2

Neural Networks for Modelling and Control[1]

2.1 INTRODUCTION

In recent times there has been a tremendous resurgence of interest in using biologically based models and learning algorithms for adaptive modelling and control. Intelligent Control (IC) applications requires algorithms which are capable of:

- **operating** in an ill-defined, time-varying environment;
- **adapting** to changes in the plant's dynamics as well as the environmental effects;
- **learning** significant information in a stable manner; and
- **placing** few restrictions on the plant's dynamics

in order to operate autonomously in hazardous environments with the minimum amount of intervention. Human learning appears to embody elements of *all* of these properties, and currently researchers are trying to endow machines with such qualities.

The search for an algorithm which provides a universal panacea for *all* the different IC problems is a tempting but unrealistic pursuit. The algorithms that are described in this chapter and in the remainder of this book are at best *initial approximations* to a human's information processing systems (if indeed human reasoning and learning can ever be described using an algorithm) and the biological implications are *not* considered. The Artificial Neural Networks (ANNs) described in this book are useful because their modelling and learning abilities can be analysed, and this is in *direct* contrast to human behaviour. Lau and Widrow [1990] hypothesised that "it may take another 50 years before we have a solid, complete microscopic, intermediate, and macroscopic view of how the brain works. Engineers can't wait that long". Similarly, Hecht-Nielson [1990] speculates that "the current level of understanding of brain and mind function is so primitive that it would be fair to say that not even one area of the brain or one type of mind function is yet understood at anything approaching a first-order level ... neurocomputing systems based upon these ideas probably have no close relationship whatsoever to

[1] An earlier version of this chapter appeared as Chapter 2 in Harris [1994].

the operation of the human brain". It is necessary to preface this introductory neural network chapter with such comments in order to emphasise that current neural network theories are far from providing a complete explanation of the operation of the human mind, and so current research can be divided into two (not necessarily distinct) categories:

- **devise** new theories about the brain's functionality;
- **application** to real-world problems.

There is constant cross-stimulation between these two research fields, but the second area should only use neural theories *if* they have something more to offer than a conventional, non-neural solution. In the past, neural algorithms have been applied to modelling and control problems without any consideration of their suitability and whether any other algorithm may be more appropriate (exactly the same comments can be made about fuzzy systems as well).

Learning algorithms have much to offer the control engineer. It is hoped that increased adaptation will result in improved system performance; increasing the quality of the solution and reducing the design and operational cost, although the current reality is far removed from this ideal. Generally the adaptive algorithms are based around *linear* plant and controller models, and a number of parameters must be chosen which determines the flexibility of the adaptation schemes. Neural networks provide *one* method with which these algorithms can be applied to nonlinear systems, although it is not the only approach and some of the "neural" learning algorithms seem primitive in comparison with the techniques developed in the adaptive control/signal processing fields. For instance, the majority of supervised learning algorithms are gradient based and it is only recently that adaptive strategies based on stability concepts have appeared [Sanner and Slotine, 1992, Tzirkel-Hancock and Fallside, 1991], mimicking the adaptive control field in the sixties, [Åström and Wittenmark, 1989, Narendra and Annaswamy, 1989].

The approach taken in this book is to evaluate the ANNs from an engineering viewpoint; the modelling capabilities are analysed *separately* from the learning algorithms. It is often claimed that the majority of ANNs are model-free estimators, but it is the authors' view that these comments are generally misinformed, as many of the neural models currently used have a *fixed* network structure and use (nonlinear) gradient descent rules to adapt the parameters. The network's structure may be very flexible due to the nonlinear adaptation, but it is *model based*, and these networks are termed *soft* or *weak* modelling schemes to distinguish them from conventional linear adaptive algorithms. Therefore it is important to examine the modelling capabilities of different ANNs, to determine what functional, representational and generalisation properties abilities they possess.

The supervised and unsupervised learning algorithms are then examined and it is shown that many of these adaptive rules can be applied to a wide range of different neural (and fuzzy, see Chapter 10) systems. The development of new learning algorithms can generally proceed *independently* of the model to which it is applied, after which the suitability of a learning algorithm for training a *particular*

model should be assessed. For instance, in Section 4.5 it is argued that although nonlinear, gradient descent rules can be used to train a Multi-Layered Perceptron (MLP), the basic optimisation problem is badly conditioned and it may be more suitable to use optimisation strategies which incorporate second-order information about the performance surface [Gill *et al.*, 1981].

