

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

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

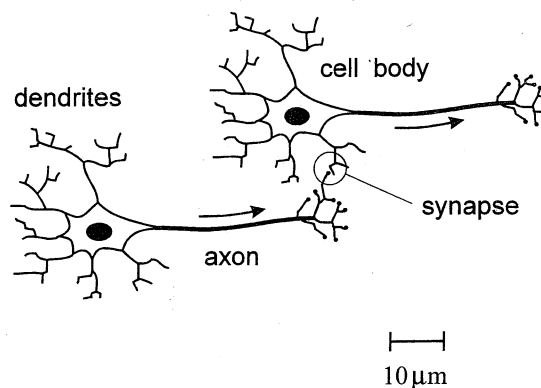


FIG. 1. Schematic illustration of two biological neurons. The dendrites act as inputs, and when a neuron fires an action potential propagates along its axon in the direction shown by the arrow. Interaction between neurons takes place at junctions called synapses.

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.

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

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 Pitts⁷ 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) network¹¹ 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 be-

gan 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, \dots, x_d and maps them onto c dependent variables y_1, \dots, 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,

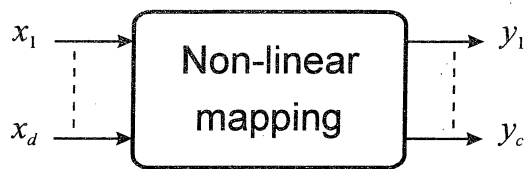


FIG. 4. Schematic illustration of a general non-linear functional mapping from a set of input variables x_1, \dots, x_d to a set of output variables y_1, \dots, y_c . Each of the y_k can be an arbitrary non-linear function of the inputs.

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 $\mathbf{x} \equiv (x_1, \dots, x_d)$ and $\mathbf{y} \equiv (y_1, \dots, y_c)$. 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, \dots, w_l . Again, the weights (and biases) can conveniently be grouped together to form a *weight vector* $\mathbf{w} \equiv (w_1, \dots, w_l)$. 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 m th order polynomial given by

$$y = w_m x^m + \dots + w_1 x + w_0 = \sum_{j=0}^m w_j x^j. \quad (4)$$

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, \dots, 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 $q = 1, \dots, n$. 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

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

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.

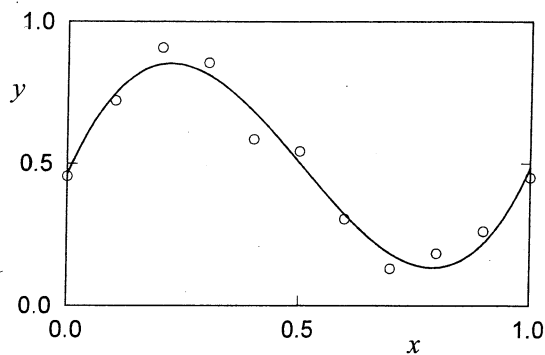


FIG. 5. An example of curve fitting using a polynomial function. Here 11 data points have been generated by sampling the function $\sin(2\pi x)$ at equal intervals of x and then adding zero mean Gaussian noise with variance of 0.05. The solid curve shows a cubic polynomial fitted by minimizing a sum-of-squares 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 $\mathbf{x}^q = (x_1^q, \dots, x_d^q)$ 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

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

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

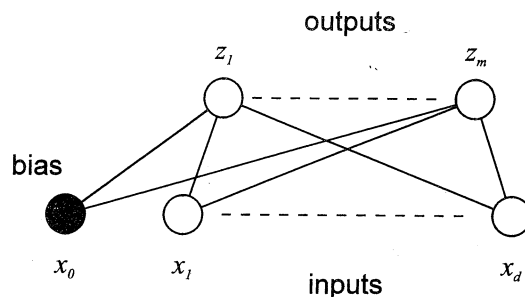


FIG. 6. A single-layer network having d inputs x_1, \dots, x_d and m outputs z_1, \dots, z_m . Each line in the diagram corresponds to one of the weight parameters in the network function. The biases are shown as weights from an extra input unit (denoted by the solid black circle) whose activation is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

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.

