tilde{g}$ applied to the output units need not be the same as the activation function g used for the hidden units.

It should be noted that there are two distinct ways of counting the number of layers in a network, both of which are in common use in the neural computing literature. In one convention, a network of the form shown in Fig. 7 would be called a 2-layer network, in which the layers refer to the hidden and output units, or equivalently to the layers of weights. Alternatively, the same network might also be called a 3-layer network in which the layers refer to units and the inputs are counted as one of the layers. We prefer to call this a 2-layer network, since it is the number of layers of weights which primarily determines the capabilities of the network.

If the activation functions $g(\ )$ and $\tilde{g}(\ )$ for the network structure shown in Fig. 7 are taken to be linear, the network transformation reduces to the product of two matrices, which is itself just a matrix. However, if the activation function $g(\ )$ for the hidden units is taken to be non-linear then the network acquires some powerful general-purpose representational capabilities. As we shall see later, in order to train the network we shall need to ensure that its mapping function $\mathbf{y}(\ )$ is differentiable. For these reasons, a sigmoidal (S-shaped) activation function $g(a)$ of the form shown in Fig. 3(d) is often used. In practice, a convenient choice is the "tanh" function given by

$$g(a)=\tanh a \equiv \frac{e^a-e^{-a}}{e^a+e^{-a}}. \tag{10}$$

This has the property, which will prove useful when we discuss network training, that its derivative can easily be expressed in terms of the function itself

$$g'(a)=1+g(a)^2. \tag{11}$$

Another common choice of activation function is the logistic sigmoid given by

$$g(a)=\frac{1}{1+e^{-a}} \tag{12}$$

which also has a simple expression for its derivative

$$g'(a)=g(a)\{1-g(a)\}. \tag{13}$$

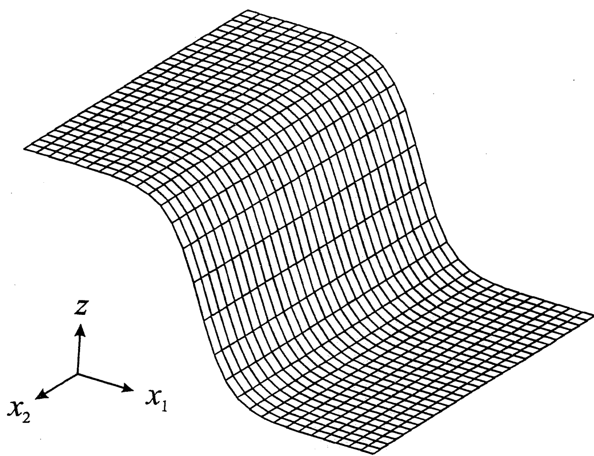

FIG. 8. Plot of the response $z(x_1,x_2)$ of a unit with a sigmoidal activation function, as a function of its two input variables x_1 and x_2.

The response of a single unit with a logistic sigmoidal activation function, as a function of the input variables for the case of 2 dimensions, is plotted in Fig. 8.

With sigmoidal hidden units, the universal approximation properties of the network hold even if the output units have linear activation functions [so that $\tilde{g}(a)=a$ and in effect no activation function is applied]. For interpolation problems, in which we wish to generate mappings whose outputs represent smoothly varying quantities, it is convenient and sufficient to choose the output unit activation functions to be linear. For classification problems, however, it is often convenient to apply a logistic sigmoidal activation function of the form (12) to the output units, as this ensures that the network outputs will lie in the range $(0,1)$ which assists in the interpretation of network outputs as probabilities. (Note that in most applications we should also arrange for the outputs to sum to unity, and this can be achieved by using other forms of activation function.[1]) The use of sigmoidal activation functions on the network outputs would be inappropriate for many interpolation problems since we do not in general want to restrict the range of possible network outputs.

Unlike the single-layer network in Eq. (7), a 2-layer network of the form (9) has very general capabilities for function approximation. It has been shown that, provided the number m of hidden units is sufficiently large, such a network can represent any continuous mapping, defined over a finite range of the input variables, to arbitrary accuracy.[24–29] As a simple illustration of this "universal" capability, consider a mapping from a single input variable x to a single output variable y. In Fig. 9 we see four examples of function approximation using a network having 5 hidden units. The circles show data obtained by sampling various functions at equally spaced values of x, and the curves show the network functions obtained by training the network using techniques to be described later. We see that the same network function, with the weights suitably chosen, can indeed represent a wide range of functional forms.

The multilayer perceptron structure which we have considered has a particularly simple topology consisting of two

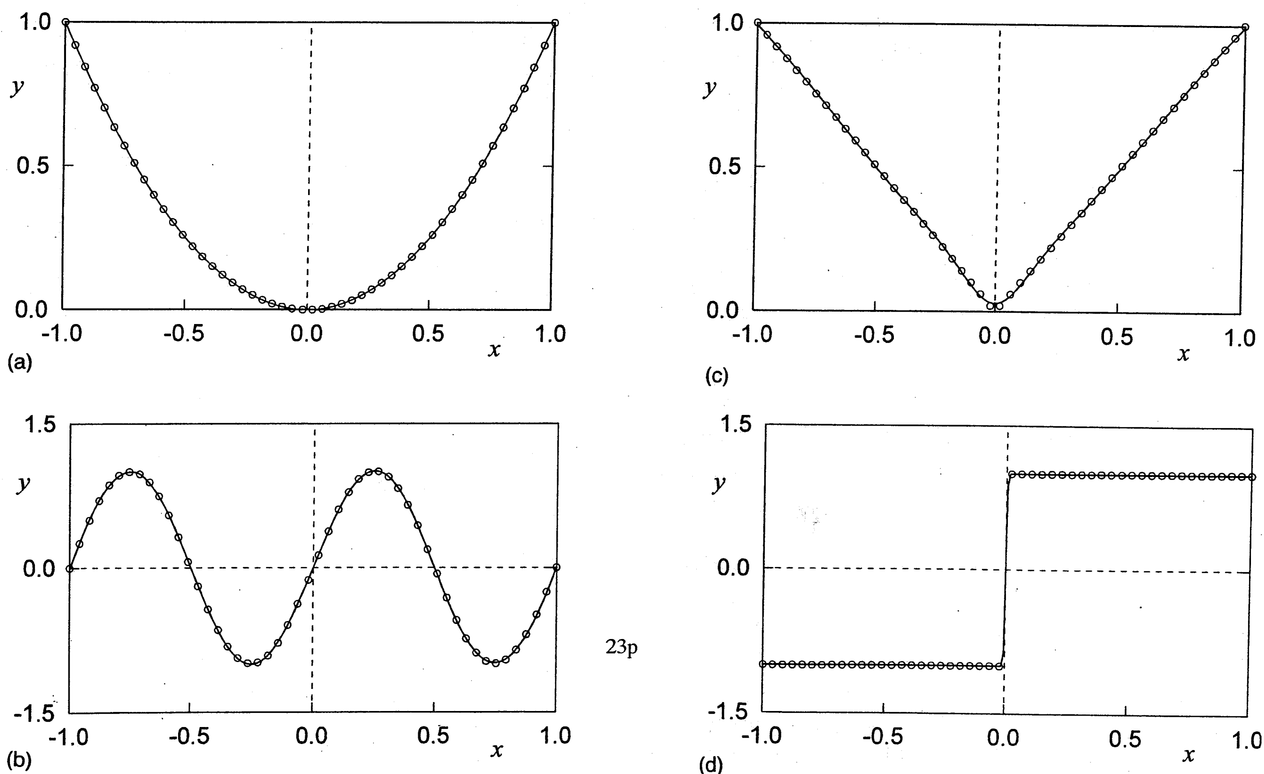

FIG. 9. Four examples of functions learned by a multilayer perceptron with one input unit, 5 hidden units with "tanh" activation functions, and 1 linear output unit. In each case the network function (after training using 1000 cycles of the BFGS quasi-Newton algorithm) is shown by the solid curve. The circles show the data points used for training, which were obtained by sampling the following functions: (a) x^2, (b) $\sin(2\pi x)$, (c) $|x|$, and (d) the Heaviside step function $H(x)$. We see that the same network can be used to approximate a wide range of different functions, simply by choosing different values for the weight and bias parameters. (From Ref. 1.)

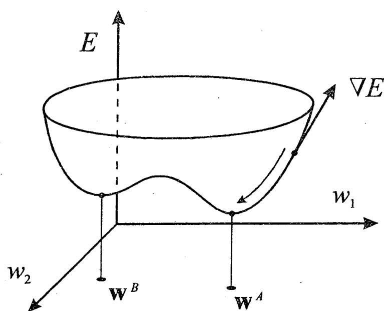

FIG. 10. Schematic illustration of the error function $E(\mathbf{w})$ seen as a surface over weight space (the space spanned by the values of the weight and bias parameters $\mathbf{w}=\{w_1,...,w_I\}$). The weight vector $\mathbf{w}^A$ corresponds to the global minimum of the error function, while the weight vector $\mathbf{w}^B$ corresponds to a local minimum. Network training by the gradient descent algorithm begins with a random choice of weight vector and then proceeds by making small changes to the weight vector so as to move it in the direction of the negative of the error function gradient ∇E, until the weight vector reaches a local or global minimum. (From Ref. 1.)

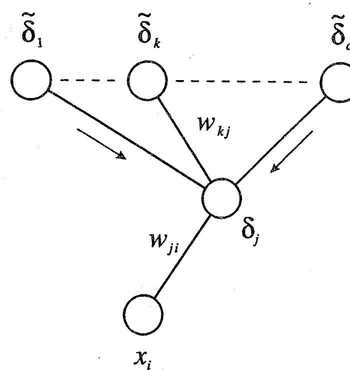

FIG. 11. An illustration of how backpropagation of error signals is used to evaluate derivatives of the error function with respect to the weight (and bias) parameters in the first layer of a 2-layer network. The error signal δ_j at hidden unit j is obtained by summing error signals $\tilde{\delta}_k$ from the output units $k=1,...,c$ after first multiplying them by the corresponding weights w_{kj}. The derivative of the error function with respect to a weight w_{ji} is then given by the product of the error signal δ_j at hidden unit j with the activation z_i of input unit i. (From Ref. 1.)

layers of weights, with full connectivity between inputs and hidden units and between hidden units and output units. In principle, there is no need to consider other architectures, since the 2-layer network already has universal approximation capabilities. In practice, however, it is often useful to consider more general topologies of neural network. One important motivation for this is to allow additional information (called prior knowledge) to be built into the form of the mapping. This will be discussed further in Sec. VI, and a simple example will be given in Sec. VIII. An example of a more complex network structure (having 4 layers of weights) used for fast recognition of postal codes, can be found in Ref. 30. In each case there is a direct correspondence between the network diagram and the corresponding non-linear mapping function.

B. Network training

As we have already discussed, the fitting of a network function to a set of data (network training) is performed by seeking a set of values for the weights which minimizes some error function, often chosen to be the sum-of-squares error given by Eq. (6). The error function can be regarded geometrically as an error surface sitting over weight space, as indicated schematically in Fig. 10. The problem of network training corresponds to the search for the minimum of the error surface. An absolute minimum of the error function, indicated by the weight vector $\mathbf{w}^A$ in Fig. 11, is called a *global minimum*. There may, however, also exist other higher minima, such as the one corresponding to the weight vector $\mathbf{w}^B$ in Fig. 10, which are referred to as *local minima*.

For single-layer networks with linear activation functions, the sum-of-squares error function is a generalized quadratic, as was the case for polynomial curve fitting. It has no local minima, and its global minimum is easily found by solution of a set of linear equations. For multilayer networks, however, the error function is a highly non-linear function of the weights,[31] and the search for the minimum generally proceeds in an iterative fashion, starting from some randomly chosen point in weight space. Some algorithms will find the nearest local minimum, while others are able to escape local minima and offer the possibility of finding a global minimum. In general, the error surface will be extremely complex and for many practical applications a good local minimum may be sufficient to achieve satisfactory results.

Many of the algorithms for performing the error function minimization make use of the derivatives of the error function with respect to the weights in the network. These derivatives form the components of the gradient vector $\nabla E(\mathbf{w})$ of the error function, which, at any given point in weight space, gives the gradient of the error surface, as indicated in Fig. 10. Since there is considerable benefit to the training procedure from making use of this gradient information, we begin with a discussion of techniques for evaluating the derivatives of E.

One of the important features of the class of non-linear mapping functions given by the multilayer perceptron is that there exists a computationally efficient procedure for evaluating the derivatives of the error function, based on the technique of *error backpropagation*.[17] Here we consider the problem of finding the error derivatives for a network having a single hidden layer, as given by the expression in Eq. (9), for the case of a sum-of-squares error function given by Eq. (6). In principle this is very straightforward since, by substituting Eq. (9) into Eq. (6) we obtain the error as an explicit function of the weights, which can then be differentiated using the usual rules of differential calculus. However, if some care is taken over how this calculation is set out, it leads to a procedure which is both computationally efficient and which is readily extended to feedforward networks of arbitrary topology. This same technique is easily generalized to other error functions which can be expressed explicitly as functions of the network outputs. It can be also used to evaluate the elements of the *Jacobian* matrix (the matrix of derivatives of output values with respect to input values) which can be used to study the effects on the outputs of small changes in the input values.[1] Similarly, it can be extended to the evaluation of the second derivatives of the error with

respect to the weights (the elements of the *Hessian* matrix) which play an important role in a number of advanced network algorithms.[32]

First note that the total sum-of-squares error function (6) can be written as a sum over all patterns of an error function for each pattern separately

$$E=\sum_{q=1}^{n} E^q, \quad E^q=\frac{1}{2}\sum_{k=1}^{c} \{y_k(\mathbf{x}^q;\, \mathbf{w})-t_k^q\}^2 , \tag{14}$$

where $y_k(\mathbf{x};\mathbf{w})$ is given by the network mapping Eq. (9). We can therefore consider derivatives for each pattern separately, and then obtain the required derivative by summing over all of the patterns in the data set. For simplicity of notation we shall omit the explicit pattern index q from the various network variables during our discussion of the evaluation of derivatives. It should be borne in mind, however, that all of the input and intermediate variables in the network are evaluated for a given input pattern.

Consider first the derivatives with respect to a weight in the second layer (the layer of weights from hidden to output units). It is convenient to use the notation introduced in Sec. I, and write the network output variables in the form

$$y_k=\tilde{g}(\tilde{a}_k), \quad \tilde{a}_k=\sum_{j=0}^{m} \tilde{w}_{kj}z_j . \tag{15}$$

The derivatives with respect to the final-layer weights can then be written in the form

$$\frac{\partial E^q}{\partial \tilde{w}_{kj}}=\frac{\partial E^q}{\partial \tilde{a}_k}\frac{\partial \tilde{a}_k}{\partial \tilde{w}_{kj}} . \tag{16}$$

We now introduce the definition

$$\tilde{\delta}_k\equiv\frac{\partial E^q}{\partial \tilde{a}_k} . \tag{17}$$

Then, by making use of Eq. (15), we can write the derivative in the form

$$\frac{\partial E^q}{\partial \tilde{w}_{kj}}=\tilde{\delta}_k z_j . \tag{18}$$

We can find an expression for $\tilde{\delta}_k$ by using Eqs. (14), (15), and (17) to give

$$\tilde{\delta}_k=\tilde{g}'(\tilde{a}_k)\{y_k-t_k\}. \tag{19}$$

Because $\tilde{\delta}_k$ is proportional to the difference between the network output and the desired value, it is sometimes referred to as an *error*. Note that, for the sigmoidal activation functions discussed earlier, the derivative $\tilde{g}'(a)$ is easily re-expressed in terms of $\tilde{g}(a)$, as in Eqs. (11) and (13). This provides a small computational saving in a numerical implementation of the algorithm. Note also that the expression for the derivative with respect to a particular weight, given by Eq. (18), takes the simple form of the product of the error at the output end of the weight times the activation of the hidden unit at the other end of the weight. The derivative of the error with respect to any weight in a multilayer perceptron network (of arbitrary topology) can always be written in a form analogous to Eq. (18).