The final section of this chapter reviews several algorithms for assessing the trained ANN. This is a neglected topic in the neural literature, although it forms a critical part of any design methodology if these networks are to be applied in areas where *safety*, *correctness* and *certification* are prime concerns. The information provided by measuring the network's performance using the training set is described within the framework of the bias/variance dilemma, which states that the network's structure should be constrained such that it is unable to model any noise. Various forms of network testing are also proposed, based on constructing test sets, correlation-based modelling algorithms, chi-squared tests and network interrogation. The first three algorithms can be applied to nonlinear systems, and were first proposed for assessing nonlinear Volterra models, although they have been applied to ANNs as well [Billings and Chen, 1992, Billings and Voon, 1986]. The last point investigates the network's *transparency*: How easy is it to understand the knowledge stored in an ANN? Training sets rarely contain a complete description of the desired input/output relationship and once learning has ceased, it may be necessary to modify the stored information. This can only be performed if the knowledge is stored in a transparent fashion.

The class of ANNs studied in this book are Associative Memory Networks (AMNs) which are feedforward, supervised ANNs. These networks are universal approximation algorithms which can incorporate *a priori* knowledge in their structure, are suitable to be trained using instantaneous gradient descent algorithms and have a natural fuzzy interpretation, which makes the knowledge stored in the network transparent to the designer. This class of network is studied in detail from Chapter 3 onwards, although it is introduced in this chapter by considering the learning properties of an adaptive look-up table. The advantages in using these networks are that they allow conventional linear learning theory to be applied to a wide range of nonlinear modelling and control problems, and enable *a priori* functional knowledge to be incorporated into the network's structure.

2.2 NEUROMODELLING AND CONTROL ARCHITECTURES

Before the neuromodelling and control algorithms are described, it is useful to have an understanding of how the basic learning modules may be applied. These modelling and control architectures are generally network *independent*; most learning algorithms can be used, although some may be more suitable than others. The degree with which a particular learning algorithm satisfies these properties determines its suitability for on-line learning modelling control. It does not solely

depend on the network's modelling capabilities or on the learning algorithm, bu a combination of these two factors.

2.2.1 Representational Issues

Many neuromodelling and control algorithms are expressed in the continuous time domain, using measured variables which assess the state of the plant, the control signal and the desired plant's response, in order to predict the change in the plant's state (model) or to calculate the required change in control signal necessary to make the plant behave as required (controller). However the vast majority of neuromodelling and control applications are implemented as sampled systems, and the two sets of state equations for these two single input, single output, nonlinear systems are:

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{f}(\mathbf{x}(t), u(t)) \\ y(t) &= g(\mathbf{x}(t)) \end{aligned}$$

where $\mathbf{x}(t)$ is the vector of plant states at time t, $u(t)$ is the current control signal and $y(t)$ is the observable plant output. The corresponding discrete time state equations are:

$$\begin{aligned} \mathbf{x}(t+1) &= \mathbf{f}(\mathbf{x}(t), u(t)) \\ y(t) &= g(\mathbf{x}(t)) \end{aligned}$$

The majority of plant models assume that the output signal is sufficiently rich to contain information about all of the plant's states, so the above discrete time, state equation may be reformulated as:

$$y(t) = h(\mathbf{y}(t-1), \mathbf{u}(t)) \tag{2.1}$$

where $\mathbf{y}(t-1)$ is a vector of length n_y formed from the past outputs $y(t-1), \ldots, y(t-n_y)$ and $\mathbf{u}(t)$ is a vector of length n_u formed from the past and current control actions $u(t), \ldots, u(t-n_u)$.