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

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

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 $x_0=1$.

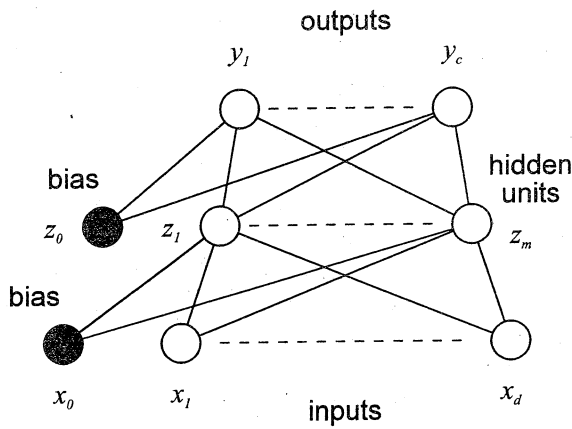


FIG. 7. A multilayer perceptron neural network having two layers of weights. Such networks are capable of approximating any continuous non-linear function to arbitrary accuracy provided the number m of hidden units is sufficiently large. (From Ref. 1.)

Note that Fig. 6 can be regarded as a diagrammatic shorthand 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

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

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

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.

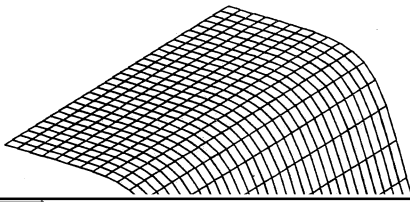
If the activation functions $g(\)$ and $\tilde{g}(\)$ for the network structure shown in Fig. 7 are taken to be linear, the network transformation reduces to the product of two matrices, which is itself just a matrix. However, if the activation function $g(\)$ for the hidden units is taken to be non-linear 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 $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

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

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

$$g'(a) = 1 - g(a)^2. \quad (11)$$

Another common choice of activation function is the logistic



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

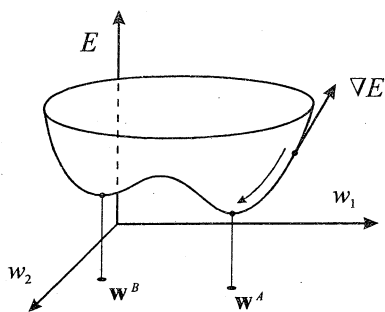


FIG. 10. Schematic illustration of the error function $E(\mathbf{w})$ seen as a surface over weight space (the space spanned by the values of the weight and bias parameters $\mathbf{w} = \{w_1, \dots, w_j\}$). The weight vector \mathbf{w}^A corresponds to the global minimum of the error function, while 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.)

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

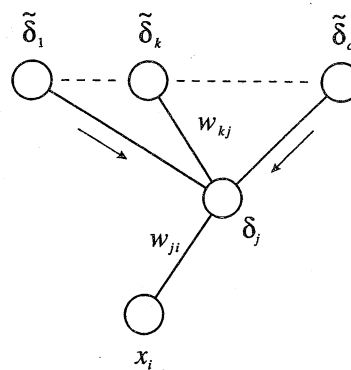


FIG. 11. An illustration of how backpropagation of error signals is used to evaluate derivatives of the error function with respect to the weight (and bias) parameters in the first layer of a 2-layer network. The error signal δ_j at hidden unit j is obtained by summing error signals $\tilde{\delta}_k$ from the output units $k=1, \dots, c$ 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.)

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

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

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

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

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

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

We now introduce the definition

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

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

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

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

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

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

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

We can then write the required derivative as

$$\frac{\partial E^q}{\partial w_{ji}} = \frac{\partial E^q}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}. \quad (21)$$

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

$$\delta_j \equiv \frac{\partial E^q}{\partial a_j}, \quad (22)$$

we can then write the derivative in the form

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

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

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

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

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

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.