In order to find a corresponding expression for the derivatives with respect to weights in the first layer, we start by writing the activations of the hidden units in the form

$$z_j=g(a_j), \quad a_j=\sum_{i=0}^{d} w_{ji}x_i . \tag{20}$$

We can then write the required derivative as

$$\frac{\partial E^q}{\partial w_{ji}}=\frac{\partial E^q}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} . \tag{21}$$

From Eq. (20) we note that $\partial a_j/\partial w_{ji}=x_i$. If we then define

$$\delta_j\equiv\frac{\partial E^q}{\partial a_j} , \tag{22}$$

we can then write the derivative in the form

$$\frac{\partial E^q}{\partial w_{ji}}=\delta_j x_i . \tag{23}$$

Note that this has the same form as the derivative for a second-layer weight given by Eq. (18), so that the derivative for a given weight connecting an input to a hidden unit is given by the product of the δ for the hidden unit and the value of the input variable.

Finally, we need to find an expression for the δ's. This is easily obtained by using the chain rule for partial derivatives

$$\delta_j=\frac{\partial E^q}{\partial a_j}=\sum_{k=1}^{c}\frac{\partial E^q}{\partial \tilde{a}_k}\frac{\partial \tilde{a}_k}{\partial a_j} . \tag{24}$$

By making use of Eqs. (15), (17), and (20) we obtain

$$\delta_j=g'(a_j)\sum_{k=1}^{c} \tilde{w}_{kj}\tilde{\delta}_k . \tag{25}$$

The expression in Eq. (25) can be interpreted in terms of the network diagram as a propagation of error signals, given by $\tilde{\delta}_k$, backwards through the network along the second-layer weights. This is illustrated in Fig. 11, and is the origin of the term *error backpropagation.*

It is worth summarizing the various steps involved in evaluation of the derivatives for a multilayer perceptron network

(1) For each pattern in the data set in turn, evaluate the activations of the hidden units using Eq. (20) and of the output units using Eq. (15). This corresponds to the forward propagation of signals through the network.
(2) Evaluate the individual errors for the output units using Eq. (19).

(3) Evaluate the errors for the hidden units using Eq. (25). This is the error backpropagation step.
(4) Evaluate the derivatives of the error function for this particular pattern using Eqs. (18) and (23).
(5) Repeat steps 1 to 4 for each pattern in the data set and then sum the derivatives to obtain the derivative of the complete error function.

An important feature of this approach to the calculation of derivatives is its computational efficiency. Since the number of weights is generally much larger than the number of units, the dominant contribution to the cost of a forward or a backward propagation comes from the evaluation of the weighted sums (with the evaluation of the activation functions being negligible by comparison). Suppose the network has a total of $\mathcal{N}$ weights, and we wish to know how the cost of evaluating the derivatives scales with $\mathcal{N}$. Since the error function $E^q(\mathbf{w})$ for pattern q is a function of all of the weights, a single evaluation of E^q will take $\mathcal{O}(\mathcal{N})$ steps (i.e., the number of numerical steps needed to evaluate E will grow like $\mathcal{N}$). Similarly, the direct evaluation of any one of the derivatives of E^q with respect to a weight would also take $\mathcal{O}(\mathcal{N})$ steps. Since there are $\mathcal{N}$ such derivatives we might expect that a total of $\mathcal{O}(\mathcal{N}^2)$ steps would be needed to evaluate all of the derivatives. However, the technique of backpropagation allows all of the derivatives to be evaluated using a single forward propagation, followed by a single backward propagation, followed by the use of the formulas (18) and (23). Each of these requires $\mathcal{O}(\mathcal{N})$ operations and so all of the derivatives can be evaluated in $\mathcal{O}(3\mathcal{N})$ steps. For a data set of n patterns the derivatives of the complete error function $E=\Sigma_q E^q$ can therefore be found in $\mathcal{O}(3n\mathcal{N})$ steps, as compared with the $\mathcal{O}(n\mathcal{N}^2)$ steps that would be needed by a direct evaluation of the separate derivatives. In a typical application $\mathcal{N}$ may range from a few hundred to many thousands, and the saving of computational effort is therefore significant. Since, even with the use of backpropagation to evaluate error derivatives, the training of a multilayer perceptron is computationally demanding, the importance of this result is clear. In this respect, error backpropagation is analogous to the fast Fourier transform (FFT) technique which allows the evaluation of Fourier transforms to be reduced from $\mathcal{O}(\mathcal{M}^2)$ to $\mathcal{O}(\mathcal{M}\ln\mathcal{M})$, where $\mathcal{M}$ is the number of Fourier components.

C. Gradient descent

The simplest scheme for using the gradient vector to minimize the error function is to take a fixed step in the direction of greatest rate of decrease of the error, i.e., in the direction of the negative of the gradient $\nabla E(\mathbf{w})$. This gives rise to the technique of *gradient descent*. The minimization proceeds in a series of steps, with the weight values being updated at each step, and the derivatives being re-evaluated for each new set of weight values. Each weight (for both first and second layers) is updated using the expression

$$\Delta w_{ji}^{(\tau)} \equiv w_{ji}^{(\tau)} - w_{ji}^{(\tau-1)} = -\eta \left. \frac{\partial E}{\partial w_{ji}} \right|_{w^{(\tau)}}, \tag{26}$$

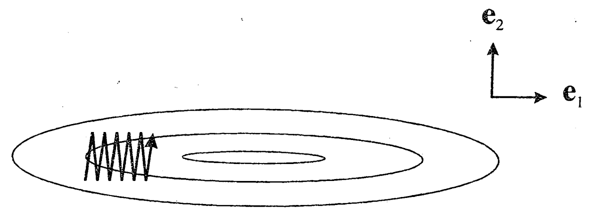

FIG. 12. Schematic illustration of the contours of a quadratic error surface in a 2-dimensional weight space in the neighborhood of a minimum, for which the curvature along the $\mathbf{e}_1$ direction is much less than the curvature along the $\mathbf{e}_2$ direction. Simple gradient descent, which takes successive steps in the direction of the negative of the error surface gradient, $\Delta\mathbf{w} = -\eta\nabla E$, suffers from oscillations across the direction of the valley if the value of the learning rate parameter η is too large.

where τ denotes the step number in the iteration, and the parameter η is called the *learning rate* and in the simplest scheme is set to a fixed value chosen by guesswork.

Provided the value of η is sufficiently small then Eq. (26) will lead to a decrease in the value of E (assuming the gradient is not already zero by virtue of the weight vector being at a minimum of E). Increasing the value of η can lead to a more substantial reduction of E at each step and thus can speed up the training process. However, too great a value for η can lead to instability. A further problem with this simple approach is that the optimum value for η will typically change with each step.

One of the main problems with simple gradient descent, however, arises when the error surface has a curvature along one direction $\mathbf{e}_1$ in weight space which is substantially smaller than the curvature along a second direction $\mathbf{e}_2$, as illustrated schematically in Fig. 12. The learning rate parameter then has to be very small in order to prevent divergent oscillations along the $\mathbf{e}_2$ direction, and this leads to very slow progress along the $\mathbf{e}_1$ direction for which the gradient is small. Ideally, the learning rate should be larger for components of the weight change vector along directions of low curvature than for directions of high curvature. One simple way to try to achieve this involves the introduction of a *momentum* term[18] into the learning equations. The weight update formula is modified to give

$$\Delta w_{ji}^{(\tau)} = -\eta \left. \frac{\partial E}{\partial w_{ji}} \right|_{w^{(\tau)}} + \mu\, \Delta w_{ji}^{(\tau-1)}, \tag{27}$$

where μ is a constant parameter in the range $0 \leq \mu < 1$ whose value is again set by hand. To see the effect of the momentum term consider a set of successive steps for which the gradient terms happen to be the same. We can then sum the resulting arithmetic series to give a combined step of the form

$$\Delta w_{ij}^{\mathrm{eff}} = -\frac{\eta}{1-\mu} \frac{\partial E}{\partial w_{ij}} \tag{28}$$

which corresponds to gradient descent with a much larger effective learning rate. In directions for which the curvature is low, the successive gradients will be similar and the effective learning rate will be increased. However, in directions of

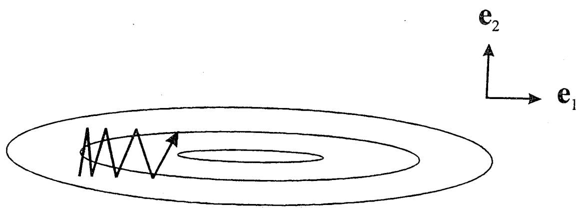

FIG. 13. As in Fig. 12, but showing the effect of introducing a momentum term, so that the weight updates are given by $\Delta \mathbf{w}^{(\tau)} = -\eta \nabla E + \Delta \mathbf{w}^{(\tau-1)}$, where τ denotes the iteration step number. This leads to an increase in the effective value of η in the direction of low curvatuve $\mathbf{e}_1$.

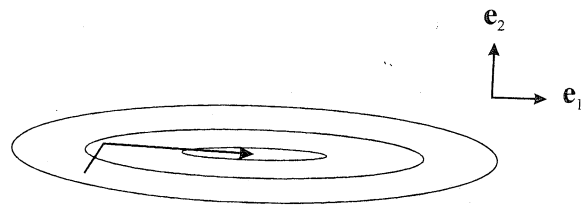

FIG. 14. As in Fig. 12, but showing the effect of using a conjugate gradients or a quasi-Newton algorithm, which can find the minimum of a quadratic error function exactly in a fixed number of steps, even though the curvatures of the error surface may be very different along different directions.

high curvature in which successive gradients have opposite signs (due to oscillations across the valley floor as indicated in Fig. 12) the successive contributions from the momentum term tend to cancel and so there is little change to the effective learning rate. The overall result is improved progress along the direction of the valley, as indicated in Fig. 13.

So far we have based our learning procedures on the total error function E. We can, however, also make use of the fact that E is a sum of terms, one for each pattern in the training set. Instead of accumulating the derivatives of E^q across all patterns and then changing the weights, we can make changes to the weights after presentation of each pattern separately to give

$$\Delta w_{ij}^{(\tau)} = -\eta \left. \frac{\partial E^q}{\partial w_{ij}} \right|_{w^{(\tau)}} + \mu\, \Delta w_{ij}^{(\tau-1)} . \qquad (29)$$

Note that the patterns are generally presented in a random order from the training set. This *pattern-based* or *sequential* version of gradient descent offers potential advantages when there is a degree of redundancy of the patterns in the training set. To see this, consider an extreme example in which each distinct pattern is replicated 10 times in the data set. The evaluation of the total error function or its gradient, as used by batch algorithms, will take 10 times as long. By contrast, pattern-based gradient descent updates the weight vector after presentation of each pattern and so will not be particularly affected by such redundancy. An additional advantage of the pattern-based approach is that the effective randomness introduced into the learning prescription can help in preventing the algorithm from becoming stuck in local minima. Note that the gradient descent procedure can be seen as a particular case of the technique of stochastic approximation.[33,34]

The term "backpropagation" is sometimes used to describe the particular training algorithm based on gradient descent (in which the derivatives are calculated as described earlier). In this review we have used the term to describe the evaluation of the derivatives of the error function, since it is here that errors are propagated backwards through the network. These derivatives can be used in a variety of training algorithms, of which gradient descent is only one. Similarly, backpropagation of "errors" can also be used to evaluate other derivatives such as the elements of the Hessian[32] and Jacobian[1] matrices.

Since the error functions for neural network training are highly non-linear, there is no simple prescription for deciding when to halt the training process. In practice, various stopping criteria are used, such as training for a fixed number of iterations, training for a given period of CPU time, training until the reduction in error over a given number of iterations falls below some threshold, and so on.

In many practical applications of neural networks, the network training process can be computationally intensive, requiring many hours of CPU time on fast workstations. Numerous modifications of the basic gradient descent algorithm have therefore been proposed with a view to improving its speed. Many of these are ad-hoc and do not always give consistent results across a range of applications. Rather than discuss such methods in detail, we turn instead to some more theoretically motivated approaches to network training.

D. Alternative training algorithms

As we have seen, the problem of training a neural network amounts to the determination of suitable values for a set of parameters $\mathbf{w}$ such as to minimize an error function $E(\mathbf{w})$, where the derivatives of E can readily be evaluated. This is a standard problem in non-linear optimization theory and a range of sophisticated techniques for solving it have been developed over many years.[35,36] Two of the most popular classes of technique are known respectively as *conjugate gradients* and *quasi-Newton* methods. Both of these approaches make use of line searches, that is constrained minimizations of the function along specific search directions.

The conjugate gradients algorithm chooses successive search directions so that minimization along the new direction does not "spoil" the minimization along previous directions.[35,37] For the particular case of an error function which is a general quadratic function of the weights (as would be the case for a linear network and a sum-of-squares error function) the procedure becomes exact, and the algorithm is guaranteed to find the minimum in $\mathcal{N}$ steps, where $\mathcal{N}$ is the number of weights. This is illustrated schematically for an error surface in two dimensions in Fig. 14.

By contrast, quasi-Newton methods use successive steps to build up an approximation to the inverse of the Hessian matrix.[35,36] The Hessian is the matrix of second derivatives of the error function,[32] and the minimum of the error function can be expressed (to lowest order in a Taylor expansion

in the neighborhood of the minimum) in terms of its inverse. For $\mathcal{N}$ weights, this requires $\mathcal{O}(\mathcal{N}^2)$ storage, which is generally not a problem except for very large networks having thousands of weights. Again, the algorithm will find the minimum of a quadratic error function exactly in a fixed number of steps, as in Fig. 14. One advantage of such algorithms is that the line searches do not need to be done as precisely as with conjugate gradients. The results plotted in Fig. 9 were obtained using the BFGS (Broyden-Fletcher-Goldfarb-Shanno) quasi-Newton method.[35] Algorithms have also been developed which try to combine the best features of conjugate gradient and quasi-Newton algorithms. One such algorithm is the limited memory BFGS method.[38,39]

Although the error surface may be far from quadratic, these methods are generally very robust, and typically give at least an order of magnitude improvement in convergence speed compared with simple gradient descent with momentum. A disadvantage is that they are somewhat more complex to implement in software than gradient descent, and also they have no built-in mechanism to escape from local minima. Also they are intrinsically batch (rather than sequential) methods, and so cannot deal effectively with redundancy in the training set, as discussed earlier. They can, however, prove particularly useful for problems where high precision is required as is the case in many instrumentation applications.