Once a discrete or a continuous representation has been decided, the order of the plant (the delays n_y and n_u) must be determined. Over-estimating their values results in poor convergence rates and generalisation as the model is over-parameterised, although choosing too small a value means that unmodelled dynamics exist that may affect the stability of any learning control system. It has been claimed the adaptive neurofuzzy systems can be used when the order of the plant is underestimated, and although this can occur for certain controllers, neurofuzzy mappings are simply nonlinear functions and if the information which guarantees stability is not available in the input vector, the control loop can become unstable. A large body of theory has been developed for choosing these quantities when these are linear mappings, and in Section 8.5, several iterative construction algorithms which automatically determine which variables are important are presented.

NARMAX Representation

During the eighties, Billings and his collegues [1986, 1989, 1992] developed a general nonlinear modelling structure called Nonlinear AutoRegressive Moving Average models with eXogenous inputs (NARMAX), and a range of correlation and statistical tests that can assess the adequacy of the network. A NARMAX model is described by:

$$\mathbf{y}(t) = \mathbf{h}(\mathbf{y}(t-1), \ldots, \mathbf{y}(t-n_y), \mathbf{u}(t-1), \ldots, \mathbf{u}(t-n_u)) + \mathbf{e}(t) \tag{2.2}$$

where $\mathbf{y}$, $\mathbf{u}$ and $\mathbf{e}$ are system output, input and additive disturbance vectors respectively. This is a very general relationship and many ANNs can be interpreted in this form. Volterra models and polynomials were first used for modelling nonlinear dynamic processes [Billings and Voon, 1986, Billings *et al.*, 1989], although more recently multi-layer perceptrons [Billings *et al.*, 1992] and radial basis function [Chen *et al.*, 1990] neural networks have been developed within the same modelling framework. The network performance measures can determine deficiencies in the input data as well as in the representation formed by the network (see Section 2.5.3), so that this type of theoretical framework is extremely useful for developing neuromodelling and control algorithms.

2.2.2 Modelling Strategies

There are four principal architectures which can exploit a learning modules modelling ability: as a basic plant model, an inverse plant model, a specialised inverse plant model and an operator model [Hunt *et al.*, 1992, Tolle and Ersü, 1992, Widrow and Stearns, 1985], as shown in Figure 2.1. For three of these four cases, the desired value of the network's output is *directly* available and any supervised learning rule can be used to train the weight set. The error in the specialised inverse plant modelling algorithm is formed at the *output* of the plant, whereas the network's output forms the *input* to the plant. Therefore some method is required for feeding back the plant output error, in order to train the inverse model and this is discussed later in this chapter and in Chapter 11. The success of all these schemes depends on the input signal being sufficiently exciting, in order to provide training data across the whole of the network's input space, the approximation capabilities of the network and the ability of the training rule to filter out the measurement and modelling error/noise.

Plant Modelling

A plant model may be required for a variety of reasons: to use within a larger feedback control loop which requires an estimate of the plant output, for predicting the performance of the plant when the true output is unavailable (due to time

Direct Inverse Plant Modelling

The objective with inverse plant modelling is to formulate a controller, such that the overall controller/plant architecture has a *unity* transfer function. Inevitably, modelling errors perturb the transfer function away from unity, although the use of such an inverse model as a feedforward precompensator in addition to a standard, linear feedback controller generally provides good performance for a wide range of nonlinear plants.

For the inverse model to be well defined, the training examples must be unique. This is satisfied when the plant is invertible or if the training data for a non-invertible plant are contained in a restricted area of the input space, so that the plant is locally invertible. However, care must be taken when using this approach for plants whose Jacobian varies significantly and when the modelling error does not tend to zero. This is because the network minimises the MSE in the *control* space rather than the plant output space, through using the cost function:

$$J_u = E\left(\epsilon_u^2(t)\right) \tag{2.5}$$

where $\epsilon_u(t) = \widehat{u}(t) - u(t)$, $u(t)$ is the control output of the network, and $\widehat{u}(t)$ is the measured control signal. To a first approximation, the error in the control signal (for a single-input, single-output plant) is related to an error in the plant output by:

$$\epsilon_y(t) \approx \frac{dy(t)}{du(t)}\epsilon_u(t)$$

where $\frac{dy}{du}$ is the plant's derivative, or its *Jacobian*. Thus the two cost functions given in Expressions 2.4 and 2.5 are approximately related by:

$$J_y \approx E\left(\left(\frac{dy(t)}{du(t)}\right)^2 \epsilon_u^2(t)\right) \tag{2.6}$$

When the plant is nonlinear, the value of the Jacobian varies and different weightings are applied to the control errors. The cost functions are *not* equivalent, in the sense that one is simply a linearly scaled version of the other, and the designer should be aware that minimising one performance measure does not necessarily mean that the other is also minimised. If the output errors are uncorrelated with the plant Jacobian though, this expression simplifies to:

$$J_y \approx E\left(\left(\frac{dy(t)}{du(t)}\right)^2\right) J_u \tag{2.7}$$

and the effect of the Jacobian can be incorporated into the learning rate.