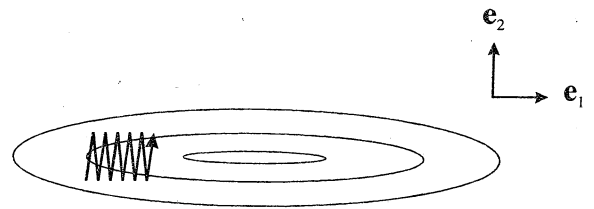


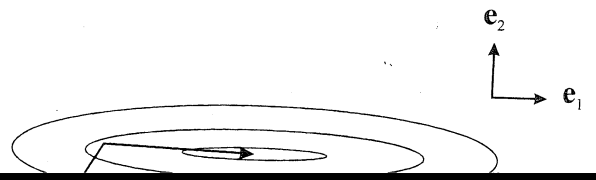
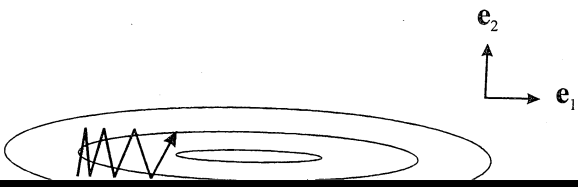
FIG. 12. Schematic illustration of the contours of a quadratic error surface in a 2-dimensional weight space in the neighborhood of a minimum, for which the curvature along the e_1 direction is much less than the curvature along the e_2 direction. Simple gradient descent, which takes successive steps 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.

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 $E = \sum_q E^q$ 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

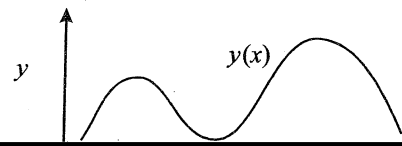
where τ denotes the step number in the iteration, and the parameter η is called the *learning rate* and in the simplest scheme is set to a fixed value chosen by guesswork.

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 e_1 in weight space which is substantially smaller than the curvature along a second direction 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 e_2 direction, and this leads to very slow progress along the 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* term¹⁸ into the learning equations. The weight update formula is modified to give



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, and this is a good



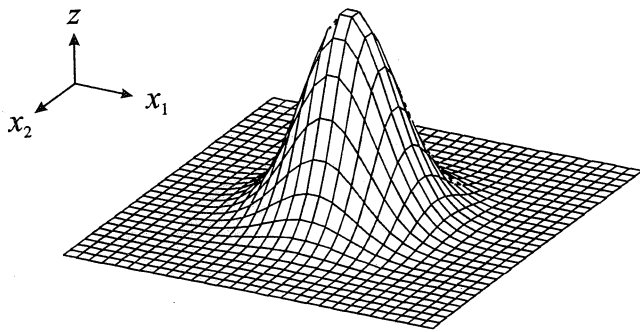


FIG. 16. Plot of the activation $z(x_1, x_2)$ of a Gaussian hidden unit as used in a radial basis function network, as a function of two input variables x_1 and x_2 . This plot should be compared with the sigmoid shown in Fig. 8.

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

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

where $\boldsymbol{\mu}_j$ is a vector representing the center of the j th 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 ba-

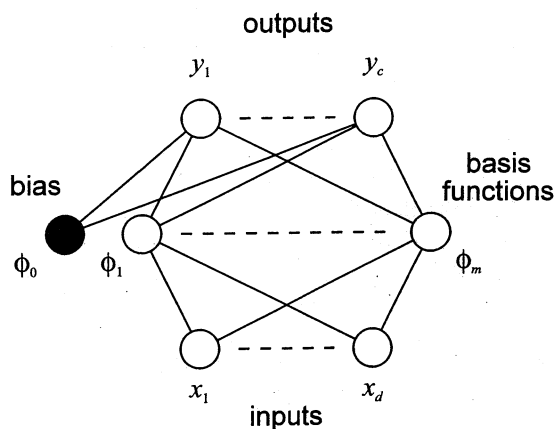


FIG. 17. Architecture of a radial basis function neural network having d inputs x_1, \dots, x_d and c outputs y_1, \dots, y_c . Each of the m basis functions ϕ_j computes a localized (often Gaussian) function of the input vector. The lines connecting the inputs to the basis function ϕ_j represent the 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.)

sis function. The second layer of the network is identical to that of a multilayer perceptron in which the output units have linear activation functions.