IV. RADIAL BASIS FUNCTION NETWORKS

One of the limitations of the multilayer perceptron is that the training process can be computationally very intensive. Since in practice, as we shall discuss later, it is necessary to repeat the training process many times in order to optimize the network architecture, this can become a significant problem. A further difficulty is that the internal representations formed by a trained multilayer perceptron can be hard to interpret. For any given input vector, the output is obtained by non-linear combinations of inputs involving all of the hidden units in a way which is generally very difficult to unravel. For most applications this is of little consequence, but occasionally it is useful to be able to interpret the activations of the hidden units. In this section we discuss an alternative architecture of neural network which, to some extent, overcomes these difficulties.

A. Structure of the radial basis function network

The radial basis function (RBF) network[40,41] is based on the simple intuitive idea that an arbitrary function $y(x)$ can be approximated as the linear superposition of a set of localized *basis functions* $\phi_j(x)$. This is indicated schematically in Fig. 15, and leads to a network structure which is, in many respects, similar to the multilayer perceptron.

The RBF network has its origins in techniques used for *exact* interpolation between data points in high dimensional spaces.[42,43] This is achieved by representing the outputs of the network as a linear superposition of basis functions, one for each data point in the training set, in the form

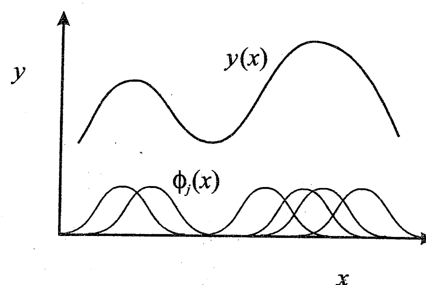

FIG. 15. Intuitively we expect that an arbitrary continuous function $y(x)$ can be approximated by a linear combination of localized bump-like functions $\phi_j(x)$. This concept leads to the radial basis function neural network model.

$$y_k=\sum_{q=1}^{n} \tilde{w}_{kq}\phi_q(\mathbf{x}), \tag{30}$$

where $\phi_q(\mathbf{x})$ is a radially symmetric function centered on the qth data point. There are many possible forms for the basis functions, of which a common choice is the Gaussian, given by

$$\phi_q(\mathbf{x})=\exp\left\{-\frac{|\mathbf{x}-\mathbf{x}^q|^2}{2\sigma^2}\right\}, \tag{31}$$

where the parameter σ controls the width of the Gaussian. Exact interpolation requires that the values of y_k reproduce the target values exactly, so that the parameters $\tilde{w}_{kq}$ are given by solution of the linear equations

$$y_k(\mathbf{x}^q)=t_k^q, \quad k=1,\ldots,n. \tag{32}$$

These represent n equations in n unknowns, and they generally have a unique solution provided the data points are not coincident. The resulting function, given by Eq. (30), then represents an interpolating function which is defined for all values of the input vector $\mathbf{x}$ and which passes exactly through each of the training points.

In applications of neural networks, however, we are not interested in exact interpolation but rather in finding a smooth representation of the underlying trends in the data. Indeed, an exact fit to the data, when the data are noisy as is the case in the majority of applications, can lead to particularly poor results when the trained network is presented with new data. This important phenomenon will be discussed at greater length in Sec. V. A smooth interpolation can be achieved by minimizing a sum-of-squares error function as before, using fewer basis functions than data points, so that the evaluation of the parameters $\tilde{w}$ becomes an overdetermined problem. This leads to the radial basis function neural network, which corresponds to a set of functions given by

$$y_k(\mathbf{x})=\sum_{j=0}^{m} \tilde{w}_{kj}\phi_j(\mathbf{x}), \tag{33}$$

where $m<n$ is the number of basis functions. Here $\phi_j(\mathbf{x})$ represents the activation of hidden unit j when the network is

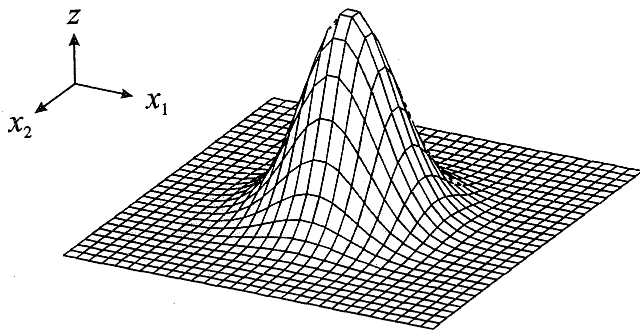

FIG. 16. Plot of the activation $z(x_1,x_2)$ of a Gaussian hidden unit as used in a radial basis function network, as a function of two input variables x_1 and x_2. This plot should be compared with the sigmoid shown in Fig. 8.

presented with input vector $\mathbf{x}$. Again, a bias for the output units has been included, and this has been represented as an extra "basis function" ϕ_0 whose activation is fixed to be $\phi_0=1$. For most applications the basis functions are chosen to be Gaussian, so that we have

$$\phi_j(\mathbf{x})=\exp\left\{-\frac{|\mathbf{x}-\boldsymbol{\mu}_j|^2}{2\sigma_j^2}\right\}, \tag{34}$$

where $\boldsymbol{\mu}_j$ is a vector representing the center of the jth basis function. Note that each basis function is given its own width parameter σ_j. A plot of the response of a Gaussian unit as a function of 2 input variables is shown in Fig. 16. Note that this is localized in the input space, unlike the ridge-like response of a sigmoidal unit shown in Fig. 8.

The RBF network can be represented by a network diagram as shown in Fig. 17. Each of the hidden units corresponds to one of the basis functions, and the lines connecting the inputs to hidden unit j represent the elements of the vector $\boldsymbol{\mu}_j$. Instead of a bias parameter, each unit now has a parameter σ_j which describes the width of the Gaussian basis function. The second layer of the network is identical to that of a multilayer perceptron in which the output units have linear activation functions.

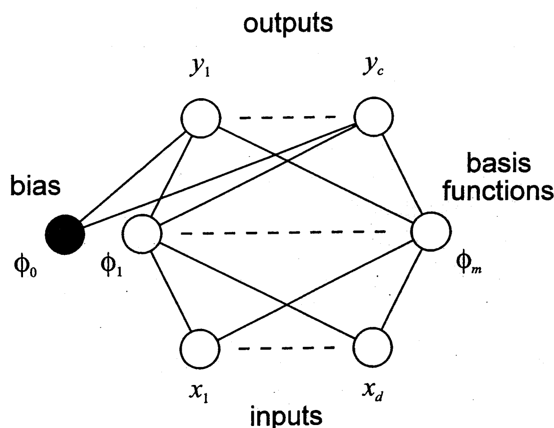

FIG. 17. Architecture of a radial basis function neural network having d inputs $x_1,...,x_d$ and c outputs $y_1,...,y_c$. Each of the m basis functions ϕ_j computes a localized (often Gaussian) function of the input vector. The lines connecting the inputs to the basis function ϕ_j represent the elements of the vector $\boldsymbol{\mu}_j$ which describes the location of the center (in input space) of that basis function. The second layer of the network, connecting the basis functions with the output units, is identical to that of the multilayer perceptron shown in Fig. 7. (From Ref. 1.)

Again, it can be shown formally that such a structure is capable of approximating essentially arbitrary continuous functions to arbitrary accuracy provided a sufficiently large number of hidden units (basis functions) is used and provided the network parameters (centers $\boldsymbol{\mu}_j$, widths σ_j, and second-layer weights $\tilde{w}_{kj}$) are suitably chosen.[44,45]

As with the multilayer perceptron, we seek a least-squares solution for the network parameters, obtained by minimizing a sum-of-squares error of the form given in Eq. (6). Since the network mapping is an analytic function of the network parameters, this could be done by simply optimizing all of the weights in the network together using one of the standard algorithms discussed earlier. Such an approach would, however, offer little advantage over the MLP network.

A much faster approach to training is based on the fact that the hidden units have a localized response, that is, each unit only produces an output which is significantly different from zero over a limited region of input space. This leads to a two-stage training procedure in which the basis function parameters ($\boldsymbol{\mu}_j$ and σ_j) are optimized first, and then, subsequently, the final-layer weights $\{\tilde{w}_{kj}\}$ are determined.

B. Choosing the basis function parameters

In the use of radial basis functions for exact interpolation, a basis function was placed over every data point. In the case of an RBF neural network we can adopt a similar strategy of placing basis functions in the regions of input space where the training data are located. Various heuristic procedures exist for achieving this, and we shall limit our discussion to two of the simplest. We shall also discuss a more systematic approach based on maximum likelihood.

The fastest and most straightforward approach to choosing the centers $\boldsymbol{\mu}_j$ of the basis functions is to set them equal to some subset (usually chosen randomly) of the input vectors from the training set. This only sets the basis function centers, and the width parameters σ_j must be set using some other heuristic. For instance, we can choose all the σ_j to be equal and to be given by the average distance between the basis function centers. This ensures that the basis functions overlap to some degree and hence give a relatively smooth representation of the distribution of training data. Such an approach to the choice of $\boldsymbol{\mu}_j$ and σ_j is very fast, and allows an RBF network to be set up very quickly. The subset of input vectors to be used as basis function centers can instead be chosen from a more principled approach based on *orthogonal least squares*,[46] which also determines the second-layer weights at the same time. In this case, the width parameters σ_j are fixed and are chosen at the outset.

A slightly more elaborate approach is based on the K-means algorithm.[47] The goal of this technique is to associate each basis function with a group of input pattern vectors, such that the center of the basis function is given by the mean of the vectors in the group, and such that the basis function center in each group is closer to each pattern in the group than is any other basis function center. In this way, the

data points are grouped into "clusters" with one basis function center acting as the representative vector for each cluster. This is achieved by an iterative procedure as follows. First, the basis function centers are initialized (for instance by setting them to a subset of the pattern vectors). Then each pattern vector is assigned to the basis function with the nearest center $\boldsymbol{\mu}_j$, and the centers are recomputed as the means of the vectors in each group. This process is then repeated, and generally converges in a few iterations. Again, it only sets the centers, and the width parameters must be set using a technique of the kind described above.

A more principled approach to setting the basis function parameters is based on the technique of maximum likelihood. Let us define $p(\mathbf{x})$ to be the (unknown) probability density function for the input data, so that the probability of a new input vector falling within a small volume $\Delta\mathbf{x}$ of input space is given by $p(\mathbf{x})\Delta\mathbf{x}$, and $\int p(\mathbf{x})\, d\mathbf{x}=1$. The idea is to use the basis functions to form a representation for $p(\mathbf{x})$, and to determine the parameters of the basis functions by using the input vectors from the training set. The probability density is expressed as a linear combination of the basis functions in the form of a *Gaussian mixture model*[48]

$$p(\mathbf{x})=\frac{1}{m}\sum_{j=1}^{m}\frac{1}{(2\pi)^{d/2}\sigma_j^d}\,\phi_j(\mathbf{x}), \tag{35}$$

where the prefactor in front of $\phi_j(\mathbf{x})$ is chosen to ensure that the probability density function integrates to unity: $\int p(\mathbf{x})\, d\mathbf{x}=1$. If the input vectors from the training set are drawn independently from this distribution function, then the *likelihood* of this data set is given by the product

$$\mathscr{L}=\prod_{q=1}^{n} p(\mathbf{x}^q). \tag{36}$$

The basis function parameters can then be set by maximizing this likelihood. Since the likelihood is an analytic non-linear function of the parameters $\{\boldsymbol{\mu}_j,\sigma_j\}$, this maximization can be achieved by standard optimization methods (such as the conjugate gradients and quasi-Newton methods described earlier). It can also be done using re-substitution methods based on the *EM-algorithm.*[49] Such methods are relatively fast and allow values for the parameters $\{\boldsymbol{\mu}_j,\sigma_j\}$ to be obtained reasonably quickly. In contrast to the MLP, the hidden units in this case have a particularly simple interpretation as the components in a mixture model for the distribution of input data. The sum of their activations (suitably normalized) then provides a quantitative measure of $p(\mathbf{x})$, which can play an important role in validating the outputs of the network.[3]

C. Choosing the second-layer weights

We shall suppose that the basis function parameters (centers and widths) have been chosen and fixed. As usual, the sum-of-squares error can be written as

$$E=\frac{1}{2}\sum_{q=1}^{n}\sum_{k=1}^{c}\{y_k^q-t_k^q\}^2. \tag{37}$$

We note that, since y_k is a linear function of the final layer weights, E is a quadratic function of these weights. Substituting Eq. (33) into Eq. (37), we can minimize E with respect to these weights explicitly by differentiation, to give

$$0=\sum_{q=1}^{n}\phi_j^q\left\{\sum_{j'=1}^{m}\tilde{w}_{kj'}\phi_{j'}^q-t_k^q\right\}, \tag{38}$$

where $\phi_j^q\equiv\phi_j(\mathbf{x}^q)$. It is convenient to write this in matrix notation in the form

$$0=\mathbf{\Phi}^T\{\mathbf{\Phi}\mathbf{W}^T-\mathbf{T}\}, \tag{39}$$

where the matrices have the following elements: $\mathbf{\Phi}=(\phi_j^q)$, $\mathbf{W}=(\tilde{w}_{kj})$, and $\mathbf{T}=(t_k^q)$. The notation $\mathbf{M}^T$ denotes the transpose of a matrix $\mathbf{M}$. This equation has a formal solution for the weights given by

$$\mathbf{W}^T=\mathbf{\Phi}^\dagger\mathbf{T}, \tag{40}$$

where $\mathbf{\Phi}^\dagger$ is the *pseudo-inverse*[50] of the matrix $\mathbf{\Phi}$ and is given by

$$\mathbf{\Phi}^\dagger\equiv(\mathbf{\Phi}^T\mathbf{\Phi})^{-1}\mathbf{\Phi}^T. \tag{41}$$

(Note that this formula for the pseudo-inverse assumes that the relevant inverse matrix exists. If it does not, then the pseudo-inverse can still be uniquely defined by an appropriate limiting process.[50]) In a practical implementation, the weights are found by solving the linear equations (39) using singular value decomposition[35] to allow for possible numerical ill-conditioning. Thus the final layer weights can be found explicitly in closed form. Note, however, that the optimum value for these weights, given by Eq. (40), depends on the values of the basis function parameters $\{\boldsymbol{\mu}_j,\sigma_j\}$, via the quantities ϕ_j^q. Once these parameters have been determined, the second-layer weights can then be set to their optimal values.

Note that the matrix $\mathbf{\Phi}$ has dimensions $n\times m$ where n is the number of patterns, and m is the number of hidden units. If there is one hidden unit per pattern, so that $m=n$, then the matrix $\mathbf{\Phi}$ becomes square and the pseudo-inverse reduces to the usual matrix inverse. In this case the network outputs equal the target values exactly for each pattern, and the error function is reduced to zero. This corresponds precisely to the exact interpolation method discussed above. As we shall see later, this is generally not a desirable situation, as it leads to the network having poor performance on unseen data, and in practice m is typically much less than n. The crucial issue of how to optimize m will be discussed at greater length in the next section.