Specialised Inverse Plant Modelling

This approach again aims to provide an inverse model/plant structure which has a unity transfer function, although the method proposed is very different. A forward plant model is first constructed, and the difference between the plant's response and the desired output is used to provide an error signal, which is passed back through the forward plant model in order to adjust the inverse model's parameters [Jordan and Rumelhart, 1991, Psaltis *et al.*, 1988]. The main advantage which this approach has over the previous algorithm is that it is *goal driven.* For on-line applications, the plant output error causes the inverse model to move into previously unexplored regions of the input space, whereas the direct inverse modelling approach can only learn if the control signal is sufficiently exciting.

The learning algorithm attempts to minimise the *plant* output MSE, whereas the previous inverse modelling approach minimised the MSE *control* effort. These two quantities are approximately related by the Expressions 2.6 and 2.7, and as previously discussed if the plant is nonlinear and there exists a mismatch between the true inverse plant and the adaptive model, the direct and specialised inverse modelling approaches are *not* compatible as the optimal parameter values are generally different. As noted by Psaltis *et al.* [1987], "though there may be some benefit to performing generalised (direct) training prior to specialised training, these simulations show no clear advantage to doing so". An example is given in Brown and Harris [1993] which shows the different optimal weights, and the substantially different rates of convergence of these two modelling algorithms when the plant's gains are not close to unity and there exists modelling mismatch.

Although the inverse modelling architectures can be used to synthesise controllers, they may not be as robust as alternative learning controllers, due to the lack of feedback information [Hunt *et al.*, 1992]. This can be partially overcome by introducing on-line, inverse model adaptation, although the comments made in the previous paragraph should be taken into account.

Operator Modelling

Synthesising a controller by learning from an expert has many potential applications within the IC field [Kraiss and Küttelwesch, 1990, Shepanski and Macy, 1987, Widrow, 1987]. The learning algorithm is run in parallel with a skilled plant operator and their response forms the desired network output which is then used to train the network. This training signal typically contains a large amount of noise, due to the operator using different actions for similar inputs, and so this signal may have to be filtered [Guez and Selinsky, 1988] before the conventional network learning algorithms can be applied.

As with all modelling strategies, care must be taken to ensure that the training set contains sufficiently rich examples from the relevant operational domain, and that the network input vector contains all the information which is available to the

operator. In Section 9.2, this approach is used to construct a set of fuzzy-type rules which can reverse a vehicle into a slot, and both of these points are illustrated. The supplied training data are very noisy and are distributed only in a small part of the input domain, therefore new rules had to be initialised to cope with different initial conditions. Also a new input variable had to be introduced in order to distinguish between similar situations which require very different actions. The human operator *implicitly* used this information when parking the vehicle, but it needs to be *explicitly* set for the network.

2.2.3 Supervised Control Architectures

Low-level learning algorithms need to be posed within specific modelling control and architectures, and some of the most popular are described in this section. One of the problems in formulating an on-line learning controller is that the desired control signal is rarely available, and generally only the desired plant output can be used to train the controller.

There are two distinct approaches which have been formulated in the adaptive control field: *direct* and *indirect* schemes [Åström and Wittenmark, 1989]. A direct adaptive control scheme builds an explicit model of the desired controller, whereas an indirect scheme produces a model of the plant and synthesises the control law, using a predefined optimisation/inversion calculation. For instance, the majority of self-organising fuzzy controllers have been direct, as a fuzzy rule base is used as a controller and there exists a performance index which relates errors in the plant's output to errors in the control signal in order to update the rule base. In contrast, most of the adaptive neurocontrol schemes have been indirect, as an explicit plant model is generally constructed, to be used in a predictive control algorithm for example.