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 μ_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

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

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

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

The basis function parameters can then be set by maximizing this likelihood. Since the likelihood is an analytic non-linear function of the parameters $\{\mu_j, \sigma_j\}$, 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 $\{\mu_j, \sigma_j\}$ 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

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

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

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

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

$$0 = \Phi^T \{ \Phi \mathbf{W}^T - \mathbf{T} \}, \quad (39)$$

where the matrices have the following elements: $\Phi = (\phi_j^q)$, $\mathbf{W} = (\bar{w}_{kj})$, 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

$$\mathbf{W}^T = \Phi^\dagger \mathbf{T}, \quad (40)$$

where Φ^\dagger is the *pseudo-inverse*⁵⁰ of the matrix Φ and is given by

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

(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 decomposition³⁵ 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 $\{\mu_j, \sigma_j\}$, via the quantities ϕ_j^q . Once these parameters have been determined, the second-layer weights can then be set to their optimum values.

Note that the matrix Φ 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 Φ 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

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

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

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

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

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

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

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

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

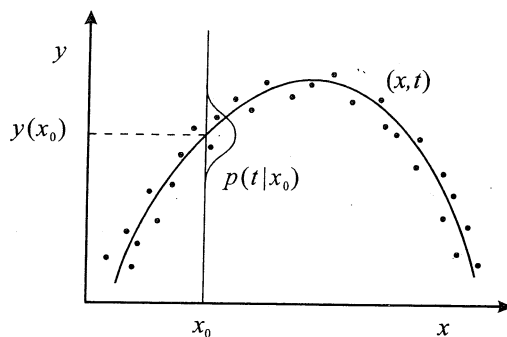


FIG. 18. Schematic illustration of some noisy data points (the black dots) each of which consists of a value of the input variable x together with a corresponding target value t . The curve shows the optimal network function $y(x)$ obtained by minimizing a sum-of-squares error function. For any given 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.)

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

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 \mathcal{E}_k ($k=1, \dots, c$) 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 $\mathcal{E}_1 \equiv$ "normal," and $\mathcal{E}_2 \equiv$ "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 $P(\mathcal{E}_k | \mathbf{x})$ of belonging to class \mathcal{E}_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