V. LEARNING AND GENERALIZATION

So far we have discussed the representational capabilities of two important classes of neural network model, and we have shown how network parameters can be determined on the basis of a set of training data. As a consequence of the great flexibility of neural network mappings, it is often easy to arrange for the network to represent the training data set

with reasonable accuracy, particularly if the size of the data set is relatively small. A much more important issue, however, is how well does the network perform when presented with new data which did not form part of the training set. This is called *generalization* and is often much more difficult to achieve than simple memorization of the training data.

A. Interpretation of network outputs

We begin our discussion of generalization in neural networks by considering an ideal limit in which an infinite amount of training data is available. This allows us to replace the finite sum in the sum-of-squares error function by an integral over the (smooth) probability density function of the data. The sum-of-squares error (6) can then be written as

$$E=\frac{1}{2}\sum_{k=1}^{c}\int \{y_k(\mathbf{x})-t_k\}^2\, p(t_k|\ \mathbf{x})\, p(\mathbf{x})\, dt_k\, d\mathbf{x}, \qquad (42)$$

where $p(t_k|\mathbf{x})$ denotes the conditional probability density of the target data for output unit k, given a value $\mathbf{x}$ for the input vector, and $p(\mathbf{x})$ denotes the probability density of input data as before. The process of network training corresponds to an attempt to adjust the network function $y(\mathbf{x})$ so as to minimize E. If we assume that the network has unlimited flexibility to generate different functions, then we can formally minimize E in Eq. (42) by functional differentiation

$$\frac{\delta E}{\delta y_{k'}(\mathbf{x}')}=0=\sum_{k=1}^{c}\int \{y_k(\mathbf{x})-t_k\}\,\delta(\mathbf{x}-\mathbf{x}')$$
$$\times\,\delta_{kk'}p(t_k|\mathbf{x})p(\mathbf{x})dt_k d\mathbf{x}. \qquad (43)$$

This is easily solved [using $\int p(t_k|\mathbf{x})\, dt_k=1$] to give the minimizing network function in the form

$$y_k(\mathbf{x})=\langle t_k|\mathbf{x}\rangle\equiv\int t_k p(t_k|\mathbf{x})dt_k\ , \qquad (44)$$

where $\langle t_k|\mathbf{x}\rangle$ denotes the conditional average of the target data for a specified value of the input vector $\mathbf{x}$, and is known as the *regression*. This result was derived without reference to neural networks, and applies in principle to any class of models which can represent general functions $y(\mathbf{x})$. The importance of neural networks is that they represent a very flexible class of functions and so in principle can provide a good approximation to the optimal function $\langle t|x\rangle$.

To illustrate the meaning of this important result, consider a network having one input x and one output y. Figure 18 shows a schematic illustration of the way in which the network function $y(x)$ is determined, at each value of x, by averaging over the distribution of the target data. Suppose that the target data are generated from some smooth deterministic function $h(x)$ to which is added zero mean noise

$$t=h(x)+\epsilon. \qquad (45)$$

A network trained on such data will generate an output which, from Eq. (44), will be given by

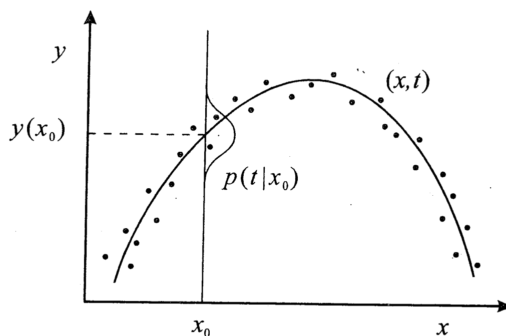

FIG. 18. Schematic illustration of some noisy data points (the black dots) each of which consists of a value of the input variable x together with a corresponding target value t. The curve shows the optimal network function $y(x)$ obtained by minimizing a sum-of-squares error function. For any given value x_0 of the input, the function $y(x_0)$ is given by the mean of t with respect to the conditional probability distribution $p(t|x_0)$. This central result, which is easily extended to the case of several input and several output variables, has a number of important consequences, as discussed in the text. (From Ref. 1.)

$$y(x)=\langle t|x\rangle=h(x) \qquad (46)$$

since $\langle\epsilon\rangle=0$. The network therefore averages over the noise on the target data and learns the underlying deterministic function. In this sense, the network mapping can be regarded as optimal.

The result (44) has several other important implications, one of which concerns the application of neural networks to classification problems. It is convenient for such applications to make use of a "1-of-N" coding scheme for the target data as follows. Suppose there are c possible *classes* $\mathscr{C}_k$ $(k=1,\ldots,c)$ to which an input vector could be assigned. In a medical screening application, for example, we may wish to assign an x-ray image (described by a vector of pixel intensities $\mathbf{x}$) to one of the two classes $\mathscr{C}_1\equiv$"normal," and $\mathscr{C}_2\equiv$"tumor." We construct a network having c output units, and we choose target values for the outputs such that, for an input vector belonging to class l, all outputs have a target of 0, except for output l which has a target of 1. If the data has a probability $P(\mathscr{C}_k|\mathbf{x})$ of belonging to class $\mathscr{C}_k$ when the input vector is $\mathbf{x}$ then the probability density of the target data (which now consists of 0's and 1's) becomes

$$p(t_k|\ \mathbf{x})=\sum_{l=1}^{c}\delta(t_k-\delta_{lk})P(\mathscr{C}_l|\mathbf{x}). \qquad (47)$$

Substituting Eq. (47) into Eq. (44) we obtain the network outputs in the form

$$y_k(\mathbf{x})=P(\mathscr{C}_k|\mathbf{x}). \qquad (48)$$

This says that the network outputs will represent the Bayesian *a-posteriori* probabilities of class membership.[51,52] The fact that the network outputs can be given a precise probabilistic interpretation has several important practical consequences. For instance, it tells us that when we present a new input vector to the network it should be assigned to the class having the largest output activation, as this minimizes the probability of misclassification.[53–56] In addition it allows

other quantities (called loss criteria) other than misclassification rate to be minimized. This is important if different misclassifications have different consequences and should therefore carry different penalties.[57] It also provides a principled way to combine the outputs of different networks to build a modular solution to a complex problem. These topics are discussed further in Refs. 1 and 51.

B. Generalization

The above analysis made two central assumptions: (i) there is an infinite supply of training data, (ii) the network has unlimited flexibility to represent arbitrary functional forms. In practice we must inevitably deal with finite data sets and, as we shall see, this forces us to restrict the flexibility of the network in order to achieve good performance. By using a very large network, and a small data set, it is generally easy to arrange for the network to learn the training data reasonably accurately. It must be emphasized, however, that the goal of network training is to produce a mapping which captures the underlying trends in the training data in such a way as to produce reliable outputs when the network is presented with data which do not form part of the training set. If there is noise on the data, as will be the case for most practical applications, then a network which achieves too good a fit to the training data will have learned the details of the noise on that particular data set. Such a network will perform poorly when presented with new data which do not form part of the training set. Good performance on new data, however, requires a network with the appropriate degree of flexibility to learn the trends in the data, yet without fitting to the noise.

These central issues in network generalization are most easily understood by returning to our earlier analogy with polynomial curve fitting. In particular, consider the problem of fitting a curve through a set of noise-corrupted data points, as shown earlier for the case of a cubic polynomial in Fig. 5. The results of fitting polynomials of various orders are shown in Fig. 19. If the order m of the polynomial is too low, as indicated for $m=1$ in Fig. 19(a), then the resulting curve gives only a poor representation of the trends in the data. When the value of y is predicted using new values of x the results will be poor. If the order of the polynomial is increased, as shown for $m=3$ in Fig. 19(b), then a much closer representation of the data trend is obtained. However, if the order of the polynomial is increased too far, as shown in Fig. 19(c), the phenomenon of *overfitting* occurs which gives a very small (in this case zero) error with respect to the training data, but which again gives a poor representation of the underlying trend in the data and which therefore gives poor predictions for new data. Figure 20 shows a plot of the sum-of-squares error versus the order of the polynomial for two data sets. The first of these is the training data set which is used to determine the coefficients of the polynomial, and the second is an independent *test* set which is generated in the same way as the training set, except for the noise contribution which is independent of that on the training data. The test set therefore simulates the effects of applying new data to the "trained" polynomial. The order of the polynomial controls the number of degrees of freedom in the function,

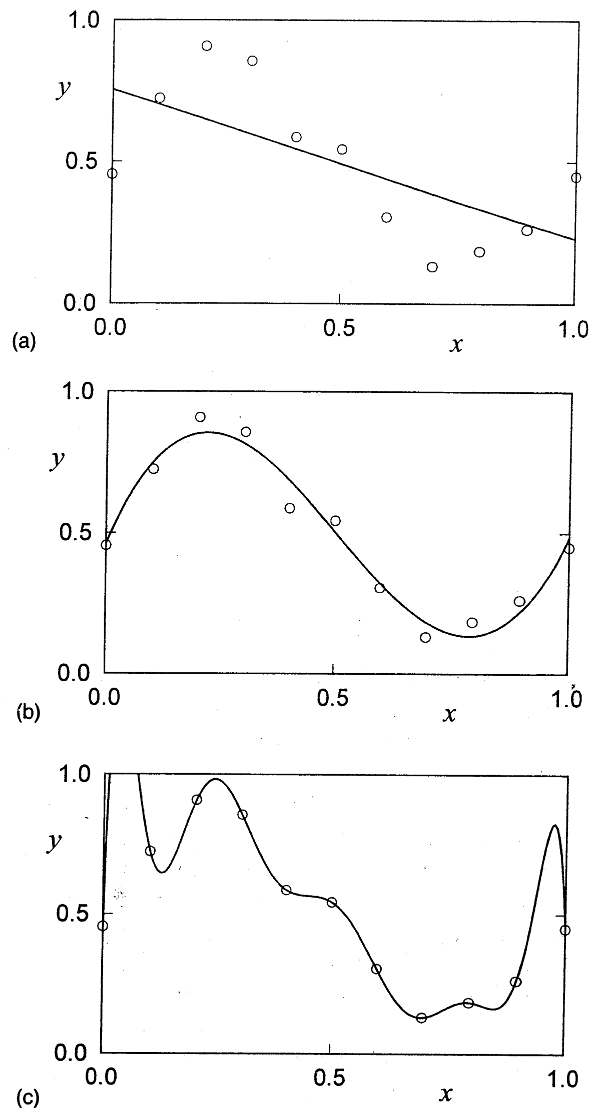

FIG. 19. Examples of curve fitting using polynomials of successively higher order, using the same data as was used to plot Fig. 5. (a) was obtained using a first order (linear) polynomial, and is seen to give a rather poor representation of the data. By using a cubic polynomial, as shown in (b), a much better representation of the data is obtained. (This figure is identical to Fig. 6, and is reproduced here for ease of comparison.) If a 10th order polynomial is used, as shown in (c), a perfect fit to the data is obtained (since there are 11 data points and a 10th order polynomial has 11 degrees of freedom). In this case, however, the large oscillations which are needed to fit the data mean that the polynomial gives a poor representation of the underlying generator of the data, and so will make poor predictions of y for new values of x. (From Ref. 1.)

and we see that there is an optimum number of degrees of freedom (for a particular data set) in order to obtain the best performance with new data.

A similar situation occurs with neural network mappings. Here the weights in the network are analogous to the coefficients in a polynomial, and the number of degrees of freedom in the network is controlled by the number of weights, which in turn is determined by the number of hidden units. (Note that the *effective* number of degrees of freedom in a neural network is generally less than the number of weights and biases. For a discussion see Ref. 58.) Again we can consider two independent data sets which we call *training* and *test* sets. We can then use the training data to train

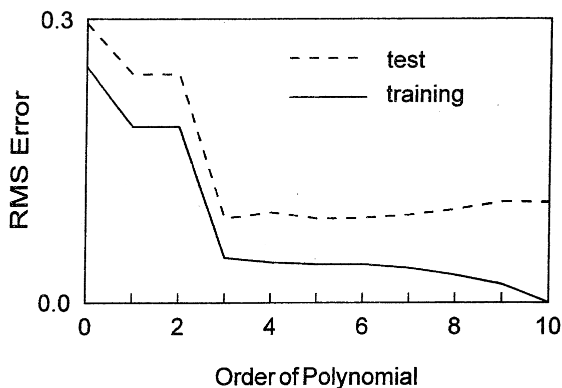

FIG. 20. A plot of the residual value of the root-mean-square error for polynomial curve fitting versus the order of the polynomial. Here the training data are the same as used to plot Fig. 19, and are the data used to determine the coefficients of the polynomial by minimizing the sum-of-squares error. The residual rms error for the training data is seen to decrease monotonically as the order of the polynomial is increased, eventually falling to zero for the 10th order polynomial which fits the training data exactly, as shown in Fig. 19(c). The test data set is generated in the same way as the training data, and also consists of 11 points with the same x values, but with different values for the random additive noise. It is seen that the error on the test data (which measures the ability of the polynomial to "generalize") is smallest for a cubic polynomial. For polynomials with more degrees of freedom than a cubic, the error for the test data is actually larger, even though the training data error is smaller. (From Ref. 1.)

several networks, differing in the number m of hidden units, and plot a graph of the residual value of the error E after training as a function of m. This would be expected to yield a monotonic decreasing function, as indicated schematically in Fig. 21, since the addition of extra degrees of freedom should not result in any increase in error, and will generally allow the error to be smaller. We can also present the test set data to the trained networks, and evaluate the corresponding values of E. These would be expected to show a decrease with m at first as the network acquires greater flexibility, but then start to increase as the problem of overfitting sets in. A common beginners mistake in applying neural networks is to use too large a network and thereby obtain apparently very good results (small training error at large values of m in Fig. 21). We see, however, that this typically leads to very poor

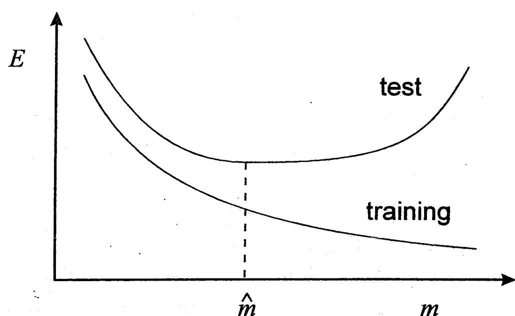

FIG. 21. A schematic plot of the residual error with respect to the training set, and the error with respect to a separate test set, as a function of the number m of hidden units in a neural network. As with polynomial curve fitting, there is an optimum number of hidden units (shown here by $m=\hat{m}$) which gives the smallest test set error, and hence the best generalization performance.

performance on new data, corresponding to large values of the test set error in Fig. 21.

A network function which has too little flexibility is said to have a large *bias*, while one which fits the noise on the data is said to have a large *variance*. One of the main goals in applying a neural network is to achieve a good tradeoff between bias and variance.[59] Instead of restricting the number of weights in the network, an alternative approach to controlling bias and variance is to add penalty terms to the error function to encourage the network mapping to have appropriate smoothness properties. This is called *regularization*, and a detailed discussion lies beyond the scope of this review.[60–65]

C. Determination of network topology

In almost all applications, the goal of network training is to find a network mapping function which makes the best possible predictions for new data. This corresponds to the network having the minimum test error, given by $m=\hat{m}$ in Fig. 21. It is this requirement which drives the selection of the network topology, in other words the number of hidden units for the networks considered in this review.