Fixed Stabilising Controllers

One of the simplest *direct* learning control schemes is shown in Figure 2.2, where a fixed, stabilising, linear, feedback controller is used to train a learning network [Kraft and Campagna, 1990, Miller, 1987, Miyamoto *et al.*, 1988]. The linear controller is designed so that the closed-loop system is stable in every operating region and the control signal provides a training signal for the learning module. The performance of the closed-loop system depends on the current operating point, although the iterative training of the network gradually improves its performance (and the performance of the control signal) *on-line*. As the operating point changes, the learning controller builds up a nonlinear model of the desired control surface, such that when the plant returns to the original operating point, the learnt response has not been forgotten and it can be improved upon. This requires a learning module which is *temporally stable*; learning about one area in the input space affects

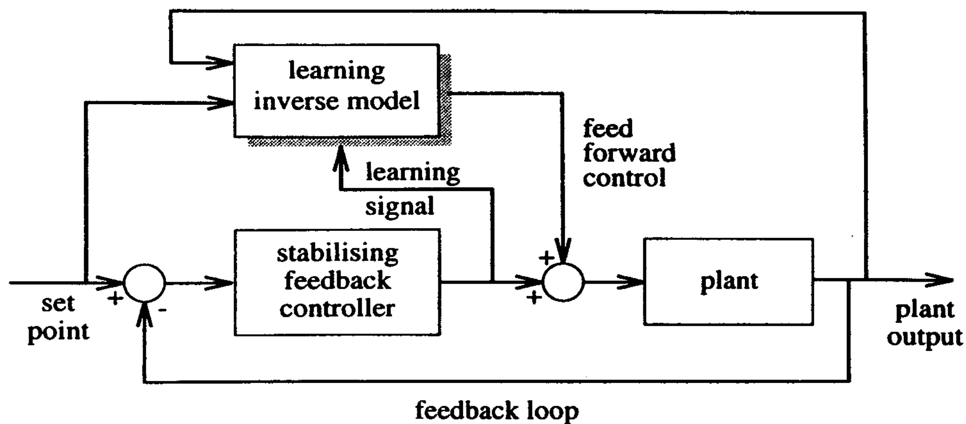

Figure 2.2 A direct learning controller; the fixed linear feedback controller is used to stabilise the system and to provide a training signal for the learning controller.

the knowledge stored in a different region minimally.

Despite the algorithm's simplicity, this approach has one main drawback; the design of the fixed linear controller. It has been claimed that the algorithm is robust with respect to the design of this controller, although the rate of convergence of the learning module depends on the quality of the training signal. A learning module is slow to adapt when the linear controller is performing poorly.

Predictive Learning Control Schemes

Indirect predictive learning control schemes attempt to formulate a control strategy by assessing the effect of their actions for many time-steps into the future and selecting the current "optimal" control action, which is then applied to the plant [Montague *et al.*, 1991, Saint-Donat *et al.*, 1994, Tolle and Ersü, 1992]. The architecture requires the development of a plant model, a performance function to evaluate the effect of a control action and an optimisation technique which can determine the best control action. This is illustrated in Figure 2.3, where a learning control element has also been included, so that after sufficient training, the full optimisation calculation does not need to be calculated and the computing resources can be allocated to other tasks. If the plant is time-varying, the model is generally adaptive, although the initial optimisation calculations may give very poor closed loop control if the process model is poor.

When the plant model is good and the performance function and the search strategy are appropriately chosen, this control scheme can provide excellent closed-loop control. However, the multi-step ahead optimisation calculation is generally very expensive and is only applied to systems which are not time-critical. Many simplifications of the above architecture can be performed which makes this technique more suitable for real-time control tasks, some of which are described in Tolle and Ersü [1992].