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

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

$$y_k(\mathbf{x}) = P(\mathcal{E}_k | \mathbf{x}). \quad (48)$$

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,

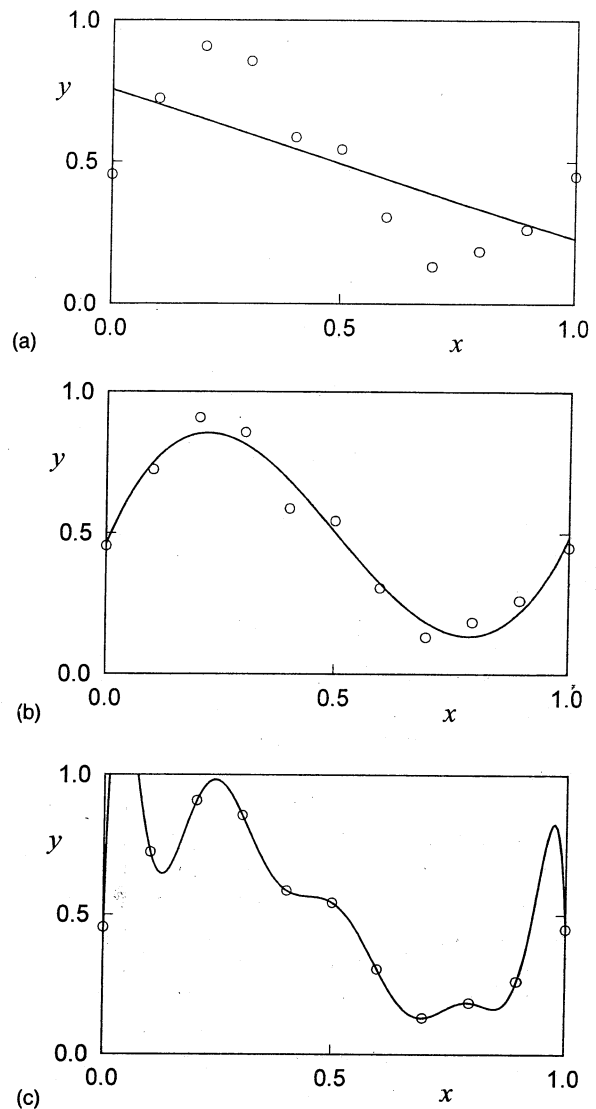
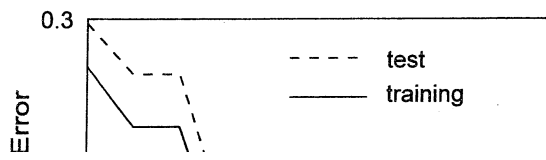


FIG. 19. Examples of curve fitting using polynomials of successively higher order, using the same data as was used to plot Fig. 5. (a) was obtained using a first order (linear) polynomial, and is seen to give a rather poor representation of the data. By using a cubic polynomial, as shown in (b), a much 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



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

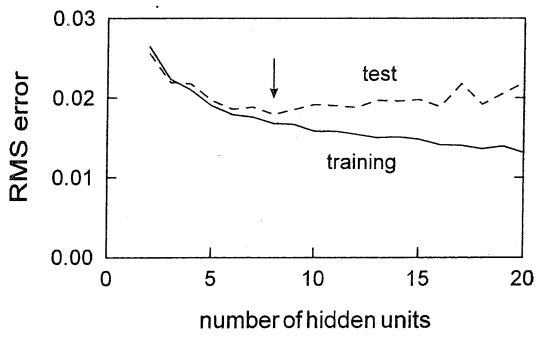


FIG. 22. An example of the optimization of the topology of a network taken from a real application (the determination of oil fraction in multi-phase flows using gamma densitometry, discussed in Sec. VIII). Here the root-mean-square error is plotted as a function of the number of hidden units for a training set consisting of 1000 examples and a test set also of 1000 ex-

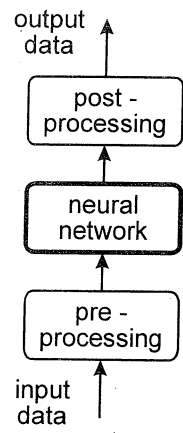


FIG. 23. Raw data are generally preprocessed before being presented to a

lem with the dimensionality of the space of input variables. variables. In a typical application the different input vari-

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 sub-optimal 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 network architecture. If the desired structure from the

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 with sophisticated user interfaces.

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 im-

tal 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 low-cost digital processor.

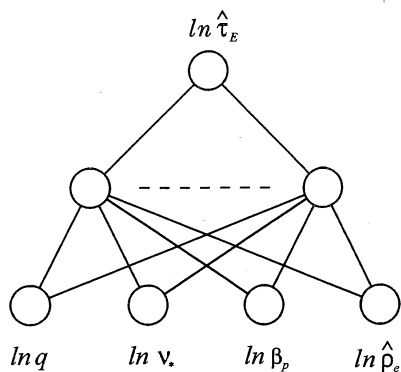


FIG. 24. Network structure used for predicting the normalized energy confinement time $\hat{\tau}_E$ of a tokamak plasma in terms of a set of dimensionless experimentally measured quantities q , ν_* , β_p , and $\hat{\rho}_e$. Note how the basic input and output quantities have been processed by taking 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

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

where the parameters C and $\alpha_1 \dots \alpha_4$ 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

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

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.