In a practical application, the simplest approach to optimizing the number of hidden units is to partition the available data at random into a training and a test set and then to plot a graph of the form shown in Fig. 21. The best network of those trained is then determined by the minimum in the test set error. In a practical application the curves which are obtained from such an exercise do not always exhibit precisely this behavior. This is a consequence of the fact that network training corresponds to a non-linear optimization problem which can suffer from local minima, as already described in Sec. III. In addition, the effects of using a finite size test set also lead to departures from the smooth behavior depicted in Fig. 21. Thus, the training error curve might not decrease monotonically, or the test error curve might have several minima. An example of the curves obtained with real data is shown in Fig. 22. Since the test set has itself been used as part of the network optimization process, the final performance of the network should, strictly speaking, be checked against a third independent set of data.

A more sophisticated approach, and one which is particularly useful when the quantity of available data is limited, is that of *cross-validation*.[66] Here the data set is partitioned randomly into S equal sized sections. Each network is trained on $S-1$ of the sections and its performance tested on the remaining section, which acts like a test set. This is repeated for the S possible choices for the section omitted from training, and the results are then averaged. This is repeated for all of the networks under consideration. The network having the smallest error on data not used for training is then selected. In effect, all of the data is used for both training and testing. The disadvantage of this approach is its greater computational demands. Again, a third independent data set should be used to confirm the final performance of the selected network. In practice, a value of $S=10$ is a typical choice, although if data are in very limited supply, a value of $S=1$ can be used, giving rise to the procedure known as *leave one out.*

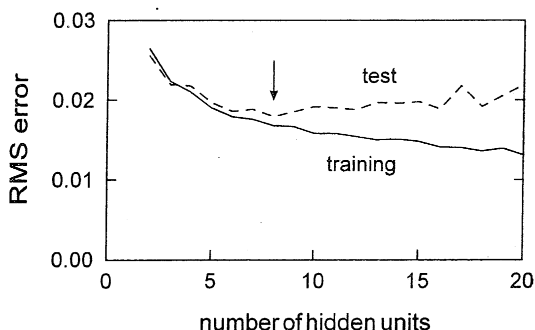

FIG. 22. An example of the optimization of the topology of a network taken from a real application (the determination of oil fraction in multi-phase flows using gamma densitometry, discussed in Sec. VIII). Here the root-mean-square error is plotted as a function of the number of hidden units for a training set consisting of 1000 examples and a test set also of 1000 examples. These networks had 12 inputs and 2 linear outputs and were trained for 300 cycles of the limited memory BFGS algorithm. Since the error function for the neural network is a complicated non-linear function of the network parameters, the global minimum of the error is not generally obtained, and the training set error does not decrease in a strictly monotonic fashion. Similarly, the test set error is seen to be a rather "noisy" function of the number of hidden units. The arrow shows the minimum point of the test error curve, corresponding to the network with 8 hidden units. (From Ref. 67.)

Since the process of training a neural network can be somewhat involved we now summarize the various stages in the process. To be specific we shall consider a multilayer perceptron network.

(1) Select a value for the number of hidden units in the network, and initialize the network weights using random numbers.
(2) Minimize the error defined with respect to the training set data using one of the standard optimization algorithms such as conjugate gradients. The derivatives of the error function are obtained using the backpropagation algorithm as described in Sec. III.
(3) Repeat the training process a number of times using different random initializations for the network weights. This represents an attempt to find good minima in the error function. The network having the smallest value of residual error is selected.
(4) Test the trained network by evaluating the error function using the test set data.
(5) Repeat the training and testing procedure for networks having different numbers of hidden units and select the network having smallest test error.

For radial basis function networks, only the training procedure is different. The problem of optimizing the number of hidden units (basis functions) is similar and can be tackled using the same techniques.

VI. DATA PREPROCESSING

So far we have discussed network mappings from input to output variables, without specifying explicitly how these variables are related to the training data. In principle the input and target data for training could consist of raw data taken directly from the application. However, such a simple approach will, in many cases, lead to poor performance, and

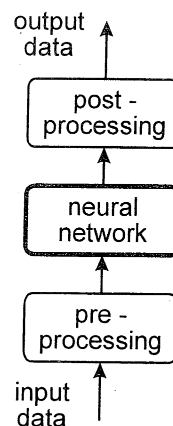

FIG. 23. Raw data are generally preprocessed before being presented to a neural network. Similarly, the output variables from a network are often post-processed in order to obtain physically meaningful quantities. To train the network, the input patterns from the training data must be transformed using the preprocessing stage in order to obtain appropriate values for the inputs to the network. Similarly, the target data from the training set must be transformed using the inverse of the post-processing stage in order to obtain the correct targets for the network outputs.

one of the most important factors in achieving a successful application of neural networks is the use of appropriate data preprocessing and representation.

Preprocessing generally takes the form of transformations applied to the input data before they are presented to the network. Similarly, the outputs of the network may be post-processed to convert them to physically meaningful quantities. This is illustrated in Fig. 23. In order to train such a network, the input data from the training set must first be preprocessed. Similarly, the inverse of the post-processing transformation must be applied to the target data in order to map it to the correct form for training. When new data are presented to the trained network, the preprocessing transformation must be applied to the input data, and the post-processing transformation must be applied to the network outputs.

Since a neural network can approximate arbitrary functional transformations, it is not immediately obvious why preprocessing should improve network performance. The pre- and post-processing functions indicated in Fig. 23, together with the intermediate network, could in principle be represented by a single neural network mapping. We therefore need to examine why, in practice, preprocessing can be so crucial to success.

A. The curse of dimensionality

It is clear that any form of processing applied to the input vectors of a data set cannot increase the amount of useful information present from which to predict the correct values for the outputs. (An exception to this is when preprocessing is used to incorporate additional knowledge about the general form of the desired solution, as will be discussed later.) However, in the early days of pattern recognition it was discovered that, for example, simply discarding some of the input variables could actually lead to improved generalization ability. This paradoxical result can be understood in terms of the scaling properties of a pattern recognition prob-

lem with the dimensionality of the space of input variables.

Consider a d-dimensional input space, and suppose that the region of interest corresponds to the unit hypercube $\mathbf{x} \in [0,1]^d$. We can specify the value of any one of the input variables x_i by dividing the corresponding axis into N segments and stating in which segment the value of the variable lies. As N increases, so we can specify the variable with increasing precision. With each variable specified in this way, the unit hypercube has been subdivided into N^d small hypercubes. In general, to specify a mapping from the input space to a single output variable, we must provide N^d items of information, representing the value of the output for the corresponding input hypercube. Thus, the size of the training set required to specify a mapping would in general grow exponentially with the dimensionality of the input space. This phenomenon is known as the *curse of dimensionality*.[68]

In practice, there are two reasons why the amount of data needed may be much less than this argument would suggest. First, correlations between the input variables mean that the data are effectively confined to a sub-space of the input space which might have much lower effective dimensionality. Adding extra input variables which are strongly correlated with the existing inputs would not lead to a significant increase in the effective dimensionality of the space occupied by the data. Second, there is generally significant structure in the data so that, for instance, the output variable may vary smoothly with the input variables. Thus, knowledge of the outputs for several input vectors allows the outputs for new inputs to be predicted by an interpolation process. It is this second property of data sets which makes generalization possible.

Notwithstanding these two mitigating effects, the quantity of data needed to specify a mapping can still grow rapidly with dimensionality. As a result, when tackling a practical problem involving a finite size data set, the performance of a neural network system can actually improve when the input dimensionality is reduced by preprocessing even though information may be lost in the dimensionality reduction process. The fixed quantity of data is better able to specify the mapping in the lower dimensional space, and this can more than compensate for the loss of information.

Dimensionality reduction plays a particularly important role in problems for which the input dimensionality is large. In applications such as the interpretation of images, for instance, the number of pixels may be many thousands. Direct presentation of the data to a network having large numbers of input units (one per pixel) and consequently large numbers of degrees of freedom will typically give very poor results.

Note that there is an additional benefit from dimensionality reduction in the form of reduced training times, arising from the fact that there are now fewer adjustable parameters in the network (since the number of input units has been reduced, and so there are fewer weights in the first layer).

B. Linear rescaling

In addition to reducing dimensionality, there are other motivations for preprocessing the data. One of the most common forms of preprocessing involves a simple linear rescaling of the input variables, and possibly also of the output variables. In a typical application the different input variables may represent very different physical quantities. For instance, one input might represent a magnetic field value, and be $\mathscr{O}(1)$, while another might represent a frequency, and be $\mathscr{O}(10^{12})$. This would typically cause difficulties in network training, since the optimal values for the weights would need to span a range of $\mathscr{O}(10^{12})$, and it would prove difficult to discover such a solution using standard training algorithms. The problem can be resolved by performing a linear rescaling of the data to ensure that each input variable has zero mean and unit standard deviation over the training set. Thus the inputs to the network $\tilde{x}_i$ are obtained from the raw input data x_i by the following transformation:

$$\tilde{x}_i = \frac{\{x_i - \bar{x}_i\}}{s_i}. \tag{49}$$

If we set

$$\bar{x}_i = \frac{1}{n} \sum_{q=1}^{n} x_i, \quad s_i^2 = \frac{1}{n-1} \sum_{q=1}^{n} \{x_i - \bar{x}_i\}^2 \tag{50}$$

then the rescaled inputs will all have zero mean and unit variance with respect to the training data set. Once the network is trained, the same rescaling (49), using the same values for the coefficients, must be applied to all future data presented to the network. A similar rescaling is often applied also to the target data for interpolation problems, and this rescaling must be inverted to post-process data obtained from the trained network. The rescaling in Eq. (49) treats the input variables as independent. A more sophisticated linear rescaling, known as *whitening*,[54] takes account also of correlations between the input variables.

C. Feature extraction

The simple rescaling of input variables described above represents an invertable transformation in which no information is lost. As we have already indicated, however, in many applications involving large numbers of input variables it can be very advantageous to reduce the dimensionality of the input vector, even though this represents a non-invertable process in which the amount of available information is potentially diminished.

One of the simplest approaches to dimensionality reduction is to discard a subset of the input variables. Techniques for doing this generally involve finding some ranking of the relative importance of different inputs and then omitting the least significant. In principle, the relative importance of the input variables depends on what kind of mapping function will be employed, and so strictly a neural network should be trained for each possible subset of the input variables. Since in practice this is likely to be computationally prohibitive, a simpler system which can be trained very quickly (such as a linear model) is often used to order the inputs. An appropriate subset is then used for training the more flexible nonlinear network.

Other forms of dimensionality reduction make no use of the target data but simply look at the statistical properties of

the input data alone. The most common such technique is *principal components analysis*[69] in which a linear dimensionality reducing transformation is sought which maximizes the variance of the transformed data. While easy to apply, such techniques run the risk of being significantly suboptimal since they take no account of the target data.

More generally, the goal of preprocessing is to find a number of (usually non-linear) combinations of the input variables, known as *features*, which are designed to make the task of the neural network as easy as possible. By selecting fewer features than input variables, a dimensionality reduction is achieved. The optimum choice of features is very problem dependent, and yet can have a strong influence on the final performance of the network system. It is here that the skill and experience of the developer count a great deal.

D. Prior knowledge

One of the most important, and most powerful, ways in which the performance of neural network systems can be improved is through the incorporation of additional information, known as *prior knowledge*, into the network development and training procedure, in addition to using the information provided by the training data set. Prior knowledge can take many forms, such as invariances which the network transformation must respect, or expected frequency of occurrences of different classes in a classification problem.

One way of exploiting prior knowledge is to build it into the data preprocessing stage. If the desired outputs from the network are known to be invariant under some set of transformations, then features can be extracted which exhibit this property, thereby ensuring that the network outputs will automatically show the same behavior. The technique of regularization, discussed in Sec. V, also implicitly involves incorporating prior knowledge into the network training process. For example, a regularization term which penalizes high curvature in the network mapping function reflects prior knowledge that the function should be smooth.[65] Prior knowledge can also be used to configure the topology of the network itself. For instance, the postal code recognition system described in Ref. 30 uses a system of local "receptive fields" with "shared weights" to achieve approximate invariance to translations of the characters within the input image.

The inclusion of explicit invariance to some set of transformations is an important use of preprocessing. It leads to 3 significant advantages compared with having the network learn the invariance property by example: (i) the invariance property is satisfied exactly, whereas it would only be learned approximately from examples; (ii) a smaller training set can be used since any set of patterns which differ only by the transformation can be represented by a single pattern in the training set; (iii) the network is able to extrapolate to new input vectors provided these differ from the training data primarily by virtue of the invariant transformation.

We shall describe some further examples of the use of prior knowledge when we review a number of case studies in Sec. VIII.

VII. IMPLEMENTATION OF NEURAL NETWORKS

So far we have discussed neural networks as abstract mathematical functions. In a practical application, it is necessary to provide an implementation of the neural network. At present, the great majority of research projects in neural networks, as well as most practical applications, makes use of simulations of the networks written in conventional software and running on standard computer platforms. While this is adequate for many applications, it is also possible to implement networks in various forms of special-purpose hardware. This takes advantage of the intrinsic parallelism of neural network models and can lead to very high processing speeds. We begin, however, with a discussion of software implementation.

A. Software implementation

Most applications of neural networks use software implementations written in high level languages such as C, PASCAL, and FORTRAN. The neural network algorithms themselves are generally relatively straightforward to implement, and much of the effort is often devoted to application-specific tasks such as data preprocessing and user interface. Neural networks are well suited to implementation in object oriented languages such as C++, which allow a network to be treated as an object, with methods to implement the basic operations of forward propagation, saving and retrieving weight vectors, etc.

There are now numerous neural network software packages available, ranging from simple demonstration software provided on disk with introductory books, through to large commercial packages supporting a range of network architectures and training algorithms and having sophisticated graphical interfaces. The latter kind of software can be very useful for quick prototyping, and provides an easy way to gain hands-on experience with neural networks without requiring a heavy investment in software development. It is important to emphasize, however, that such software cannot be treated as a black box solution to problems since, as we have seen, there are numerous subtle issues which must be addressed if satisfactory performance is to be obtained. Some of these, such as the incorporation of prior knowledge, can sometimes be highly problem-specific, and do not readily lend themselves to inclusion in commercial software. Also, the fact that such software does not usually provide direct access to source code can significantly limit its applicability to complex real-world problems.

B. Hardware implementation

One of the potential advantages of neural network techniques compared with many conventional alternatives is that of speed. There are in fact two quite distinct reasons why neural networks can prove to be significantly faster than conventional methods. The first applies to software simulations as well as hardware implementations and stems from the fact that, once trained, a neural network operating in feedforward mode can perform a multivariate non-linear transformation in a fixed (and generally very small) number of operations. This contrasts with many conventional techniques which

achieve high speed at the expense of restriction to linear transformations, or which solve non-linear problems by means of iterative, and hence computationally intensive, approaches. Of course, it should be remembered that the process of training a neural network can be computationally intensive and slow, although for many applications training is performed only during the development phase, with the network being used as a feedforward system when processing new data.

The second reason why neural networks can give very high processing speeds is that they are intrinsically highly parallel systems and so can be implemented in special-purpose parallel hardware. This gives an additional increase in speed in addition to that resulting from the feedforward nature of the network mapping.