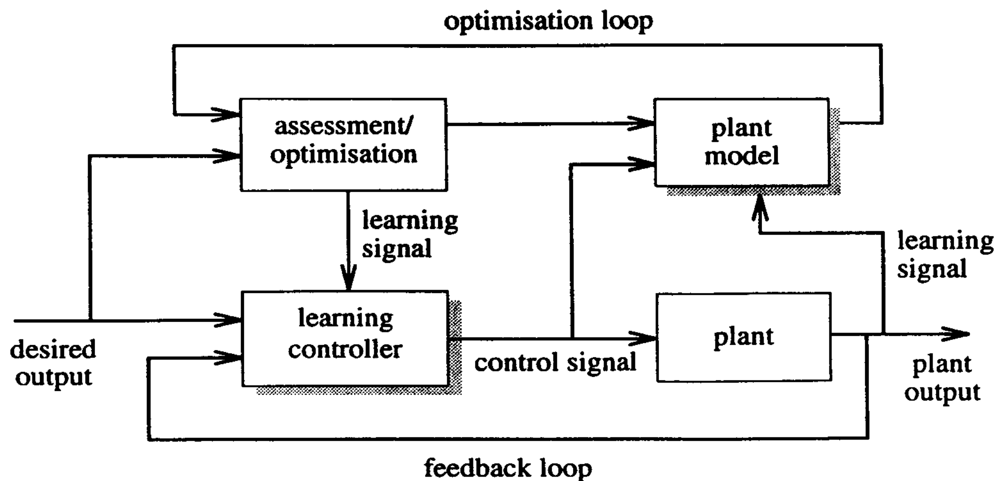

Figure 2.3 A learning predictive control architecture.

Model Reference Adaptive Control

The model reference learning control architecture has been widely used in the linear adaptive control field [Åström and Wittenmark, 1989], and is shown in Figure 2.4. The control objective is to adjust the control signal in a *stable* manner so that the

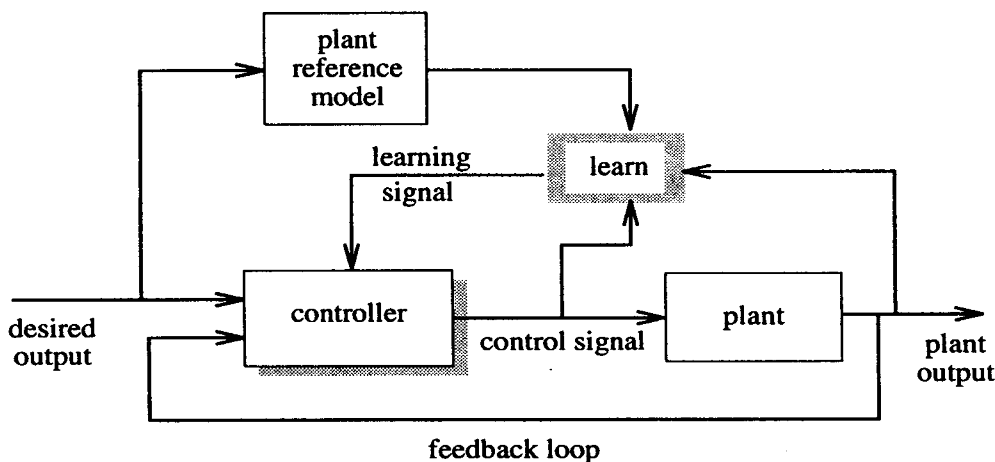

Figure 2.4 A model reference control architecture.

plant's output, $y^p(t)$, asymptotically tracks the reference model's output, $y^r(t)$, i.e.:

$$\lim_{t\to\infty} \|y^r(t) - y^p(t)\| \leq \varepsilon$$

where ε is a small positive constant [Narendra and Parthasarathy, 1990]. The performance of this algorithm depends on the choice of a *suitable* reference model and the derivation of an appropriate *learning* mechanism. Researchers in the sixties found that simple gradient-based learning rules were sometimes insufficient and

there is no reason why this should not also be the case for more general nonlinear plant models and controllers.

Internal Model Control

Internal model control [Hunt and Sbarbaro-Hofer, 1991, Hunt *et al.*, 1992] uses a similar structure to the predictive learning control scheme, as shown in Figure 2.5. A (learning) module is used to model the process directly, receiving the applied control signal, rather than the reference signal which is used in the model reference adaptive control scheme. The error between the model and the measured plant output is used as a feedback signal and this is passed to the controller. The internal model controller is generally designed to be an inverse plant model (when it exists), and either of the inverse modelling schemes described in the previous section can be used to synthesise an appropriate controller.