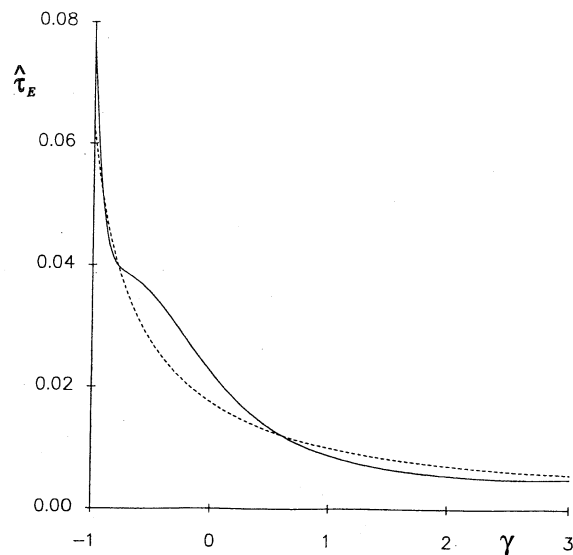


FIG. 25. The solid curve shows the behavior of the energy confinement time $\hat{\tau}_E$ versus the input variables for the energy confinement time problem corresponding to the network shown in Fig. 24. Since there are four input variables, the horizontal axis has been taken along the direction of the 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.)

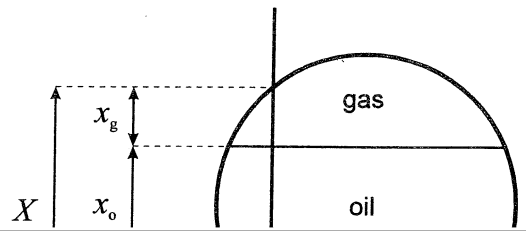
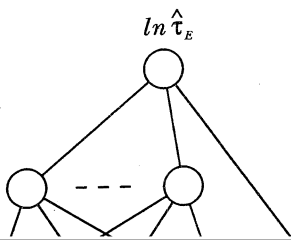
Since a neural network can potentially contain many more parameters than the five which are found in the linear regression formula (53), it is likely that the network can achieve better fits to the data, even if such an improvement is not statistically significant. This is analogous to the overfitting 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

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

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-



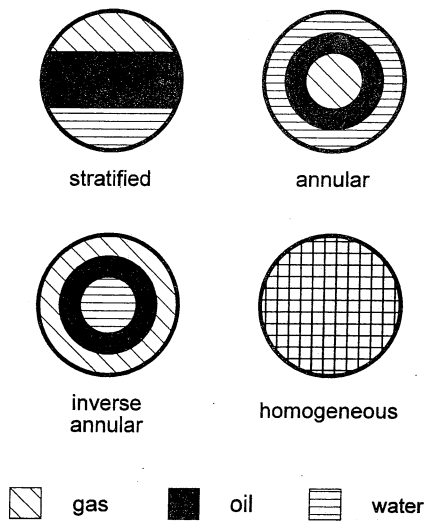


FIG. 28. Four model configurations of 3-phase flow, used to generate training and test data for a neural network which is trained to predict the fractional volumes of oil, water, and gas in the pipe. Inputs to the network are taken from 6 dual-energy gamma densitometers of the kind illustrated in Fig.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

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 pre-

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

N_{hidden}	E^{rms} (train) $\times 10^2$	E^{rms} (test) $\times 10^2$	% 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