Even with serial processor hardware, it is possible to exploit the structure of the neural network mapping to improve processing speed. For instance, many DSP (digital signal processing) and workstation processors can perform vector-matrix operations very efficiently. Since the number of weights in a typical network is generally much larger than the number of nodes, the dominant contribution to the computation comes from the product-and-sum stages rather than from the evaluation of activation functions, and this is essentially a vector-matrix operation. The resulting improvements in processing speed apply to the training phase of the network as well as to its subsequent use in feedforward mode.

It is also relatively straightforward to implement neural networks on arrays of processors such as transputers, or even a network of workstations. During training, the network can be replicated on all of the processors which then each deal with a subset of the patterns. Alternatively, different parts of the network (for instance successive layers) can be assigned to different processors, which then operate in a pipeline fashion. This relative ease of parallel implementation should be contrasted with the often severe problems of making efficient use of multiple processors with many conventional methods. If a conventional algorithm relies at some point on a single serial calculation before the rest of the algorithm can proceed, then only one processor can perform this step while the other processors remain idle.

For very high processing speeds a network can be implemented in special-purpose hardware in which the various components of the network (weights and nodes) are mapped directly into elements of the hardware system. A flexible modular implementation of the multilayer perceptron, built from conventional surface mount technology in a VME rack system,[70] was recently used successfully for real-time feedback control of a tokamak plasma.[71] This system used multiplying DAC's (digital to analogue converters) acting as digitally-set resistors to provide the weights and biases, and temperature-compensated transistor circuits to implement the non-linear sigmoids. This gives a system which has analogue signal paths but in which the synaptic weights can be set digitally, allowing the weights to be specified to reasonably high precision.

Much of the research on hardware implementations of neural networks focuses on VLSI (Very Large Scale Integration) techniques, and these can be broadly divided into digital and analogue approaches. Digital systems make use of highly developed silicon fabrication technology, are robust to small variations in the fabrication process, and offer the flexibility to be reconfigured in software to give a wide variety of architectures. They also support network training algorithms and therefore speed up this computationally intensive process.

By contrast, analogue systems suffer from low precision weights, and are sensitive to process variations. Also, they do not at present support learning, which must be done separately in software on a conventional computer. They do, however, offer a very high density of processing elements. The Intel ETANN (Electrically Trainable Analogue Neural Network) chip, currently the only analogue neural network chip available commercially, contains over 10,000 weights, giving an effective processing capability of 4GFlops (4×10^9 floating point operations per second) per chip, which is comparable with a large supercomputer. The chips can easily be cascaded to build larger networks with correspondingly higher equivalent processing capacity.

Research is also underway into wafer-scale integration of neural networks, and also into optical and optoelectronic implementations. These latter two make use of modulated laser beams to perform the basic vector-matrix operation, with sigmoidal non-linearities implemented either in non-linear optics or in conventional electronics. Holographic systems are often used to implement the weights.

VIII. EXAMPLE APPLICATIONS

In this section we shall review a number of applications of neural networks in the area of scientific instrumentation. This is in no way intended to be a comprehensive survey of applications, which would be well beyond the scope of this review, but rather a selection of examples to illustrate ideas developed in earlier sections. Information on where to find other applications can be found in the review of neural network literature in the Appendix.

A. Interpolation

One of the simplest ways to use a neural network is as a form of multivariate non-linear regression to find a smooth interpolating function from a set of data points. As an example we consider the problem of predicting a quantity known as the (dimensionless) energy confinement time $\hat{\tau}_E$ of a tokamak plasma from a knowledge of four dimensionless variables constructed from experimentally measured quantities. The dimensionless variables are denoted q (safety factor), ν_* (ratio of effective electron collisionality to trapped electron bounce frequency), β_p (poloidal beta), and $\hat{\rho}_e$ (normalized electron Larmor radius). The precise definitions of these quantities is not important here; more detailed information on this application can be found in Ref. 72. The goal is to predict $\hat{\tau}_E$ from knowledge of these dimensionless variables, and so we are seeking a functional relationship of the form

$$\hat{\tau}_E=F(q,\nu_*,\beta_p,\hat{\rho}_e). \tag{51}$$

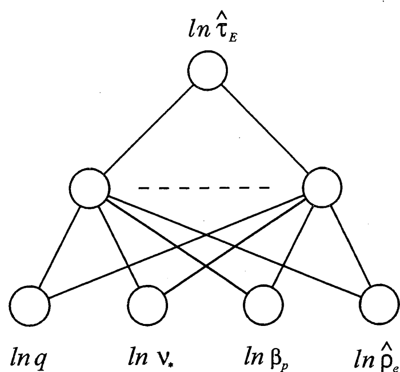

FIG. 24. Network structure used for predicting the normalized energy confinement time $\hat{\tau}_E$ of a tokamak plasma in terms of a set of dimensionless experimentally measured quantities q, ν_*, β_p, and $\hat{\rho}_e$. Note how the basic input and output quantities have been processed by taking logarithms in order to compress their dynamic range. The bias units have been omitted for clarity. (From Ref. 72.)

In principle, this function could be predicted from plasma physics considerations, but in practice the physical processes are much too complex, and so empirical methods are used.

The conventional approach to this problem is to make the arbitrary assumption that the function $F(\ )$ in Eq. (51) takes the form of a product of powers of the independent variables, so that

$$\hat{\tau}_E = e^C q^{\alpha_1} \nu_*^{\alpha_2} \beta_p^{\alpha_3} \hat{\rho}_e^{\alpha_4}, \tag{52}$$

where the parameters C and $\alpha_1 \cdots \alpha_4$ are to be determined empirically from an experimental database. By taking logarithms of Eq. (53) we obtain an expression which is linear in the unknown parameters

$$\ln \hat{\tau}_E = C + \alpha_1 \ln q + \alpha_2 \ln \nu_* + \alpha_3 \ln \beta_p + \alpha_4 \ln \hat{\rho}_e \tag{53}$$

and so the parameters can be determined from a data set of values of $(q, \nu_*, \beta_p, \hat{\rho}_e)$, together with the corresponding values of $\hat{\tau}_E$, by the usual techniques of linear regression (involving the minimization of a sum-of-squares error function).

The limitation of the conventional approach is that it makes the arbitrary assumption of a power law expression (52). This was chosen purely for computational simplicity (because the logarithmic expression is linear in the parameters) and has no physical justification (with one exception to be discussed shortly). We can overcome this limitation by using a neural network to model the function $F(\ )$ in Eq. (51). The network structure is shown in Fig. 24. Note that logarithms are used both as a form of preprocessing of the input variables and also to preprocess the target data (which is taken to be $\ln \hat{\tau}_E$). This is done to compress the dynamic range of the variables and thereby ensure that the relative accuracy is maintained even when some of the quantities have small values. It also has the effect that, if the network mapping is linear, the standard linear regression expression is recovered. Thus the neural network explicitly contains the linear regression approach as a special case.

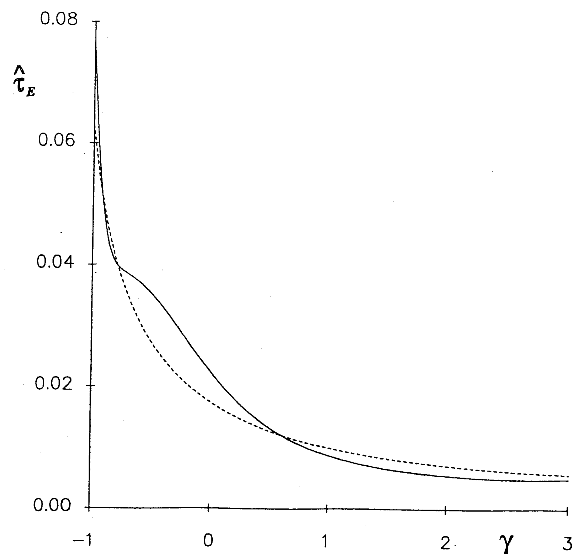

FIG. 25. The solid curve shows the behavior of the energy confinement time $\hat{\tau}_E$ versus the input variables for the energy confinement time problem corresponding to the network shown in Fig. 24. Since there are four input variables, the horizontal axis has been taken along the direction of the first principal component of the test data set, and the parameter γ measures distance along this direction. The dashed curve shows the corresponding results obtained using the linear regression. Note that the linear regression function necessarily produces a power law behavior, while the neural network function is able to represent a more general class of functions and hence can capture more of the structure in the data. (From Ref. 72.)

Since a neural network can potentially contain many more parameters than the five which are found in the linear regression formula (53), it is likely that the network can achieve better fits to the data, even if such an improvement is not statistically significant. This is analogous to the overfitting problem discussed in Sec. V. Such difficulties are avoided by optimizing the network structure using cross-validation, and by comparing the final network with linear regression using a separate test data set. The training data set consisted of 574 data points, with a further 573 in the test set, and the networks were trained using 500 complete epochs of the limited memory BFGS algorithm (discussed in Sec. III).

A reduction in rms error of about 25% is found with the neural network approach, as compared with linear regression. The resulting behavior for the function $F(\ )$ obtained from the neural network is compared with the corresponding result from the linear regression approach in Fig. 25.

This application also provides an illustration of how prior knowledge can be built into a neural network structure. Various theories of energy confinement in tokamaks predict that the dependence of the confinement time on the quantity $\hat{\rho}_e$ should in fact exhibit a power law behavior. This fact can be built into the network structure, while leaving the dependence on the remaining quantities arbitrary. Thus we seek a representation of the form

$$\hat{\tau}_E = (\hat{\rho}_e)^{\alpha} G(q, \nu_*, \beta_p). \tag{54}$$

If the data are again processed using logarithms, then the functional form in Eq. (54) can be represented by the network structure shown in Fig. 26. Since there is a direct con-

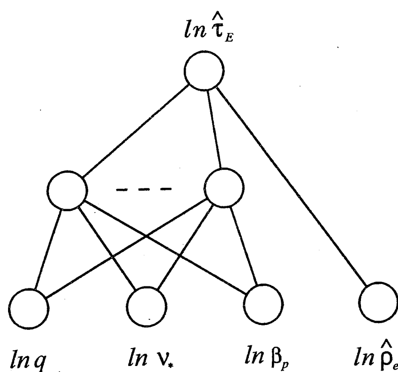

FIG. 26. A modified version of the network shown in Fig. 24 in which $\hat{\tau}_E$ is constrained to be a power law function of the input variable $\hat{\rho}_e$. This is achieved by permitting only a single connection from the $\hat{\rho}_e$ input to the output (and using a linear output unit) to ensure that $\ln\hat{\tau}_E$ is a linear function of $\ln\hat{\rho}_e$. This provides an example of how prior knowledge can be built into the structure of a network. (From Ref. 72.)

nection from the $\ln\hat{\rho}_e$ input to the output, and since the output unit has a linear activation function, this achieves the required effect. The single weight from the $\ln\hat{\rho}_e$ input to the output represents the parameter α in Eq. (54).

Another important feature of neural networks when used to perform non-linear interpolation is their ability to learn how to combine data from several sensors to produce meaningful outputs without the need to develop a detailed physical model to describe the required data transformations. This is called *sensor fusion* or *data fusion*, and plays a role in many neural network applications. An example of the fusion of magnetic field data with line-of-sight optical data to generate spatial profiles of electron density in a plasma can be found in Ref. 73.

B. Classification

We next discuss an application of neural networks involving classification. The problem concerns the monitoring of oil flow along pipelines which carry a mixture of oil, water, and gas, and the aim is to provide a non-invasive technique for measurement of oil flow rates, a problem of considerable importance to the oil industry. The approach described in Ref. 67 is based on the technique of *dual-energy gamma densitometry*. This involves measurement of the attenuation of a collimated beam of mono-energetic gammas passing through the pipe as indicated in Fig. 27. For a gamma beam passing through a single homogeneous substance the fraction of the beam intensity I attenuated per unit length is constant, and so the intensity would decay exponentially with distance according to

$$I=I_0e^{-\mu\rho x}, \tag{55}$$

where I_0 is the beam intensity in the absence of matter, x is the path length within the material, ρ is the mass density of the material, and μ is the mass absorption coefficient for the particular material at a particular gamma energy. For a gamma beam passing through a combination of oil, water, and gas, the intensity of the beam decays like

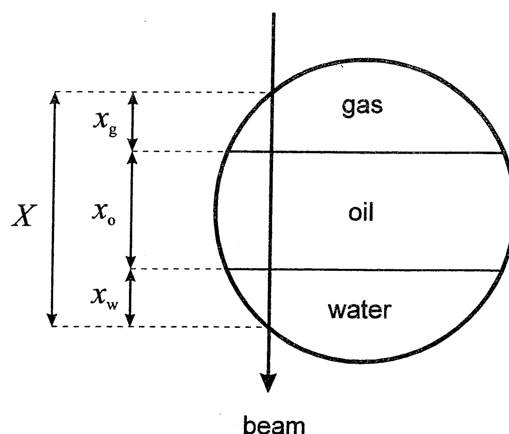

FIG. 27. Schematic cross-section of an oil pipeline containing a mixture of oil, water, and gas in a stratified configuration. Also shown is the path of a gamma beam whose attenuation provides information on the quantities of the three substances present in the pipe (gamma densitometry). The quantities x_o, x_w, and x_g represent the path lengths in the oil, water, and gas phases respectively. If attentuation measurements are made at two different wavelengths, and use is made of the constraint $x_o+x_w+x_g=X$ (where X is the total path length within the pipe) then the values of the three path lengths can be determined. By making such measurements along several chords through the pipe, sufficient information can be obtained to allow the volume fractions of the phases to be determined (with the aid of a neural network mapping) even though the 3 phases may exhibit a variety of different geometrical configurations. (From Ref. 67.)

$$I=I_0e^{-\mu_o\rho_o x_o}e^{-\mu_w\rho_w x_w}e^{-\mu_g\rho_g x_g}, \tag{56}$$

where x_o, x_w, and x_g represent the path lengths through each of the three phases, as indicated in Fig. 27. The measurement from a single beam line does not provide sufficient information to determine all 3 path lengths, and so a second gamma beam of a different energy is passed along the same path as the first beam. Since the absorption coefficients are different at the two energies, the measured attenuation of the beam provides a second independent piece of information. Finally, the three path lengths are constrained to add up to the total path length through the pipe

$$x_o+x_w+x_g=X \tag{57}$$

as shown in Fig. 27. We therefore have enough information to extract the individual path lengths, which can be expressed analytically in terms of the measured attenuations.[67]

Measurements from a single dual-energy beamline are insufficient to determine the volume fractions of the three phases within the pipe, since the phases can occur in a variety of geometrical configurations, as illustrated in Fig. 28. This shows 4 model configurations of 3-phase flows used to generate synthetic data for the neural network study. However, by making measurements along several beamlines, information on the configuration of the phases can be obtained. The system considered in Ref. 67 consists of 3 vertical and 3 horizontal beams arranged asymmetrically.