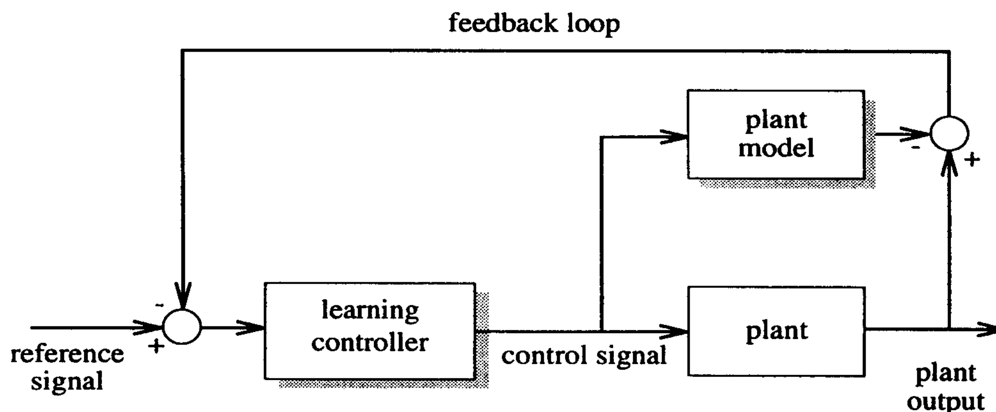

Figure 2.5 An internal model control architecture.

Many theoretical stability results about internal model control loops are available, [Hunt and Sbarbaro-Hofer, 1992, Sbarbaro-Hofer *et al.*, 1993], although they generally make assumptions the open-loop stability of the system, exact modelling and/or inverse modelling. Despite these assumptions, it is claimed that this approach extends readily to nonlinear systems and yields to robustness and stability analysis.

2.2.4 Reinforcement Learning Systems

Reinforcement learning schemes and ANNs have been very closely linked since the seminal paper by Barto *et al.* [1983]. Reinforcement control schemes are *minimally* supervised learning algorithms; the only information that is made available is whether or not a particular set of control actions has been successful. The original application attempted to balance an inverted pendulum, subject to the

constraints that the platform should not move more that a certain distance from its starting point and that the inverted pendulum remained approximately upright. If either of these constraints were violated, a failure signal was sent to the learning algorithms.

From this definition it is clear that once the controller has managed to balance the inverted pendulum fairly well, very little training takes place as failures occur infrequently. The solution proposed by Barto *et al.* [1983] was to construct a control scheme which is composed of two adaptive elements; an Associative Search Element (ASE) and an Adaptive Critic Element (ACE). The ASE attempts to reproduce the optimal control signal which satisfies the given performance objectives, while the ACE attempts to monitor the performance of the controller *internally* and to provide an internal reinforcement signal which is used to train the ASE, as illustrated in Figure 2.6. The ACE is trained using the external failure/success signal. This continuous internal training of the control element has been shown to improve vastly the performance of the overall system.

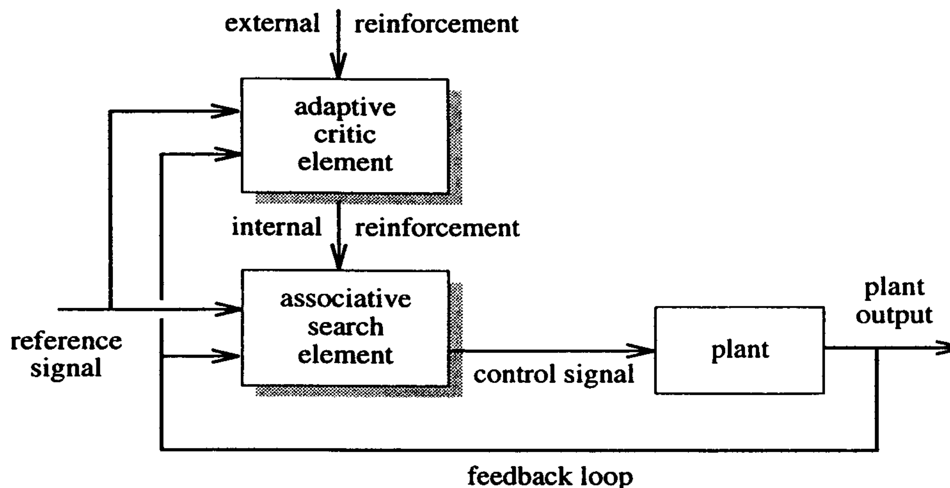

Figure 2.6 An ASE/ACE reinforcement system's architecture.