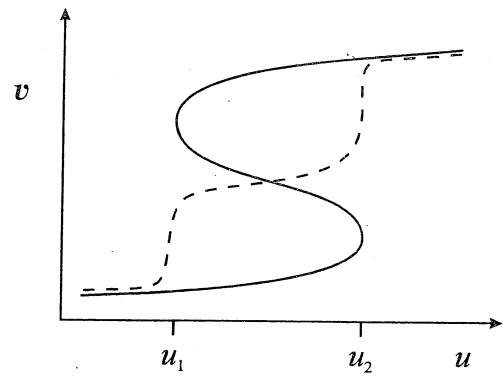


FIG. 30. Schematic example of an inverse problem for which a direct application of neural networks would give incorrect results. The mapping from u to v , shown by the solid curve, is multivalued for values of u in the range u_1 to u_2 . A network trained by minimizing a sum-of-squares error based on 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

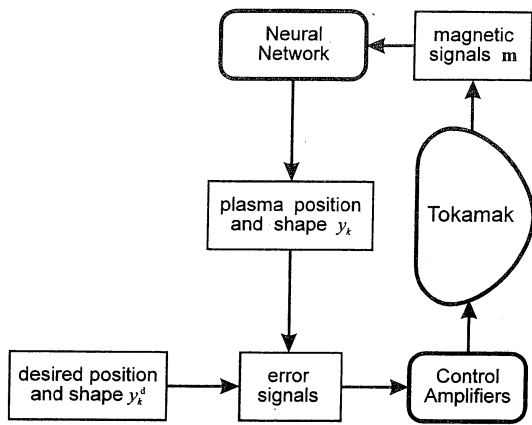


FIG. 31. Neural networks have recently been used for real-time feedback control of the position and shape of the plasma in the COMPASS tokamak experiment, using the control system shown here. Inputs to the network consist of a set of signals m from magnetic pick-up coils which surround the tokamak vacuum 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 ($\sim 10 \mu\text{s}$) 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 hardware⁷⁰ 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 non-linear 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 x^j 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 $p(x)$ of the data as a function of 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 memo-

ries 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 recognition⁸¹ and for prediction of future values of the time series⁸² 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 non-linear 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.

¹C. M. Bishop, *Neural Networks for Statistical Pattern Recognition* (Oxford University Press, Oxford, 1994).

²H. White, *Neural Comput.* **1**, 425 (1989).

³C. M. Bishop, Proc. IEE, Proceedings: Vision, Image and Speech; Special Issue on Neural Networks (1994).

⁴E. R. Kandel and J. H. Schwartz, *Principles of Neuroscience*, 2nd ed. (Elsevier, New York, 1985).

⁵Scientific American, special issue on Mind and Brain, September (1992).

⁶D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* (MIT Press, Cambridge, 1986), Vol. 2.

⁷W. S. McCulloch and W. Pitts, *Bull. Math. Biophys.* **5**, 115 (1943).

⁸D. O. Hebb, *The Organization of Behaviour* (Wiley, New York, 1949).

⁹F. Rosenblatt, *Psychol. Rev.* **65**, 386 (1958).

¹⁰F. Rosenblatt, *Principles of Neurodynamics* (Spartan Books, Washington, DC, 1962).

¹¹B. Widrow, *Self-Organizing Systems*, edited by G. T. Yovitts (Spartan Books, Washington, DC, 1962).

¹²B. Widrow and M. E. Hoff, *Adaptive Switching Circuits* (IRE WESCON Convention Record, New York, 1960), p. 96.

¹³B. Widrow and M. Lehr, *Proc. IEEE* **78**, 1415 (1990).

¹⁴M. Minsky and S. Papert, *Perceptrons* (MIT Press, Cambridge, 1959), also available in an expanded edition (1990).

¹⁵J. J. Hopfield, *Proc. Natl. Acad. Sci.* **79**, 2554 (1982).

¹⁶J. J. Hopfield, *Proc. Natl. Acad. Sci.* **81**, 3088 (1984).

¹⁷D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Nature* **323**, 533 (1986).

¹⁸D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* (MIT Press, Cambridge, 1986), Vol. 1.