Multi-phase flows are notoriously difficult to model numerically and so it is not possible to use a first-principles approach to the interpretation of the data from the densitometer. It is, however, possible to collect large amounts of train-

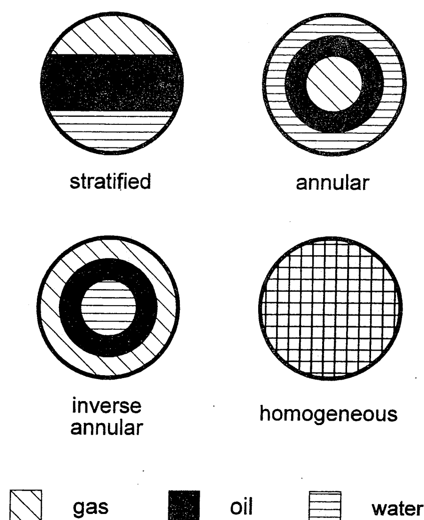

FIG. 28. Four model configurations of 3-phase flow, used to generate training and test data for a neural network which is trained to predict the fractional volumes of oil, water, and gas in the pipe. Inputs to the network are taken from 6 dual-energy gamma densitometers of the kind illustrated in Fig. 27. (From Ref. 67.)

ing data by attaching the densitometer system to a standard multi-phase test rig. This problem is therefore well suited to analysis by neural network techniques.

From the system of 6 dual-energy densitometers we can extract the corresponding 6 values of path length in oil and 6 values of path length in water [the remaining 6 path lengths in gas represent redundant information by virtue of Eq. (57)]. This gives 12 measurements from which we can attempt to determine the geometrical phase configuration. Synthetic data were generated from the 4 model phase configurations shown in Fig. 28. To generate each data point in the training set, one of the configurations was selected at random with equal probability. Then the fractions of oil, water, and gas for this configuration were selected randomly with uniform probability distribution, subject to the constraint that they must add to unity. The 12 independent path lengths are then calculated geometrically, and these form the inputs to a neural network. The dominant source of noise in this application arises from photon statistics, and these are included in the data using the correct Poisson distribution.

In order to predict the phase configuration, the network is given 4 outputs, one for each of the configurations shown in Fig. 28, and a 1-of-N coding is used as described in Sec. V. Networks were trained using a data set of 1,000 examples, and then tested using a further 1,000 independent examples. The network structure consisted of a multilayer perceptron with a single hidden layer of logistic sigmoidal units and an output layer also of logistic sigmoidal units. In order to compare the network against a more conventional approach, the same data were used to train a single-layer network having sigmoidal output units, which corresponds to a form of linear discriminant function.[1,53] The number of hidden units in the network was selected by training several networks and selecting the one with the best performance on the test set, as described in Sec. V, which gave a network having 5 hidden units. Results from the classification problem are summarized in Table I.

More detail on the performance of the network in determining the phase configuration can be obtained from "confusion matrices" which show, for each actual configuration, how the examples were distributed according to the predicted configurations. For perfect classification all entries would be zero except on the leading diagonal. Here the configurations have been ordered as (homogeneous, stratified, annular, inverse annular). The confusion matrices, for both training and test sets, generated by the trained network having 5 hidden units are shown below.

$$\underset{\text{Training}}{\overset{\text{Predicted}}{\text{Actual}\begin{pmatrix} 259 & 0 & 0 & 0 \\ 0 & 239 & 0 & 0 \\ 0 & 0 & 242 & 0 \\ 0 & 0 & 0 & 255 \end{pmatrix}}} \qquad \underset{\text{Test}}{\overset{\text{Predicted}}{\text{Actual}\begin{pmatrix} 255 & 0 & 0 & 0 \\ 1 & 247 & 0 & 1 \\ 2 & 0 & 241 & 0 \\ 3 & 0 & 0 & 250 \end{pmatrix}}}.$$

In this particular application, the real interest is in being able to determine the volume fractions of the three phases, and in particular of the oil. Once the phase configuration is known, these volume fractions can be calculated geometrically from the path length information. However, a more direct approach is to train a network to map the path length information from the densitometers directly onto the volume fractions. This leads to a network with 12 inputs, and 2 outputs corresponding to the volume fractions of oil and water (the volume fraction of gas being redundant information). This is the application which generated the plot of training and test errors versus the number of hidden units shown in Fig. 22.

C. Inverse problems

A large proportion of the data processing tasks encountered in instrumentation applications can be classified as *inverse* problems. The meaning of this term is best illustrated

TABLE I. Results for neural network prediction of phase configurations. Values of zero indicate errors of less that 1.0×10^{-2}.

N_{hidden}	E^{rms} (train) $\times10^2$	E^{rms} (test) $\times10^2$	% correct-train	% correct-test
1	26.9	27.0	98.4	98.2
2	3.86	9.51	99.7	98.1
3	3.16	5.47	99.6	99.2
4	0.0	7.98	100	99.0
5	0.0	6.16	100	99.3
6	0.0	6.33	100	99.2
1-layer	13.3	14.8	98.9	98.6

with a particular example. Consider the general tomography problem illustrated in Fig. 29. The goal is to determine the local spatial distribution of a quantity $Q(\mathbf{r})$ from a number of line-integral measurements

$$\lambda_i=\int_{\Gamma_i} Q(\mathbf{r})\ d\mathbf{r} \tag{58}$$

made along various lines of sight Γ_i through a given region of space. Here, the quantity $Q(\mathbf{r})$ might for instance be the soft x-ray emissivity from a tokamak plasma. Other examples include x-ray absorbtion tomography in medical applications, and ultrasonic tomography for non-destructive testing.

The tomography problem is characterized by the existence of a well-defined *forward* problem in which we suppose that the spatial distribution $Q(\mathbf{r})$ is known and we wish to predict the values of the line integrals λ_i. This problem is well defined and has a unique solution obtained simply from

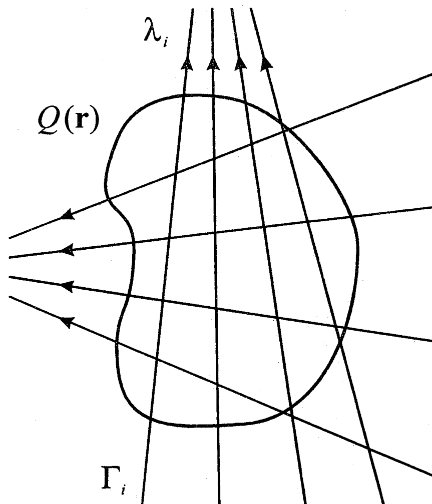

FIG. 29. The tomography problem involves determining the local spatial distribution of a quantity $Q(\mathbf{r})$ from line integral measurements $\lambda_i=\int_{\Gamma_i}Q(\mathbf{r})\ d\mathbf{r}$ made along a number of lines of sight Γ_i. If the function $Q(\mathbf{r})$ were known, then the evaluation of the various line integrals (this is called the *forward* problem) would be straightforward and would give unique results. In practice, we must solve the *inverse* problem of finding $Q(\mathbf{r})$ from the finite set of measured values given by the λ_i, which is ill-posed since there are infinitely many solutions. Neural networks are well-suited to the solution of many inverse problems.

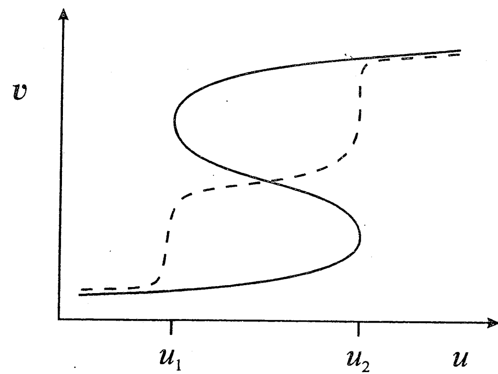

FIG. 30. Schematic example of an inverse problem for which a direct application of neural networks would give incorrect results. The mapping from u to v, shown by the solid curve, is multivalued for values of u in the range u_1 to u_2. A network trained by minimizing a sum-of-squares error based on training data generated from the solid curve would give a result of the form indicated by the dashed curve (this result follows from the fact that the function represented by the trained network is given by the conditional average of the target data, as illustrated in Fig. 18). In the region where $v(u)$ is multi-valued, the network outputs can be substantially different from the desired values.

the evaluation of the integrals in Eq. (58). However, in practice we are required to solve the inverse problem of determining the function $Q(\mathbf{r})$ from a *finite* number of measurements λ_i. This inverse problem is ill-posed[60,74] because there are infinitely many functions $Q(\mathbf{r})$ which give rise exactly to the same given set of line integral measurements. In addition, the measurements may be corrupted by noise, and so we do not necessarily seek a solution which fits the data exactly.

Many of the problems which arise in data analysis to which neural networks may be applied are inverse problems. (Note that curve fitting, and network training, are themselves inverse problems). Examples include the reconstruction of electron density profiles in tokamaks from line integral measurements,[73] and the simultaneous fitting of several overlapping Gaussians to complex spectra.[23] There is generally a well-defined forward problem which may have a fast solution, but the inverse problem is often ill-posed, and with conventional approaches may require computationally intensive iterative techniques to find a solution. The neural network approach offers the advantages of very high speed and can avoid the need for a good initial guess which is often a source of difficulty with conventional iterative methods. In some instances the forward problem can be used to generate synthetic data which can be used to train the network.

There is, however, a potential difficulty in applying neural networks to inverse problems which can lead to very poor results unless appropriate care is taken. Consider the simple example of training a network on the inverse problem shown in Fig. 30. The mapping from the variable v to the variable u, shown by the solid curve, is single-valued (in other words it is a *function*). If, however, we consider the inverse mapping from u to v, then we see that this is not single-valued for a range of u values between u_1 and u_2. As discussed in Sec. V the output of a trained network approximates the conditional average of the target data given by Eq. (44). If data

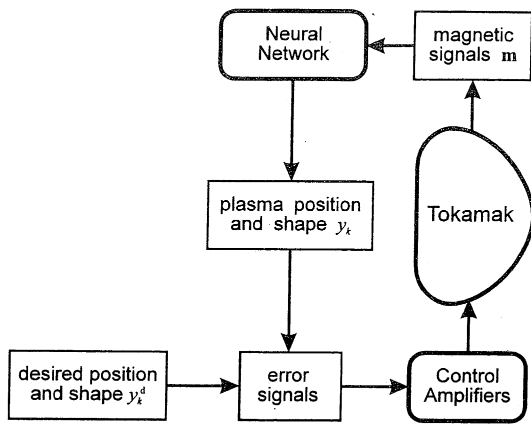

FIG. 31. Neural networks have recently been used for real-time feedback control of the position and shape of the plasma in the COMPASS tokamak experiment, using the control system shown here. Inputs to the network consist of a set of signals **m** from magnetic pick-up coils which surround the tokamak vacuum vessel. These are mapped by the network onto the values of a set of variables y_k which describe the position and shape of the plasma boundary. Comparison of these variables with their desired values y_k^d (which are preprogrammed to have specific time variations) gives error signals which are sent, via control amplifiers, to sets of feedback control coils which can modify the position and shape of the plasma boundary. Due to the very high speed (~10 μs) at which the feedback loop must operate, a fully parallel hardware implementation of the neural network was used. (From Ref. 71.)

from the solid curve in Fig. 30 are used to train a network the resulting network mapping will have the form shown by the dashed curve. In the range where the data is multivalued the output of the network can be completely spurious, since the average of several values of v may itself not be a valid value for that variable for the given value of u. This problem is not resolved by increasing the quantity of data or by improvements in the training procedure.

When applying neural networks to inverse problems it is therefore essential to anticipate the possibility that the data may be multivalued. One approach to resolving this problem is to exclude all but one of the branches of the inverse mapping (or by training separate networks for each of the branches if all possible solutions are needed). For a detailed example of how this technique is applied in practice, in this case to the determination of the coefficients in a Gaussian function fitted to a spectral line, see Ref. 22.

D. Control applications

In this review we have concentrated almost entirely on neural networks for data analysis, and indeed this represents the area where these techniques are currently having the greatest practical impact. However, neural networks also offer considerable promise for the solution of many complex problems in non-linear control.

Feedforward networks, of the kind considered in this review, can be used to perform a non-linear mapping within the context of a conventional linear feedback control loop. This technique has been exploited successfully for the feedback control of tokamak plasmas[71,75] as illustrated in Fig. 31. Here the inputs to the network consist of a number of magnetic signals (typically between 10 and 100) obtained from pick-up coils located around the tokamak vacuum vessel. These are mapped by the neural network onto a set of geometrical parameters which describe the position and shape of the boundary of the plasma. The values for these parameters as predicted by the neural network are compared with desired values which have been preprogrammed as functions of time prior to the plasma pulse. The resulting error signals are then sent to standard PID (proportional-integral-differential) linear control amplifiers which adjust the position and shape of the plasma by changing the currents in a number of control coils.

The network is trained off-line in software from a large data set of example plasma configurations obtained by numerical solution of the plasma equilibrium equations. In order to achieve real-time operation, the network was implemented in special purpose hybrid digital-analogue hardware[70] described in Sec. VII. Values for the network weights, obtained from the software simulation, are loaded into the network prior to the plasma pulse. This system recently achieved the first real-time control of a tokamak plasma by a neural network.[71]

This application provides another example of the use of prior knowledge in neural networks. It is a consequence of the linearity of Maxwell's equations that, if all of the currents in the tokamak system are scaled by a constant factor, the magnetic field values will be scaled by the same constant factor and the plasma position and shape will be unchanged. This implies that the mapping from measured signals to the position and shape parameters, represented by the network, should have the property that, if all the inputs are scaled by the same factor, the outputs should remain unchanged. Since the order of magnitude of the inputs can vary by a factor of up to 100 during a plasma pulse, there is considerable benefit in building in this prior knowledge explicitly. This is achieved by dividing all inputs by the value of the total plasma current. A hardware implementation of this normalization process was developed for real-time operation. If this prior knowledge were not included in the network structure, the network would have to learn the invariance property purely from the examples in the data set.

Another recent real-time application for neural networks was for the control of 6 mirror segments in an astronomical optical telescope in order to perform real-time cancellation of distortions due to atmospheric turbulence.[76] This technique, called adaptive optics, involves changing the effective mirror shape every 10 ms. Conventional approaches involve iterative algorithms to calculate the required deformations of the mirror, and are computationally prohibitive. The neural network provides a fast alternative, which achieves high accuracy. When the control loop is closed the image quality shows a strong improvement, with a resolution close to that of the Hubble space telescope. In this case the network was implemented on an array of transputers.

Neural networks can also be used as non-linear *adaptive* components within a control loop. In this case the network continues to learn while acting as a controller, and in principle can learn to control complex non-linear systems by trial and error. This raises a number of interesting issues connected with the fact that the training data which the network sees is itself dependent on the control actions of the network.

Such issues take us well beyond the scope of this review, however, and so we must refer the interested reader to the literature for further details.[77–79]

IX. DISCUSSION

In this review we have focused our attention on feedforward neural networks viewed as general parameterized nonlinear mappings between multi-dimensional spaces. Such networks provide a powerful set of new data analysis and data processing tools with numerous instrumentation applications. While feedforward networks currently account for the majority of applications there are many other network models, performing a variety of different functions, which we do not have space to discuss in detail here. Instead we give a brief overview of some of the topics which have been omitted, along with pointers to the literature. We then conclude with a few remarks on the future of neural computing.

A. Other network models

Most of the network models described so far are trained by a *supervised* learning process in which the network is supplied with input vectors together with the corresponding target vectors. There are other network models which are trained by *unsupervised* learning in which only the input vectors $\mathbf{x}^q$ are supplied to the network. The goal in this case is to model structure within the data rather than learn a functional mapping.

One example of unsupervised training is called *density estimation* in which the network forms a model of the probability distribution $p(\mathbf{x})$ of the data as a function of $\mathbf{x}$.[53,1] We have already encountered one example of this in Sec. IV, using the Gaussian mixture model in Eq. (35). Another example is *clustering* in which the goal is to discover any clumping of the data which may indicate structure having some particular significance.[53,80] Yet another application of unsupervised methods is data visualization in which the data is projected onto a 2-dimensional surface embedded in the original d-dimensional space, allowing the data to be visualized on a computer screen.[80] In this case the training process corresponds to an iterative optimization of the location of the surface in order to capture as much of the structure in the data as possible. Unsupervised networks are also used for dimensionality reduction of the data prior to treatment with supervised learning techniques in order to mitigate the effects of the curse of dimensionality.

One of the restrictions placed on the networks discussed in this review is that they should have a feedforward structure so that the output values become explicit functions of the inputs. If we consider network diagrams with connections which form loops then the network acquires a dynamical behavior in which the activations of the units must be calculated by evolving differential equations through time. A class of such networks having some historical significance is that developed by Hopfield[15,16] who showed that, if the connection from unit a to unit b has the same strength as the connection from unit b back to unit a, then the evolution of the network corresponds to a relaxation described by an energy function, thereby ensuring that the network evolves to a stationary state. Such networks can act as associative memories which reconstruct a complete pattern from a partial cue, or from a corrupted version of that pattern. They have also been used to solve combinatorial optimization problems, such as placing of components in an integrated circuit or the scheduling of steps in a manufacturing process.

Another aspect of the techniques considered in this review is that all of the input data have been treated as static vectors. There is also considerable interest in being able to deal effectively with time varying signals. The simplest, and most common approach, is to sample the time series at regular intervals and then treat a succession of observed values as a static vector which can then be used as the input vector of a standard feedforward network. This approach has been used with considerable success both for classification of time series in problems such as speech recognition[81] and for prediction of future values of the time series[82] in applications such as financial forecasting or the prediction of sunspot activity. A more comprehensive approach would, however, make use of dynamical networks of the kind discussed above.

It should be emphasized that most of these neural network techniques have their counterparts in conventional methods. In many cases the neural network provides a nonlinear extension of some well known linear technique. Anyone wishing to make serious use of neural networks is therefore recommended to become familiar with these conventional approaches.[53–55]

Throughout this review we have discussed learning in neural networks in terms of the minimization of an error function. However, learning and generalization in neural networks can also be formulated in terms of a Bayesian inference framework,[1,83–86] and this is currently an active area of research.

B. Future developments

Feedforward neural networks are now becoming well established as methods for data processing and interpretation, and as such will find an ever greater range of practical applications both in scientific instrumentation and many other fields. However, it is clear too that the connectionist paradigm for information processing is a very rich one which, 50 years after the pioneering work of McCulloch and Pitts, we are only just beginning to explore. It is likely to be a very long time before artificial neural networks approach the complexity or performance of their biological counterparts. Nevertheless, the fact that biological systems achieve such impressive feats of information processing using this basic connectionist approach will remain as a constant source of inspiration. While it would be unwise to speculate on future technical developments in this field, there can be little doubt that the future will be an exciting one.

APPENDIX: GUIDE TO THE NEURAL COMPUTING LITERATURE

The last few years have witnessed a dramatic growth of activity in neural computing accompanied by a huge range of books, journals, and conference proceedings. Here we aim to

provide an overview of the principal sources of information on neural networks, although we cannot hope to be exhaustive.

The following journals specialize in neural networks. It should be emphasized, however, that the subject spans many disciplines and that important contributions also appear in a range of journals specializing in other subjects.

Neural Networks is published bimonthly by Pergamon Press and first appeared in 1988. It covers biological, mathematical, and technological aspects of neural networks, and a subscription is included with membership of the International Neural Network Society.

Neural Computation is a high quality multidisciplinary letters journal published quarterly by MIT Press.

Network is another cross-disciplinary journal and is published quarterly by the Institute of Physics in the U.K.

International Journal of Neural Systems is published quarterly by World Scientific also covers a broad range of topics.

IEEE Transactions on Neural Networks is a journal with a strong emphasis on artificial networks and technology and appears bimonthly.

Neural Computing and Applications is a new journal concerned primarily with applications and is published quarterly by Springer-Verlag.

Neurocomputing is published bimonthly by Elsevier.

There are currently well over 100 books available on neural networks and it is impossible to survey them all. Many of the introductory texts give a rather superficial treatment, generally with little insight into the key issues which often make the difference between successful applications and failures. Some of the better books are those given in Refs. 87 and 88. A more comprehensive account of the material covered in this review can be found in Ref. 1.

One of the largest conferences on neural networks is the International Joint Conference on Neural Networks (IJCNN) held in the USA (and also in the Far East) with the proceedings published by IEEE. A scan through the substantial volumes of the proceedings gives a good indication of the tremendous range of applications now being found for neural network techniques. A similar annual conference is the *World Congress on Neural Networks.* A comparable, though somewhat smaller, conference is held each year in Europe as the International Conference on Artificial Neural Networks (ICANN). An excellent meeting is the annual Neural Information Processing Systems conference (NIPS) whose proceedings are published under the title *Advances in Neural Information Processing Systems* by Morgan Kaufman. These proceedings provide a snapshot of the latest research activity across almost all aspects of neural networks, and are highly recommended. Details of future conferences can generally be found in the various neural network journals.

[1] C. M. Bishop, *Neural Networks for Statistical Pattern Recognition* (Oxford University Press, Oxford, 1994).
[2] H. White, Neural Comput. **1**, 425 (1989).
[3] C. M. Bishop, Proc. IEE, Proceedings: Vision, Image and Speech; Special Issue on Neural Networks (1994).
[4] E. R. Kandel and J. H. Schwartz, *Principles of Neuroscience*, 2nd ed. (Elsevier, New York, 1985).
[5] Scientific American, special issue on Mind and Brain, September (1992).
[6] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* (MIT Press, Cambridge, 1986), Vol. 2.
[7] W. S. McCulloch and W. Pitts, Bull. Math. Biophys. **5**, 115 (1943).
[8] D. O. Hebb, *The Organization of Behaviour* (Wiley, New York, 1949).
[9] F. Rosenblatt, Psychol. Rev. **65**, 386 (1958).
[10] F. Rosenblatt, *Principles of Neurodynamics* (Spartan Books, Washington, DC, 1962).
[11] B. Widrow, *Self-Organizing Systems*, edited by G. T. Yovitts (Spartan Books, Washington, DC, 1962).
[12] B. Widrow and M. E. Hoff, *Adaptive Switching Ciruits* (IRE WESCON Convention Record, New York, 1960), p. 96.
[13] B. Widrow and M. Lehr, Proc. IEEE **78**, 1415 (1990).
[14] M. Minsky and S. Papert, *Perceptrons* (MIT Press, Cambridge, 1959), also available in an expanded edition (1990).
[15] J. J. Hopfield, Proc. Natl. Acad. Sci. **79**, 2554 (1982).
[16] J. J. Hopfield, Proc. Natl. Acad. Sci. **81**, 3088 (1984).
[17] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Nature **323**, 533 (1986).
[18] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* (MIT Press, Cambridge, 1986), Vol. 1.
[19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* (MIT Press, Cambridge, 1986), Vol. 3.
[20] *Neurocomputing: Foundations of Research*, edited by J. A. Anderson and E. Rosenfeld (MIT Press, Cambridge, 1988).
[21] *Neurocomputing*, edited by J. A. Anderson and E. Rosenfeld (MIT Press, Cambridge, 1990), Vol. 2.
[22] C. M. Bishop and C. M. Roach, Rev. Sci. Instrum. **63**, 4450 (1992).
[23] C. M. Bishop, C. M. Roach, and M. G. von Hellerman, Plasma Phys. Control. Fusion **35**, 765 (1993).
[24] K. Funahashi, Neural Networks **2**, 183 (1989).
[25] G. Cybenko, Math. Control, Signals Syst. **2**, 304 (1989).
[26] K. Hornick, M. Stinchcombe, and H. White, Neural Networks **2**, 359 (1989).
[27] K. Hornick, Neural Networks **4**, 251 (1991).
[28] V. Y. Kreinovich, Neural Networks **4**, 381 (1991).
[29] A. R. Gallant and H. White, Neural Networks **5**, 129 (1992).
[30] Le Cun Y *et al.*, Neural Computation **1**, 541 (1989).
[31] J. F. Kolen and J. B. Pollack, in *Advances in Neural Information Processing Systems* (Morgan Kaufmann, San Mateo, CA, 1991), Vol. 3, p. 860.
[32] C. M. Bishop, Neural Computation **4**, 494 (1992).
[33] H. Robbins and S. Monro, Annu. Math. Stat. **22**, 400 (1951).
[34] J. Kiefer and J. Wolfowitz, Annu. Math. Stat. **23**, 462 (1952).
[35] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. (Cambridge University Press, Cambridge, 1992).
[36] J. E. Dennis and R. B. Schnabel, *Numerical Methods for Unconstrained Optimisation and Non-linear Equations* (Prentice–Hall, Englewood Cliffs, NJ, 1983).
[37] E. M. Johansson, F. U. Dowla, and D. M. Goodman, Int. J. Neural Syst. **2**, 291 (1992).
[38] D. F. Shanno, Math. Operations Res. **3**, 244 (1978).
[39] R. Battiti, Complex Syst. **3**, 331 (1989).
[40] D. S. Broomhead and D. Lowe, Complex Syst. **2**, 321 (1988).
[41] J. Moody and C. L. Darken, Neural Comput. **1**, 281 (1989).
[42] M. J. D. Powell, in *Algorithms for Approximations*, edited by J. C. Mason and M. G. Cox (Clarendon, Oxford, 1987).
[43] C. A. Micchelli, Constructive Approx. **2**, 11 (1986).
[44] E. Hartman, J. D. Keeler, and J. Kowalski, Neural Comput. **2**, 210 (1990).
[45] J. Park and I. W. Sandberg, Neural Comput. **3**, 246 (1991).
[46] S. Chen, S. F. N. Cowan, and P. M. Grant, IEEE Trans. Neural Networks **2**, 302 (1991).
[47] J. MacQueen, in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability* (University of California Press, Berkeley, CA, 1967), Vol. 1, p. 281.
[48] G. J. McLachlan and K. E. Basford, *Mixture Models: Inference and Applications to Clustering* (Marcel Dekker, New York, 1988).
[49] A. P. Dempster, M. N. Laird, and D. B. Rubin, J. R. Stat. Soc. B **39**, 1 (1977).
[50] G. H. Golub and W. Kahan, SIAM Num. Analysis **2**, 205 (1965).
[51] M. D. Richard and R. P. Lippmann, Neural Comput. **3**, 461 (1991).

[52] D. W. Ruck *et al.*, IEEE Trans. Neural Networks **1**, 296 (1990).
[53] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis* (Wiley, New York, 1973).
[54] K. Fukunaga, *Introduction to Statistical Pattern Recognition*, 2nd ed. (Academic, San Diego, CA, 1990).
[55] P. A. Devijer and J. Kittler, *Pattern Recognition: A Statistical Approach* (Prentice–Hall, Hemel Hempstead, UK, 1982).
[56] D. J. Hand, *Discrimination and Classification* (Wiley, New York, 1981).
[57] D. Lowe and A. R. Webb, Network **1**, 299 (1990).
[58] J. Moody, in *Advances in Neural Information Processing Systems* (Morgan Kaufmann, San Mateo, CA, 1993).
[59] S. Geman, E. Bienenstock, and R. Doursat, Neural Comput. **4**, 1 (1992).
[60] A. N. Tikhonov and V. Y. Arsenin, *Solutions of Ill-posed Problems* (Winston, Washington, DC, 1977).
[61] G. Wahba, Ann. Statist. **13**, 1378 (1985).
[62] C. M. Bishop, Neural Comput. **3**, 579 (1991).
[63] T. Poggio and F. Girosi, Proc. IEEE **78**, 1481 (1990).
[64] T. Poggio and F. Girosi, Science **247**, 978 (1990).
[65] C. M. Bishop, IEEE Trans. Neural Networks **4**, 882 (1993).
[66] M. Stone, Operationforsh. Statis. Ser. Statist. **9**, 127 (1978).
[67] C. M. Bishop and G. D. James, Nucl. Instrum. Methods Phys. Res. A **327**, 580 (1993).
[68] R. E. Bellman, *Adaptive Control Processes* (Princeton University Press, Princeton, NJ, 1961).
[69] I. T. Jollife, *Principal Component Analysis* (Springer, New York, 1986).
[70] C. M. Bishop, P. Cox, P. Haynes, C. M. Roach, M. E. U. Smith, T. N. Todd, and D. L. Trotman, in *Neural Network Applications*, edited by J. G. Taylor (Springer, London, 1992), p. 114.
[71] C. M. Bishop, P. Cox, P. Haynes, C. M. Roach, M. E. U. Smith, T. N. Todd, and D. L. Trotman, Neural Computation (to be published).
[72] L. Allen and C. M. Bishop, Plasma Phys. Control Fusion **34**, 1291 (1992).
[73] C. M. Bishop, I. Strachan, J. O'Rourke, G. Maddison, and P. Thomas, Neural Comput. Appl. **1**, 4 (1993).
[74] V. A. Morozov, *Methods for Solving Ill-posed Problems* (Springer, Berlin, 1984).
[75] J. B. Lister and H. Schnurrenberger, Nucl. Fusion **31**, 1291 (1991).
[76] D. G. Sandler, T. K. Barrett, D. A. Palmer, R. Q. Fugate, and W. J. Wild, Nature **351**, 300 (1991).
[77] K. J. Åström and B. Wittenmark, *Adaptive Control* (Addison–Wesley, Redwood City, CA, 1989).
[78] W. T. Miller, R. S. Sutton, and P. J. Werbos, *Neural Networks for Control* (MIT Press, Cambridge, 1990).
[79] *Handbook of Intelligent Control*, edited by D. A. White and D. A. Sofge (Van Nostrand Reinhold, New York, 1992).
[80] T. Kohonen, *Self-organization and Associative Memory*, 3rd ed. (Springer, London, 1989).
[81] R. P. Lippmann, Neural Comput. **1**, 1 (1989).
[82] A. Lapedes and R. Farber, in *Neural Information Processing Systems*, edited by D. Z. Anderson (American Institute of Physics, New York, 1988), p. 442.
[83] D. J. C. MacKay, Neural Comput. **4**, 415 (1992).
[84] D. J. C. MacKay, Neural Comput. **4**, 448 (1992).
[85] D. J. C. MacKay, Neural Comput. **4**, 720 (1992).
[86] W. L. Buntine and A. S. Weigend, Complex Syst. **5**, 603 (1991).
[87] J. Hertz, A. Krogh, and R. G. Palmer, *Introduction to the Theory of Neural Computation* (Addison–Wesley, Redwood City, CA, 1991).
[88] R. Hecht-Nielsen, *Neurocomputing* (Addison–Wesley, Redwood City, CA, 1990).