¹⁹D. 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.

²²C. M. Bishop and C. M. Roach, *Rev. Sci. Instrum.* **63**, 4450 (1992).

²³C. M. Bishop, C. M. Roach, and M. G. von Hellerman, *Plasma Phys. Control. Fusion* **35**, 765 (1993).

²⁴K. Funahashi, *Neural Networks* **2**, 183 (1989).

²⁵G. Cybenko, *Math. Control, Signals Syst.* **2**, 304 (1989).

²⁶K. Hornick, M. Stinchcombe, and H. White, *Neural Networks* **2**, 359 (1989).

²⁷K. Hornick, *Neural Networks* **4**, 251 (1991).

²⁸V. Y. Kreinovich, *Neural Networks* **4**, 381 (1991).

²⁹A. R. Gallant and H. White, *Neural Networks* **5**, 129 (1992).

³⁰Le Cun Y *et al.*, *Neural Computation* **1**, 541 (1989).

³¹J. F. Kolen and J. B. Pollack, in *Advances in Neural Information Processing Systems* (Morgan Kaufmann, San Mateo, CA, 1991), Vol. 3, p. 860.

³²C. M. Bishop, *Neural Computation* **4**, 494 (1992).

³³H. Robbins and S. Monro, *Annu. Math. Stat.* **22**, 400 (1951).

³⁴J. Kiefer and J. Wolfowitz, *Annu. Math. Stat.* **23**, 462 (1952).

³⁵W. 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).

³⁶J. E. Dennis and R. B. Schnabel, *Numerical Methods for Unconstrained Optimisation and Non-linear Equations* (Prentice-Hall, Englewood Cliffs, NJ, 1983).

³⁷E. M. Johansson, F. U. Dowla, and D. M. Goodman, *Int. J. Neural Syst.* **2**, 291 (1992).

³⁸D. F. Shanno, *Math. Operations Res.* **3**, 244 (1978).

³⁹R. Battiti, *Complex Syst.* **3**, 331 (1989).

⁴⁰D. S. Broomhead and D. Lowe, *Complex Syst.* **2**, 321 (1988).

⁴¹J. Moody and C. L. Darken, *Neural Comput.* **1**, 281 (1989).

⁴²M. J. D. Powell, in *Algorithms for Approximations*, edited by J. C. Mason and M. G. Cox (Clarendon, Oxford, 1987).

⁴³C. A. Micchelli, *Constructive Approx.* **2**, 11 (1986).

⁴⁴E. Hartman, J. D. Keeler, and J. Kowalski, *Neural Comput.* **2**, 210 (1990).

⁴⁵J. Park and I. W. Sandberg, *Neural Comput.* **3**, 246 (1991).

⁴⁶S. Chen, S. F. N. Cowan, and P. M. Grant, *IEEE Trans. Neural Networks* **2**, 302 (1991).

⁴⁷J. MacQueen, in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability* (University of California Press, Berkeley, CA, 1967), Vol. 1, p. 281.

⁴⁸G. J. McLachlan and K. E. Basford, *Mixture Models: Inference and Applications to Clustering* (Marcel Dekker, New York, 1988).

⁴⁹A. P. Dempster, M. N. Laird, and D. B. Rubin, *J. R. Stat. Soc. B* **39**, 1 (1977).

⁵⁰G. H. Golub and W. Kahan, *SIAM Num. Analysis* **2**, 205 (1965).

⁵¹M. D. Richard and R. P. Lippmann, *Neural Comput.* **3**, 461 (1991).

⁵²D. W. Ruck *et al.*, IEEE Trans. Neural Networks **1**, 296 (1990).

⁷²L. Allen and C. M. Bishop, Plasma Phys. Control Fusion **34**, 1291 (1992).

⁵³P. O. Duda and R. E. Hart, *Pattern Classification and Scene Analysis*.

⁷³G. M. D'Alagni, *IEEE Trans. Neural Networks*.