Over the past ten years, there has been a greater theoretical understanding of the overall system [Barto *et al.*, 1993, Sutton *et al.*, 1992], as well as a growing number of simulations and applications that use modified versions of this technique [Berenji and Khedkar, 1992, Millington and Baker, 1990, Porcino and Collins, 1990, Shelton and Peterson, 1992].

2.2.5 Parameterising Linear Controllers

Many different neuromodelling and control architectures have been proposed in recent years, and the previous sections have described several which are related to conventional control schemes. The novel parameter initialising neurocontrol architectures which are now described have all been developed as a result of the recent

resurgence of interest in ANNs, as they attempt to exploit the ability of these adaptive systems to learn an arbitrary functional relationship. There are many reasons for utilising an intelligent *gain-scheduling* type approach: widespread industrial acceptance of linear feedback controllers, many theoretical and practical results are available about robustness and closed-loop stability and their low implementation cost. The neurocontrol schemes in this section attempt to exploit these properties and to produce algorithms which can calculate the parameters for both off-line and on-line control. Successful systems would result in reduced commissioning costs and possess the ability to adapt to time-varying process dynamics.

All of these approaches assume some previous knowledge about the plant's structure, as this simplifies the problem. In Kumar and Guez [1991], an indirect control design architecture is adopted. The plant is assumed to be a slowly varying second-order linear system and a set of *features* which describe the closed-loop response of the plant are extracted. These features could include the delay time, rise time, peak overshoot, settling time, etc., and are passed to an Adaptive Resonance Theory (ART) based classifier which predicts the parameters of the plant. This output, together with a set of desired closed-loop response characteristics, is used to produce a set of linear control gains using conventional pole placement design techniques.

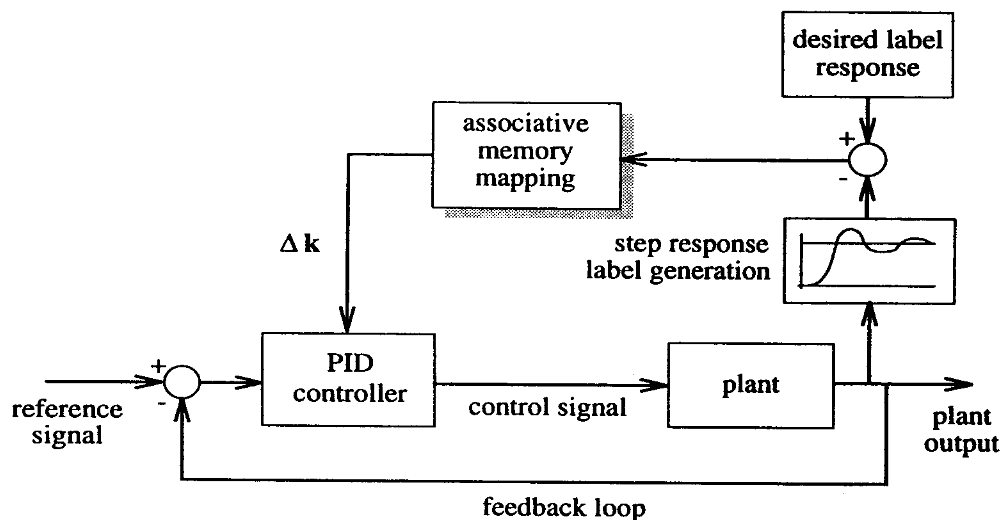

Figure 2.7 An architecture for predicting the required change in a PID controller directly based on a set of labels that describe the system's closed-loop response.

The idea of extracting the feature labels which describe the closed-loop system's response has also been independently proposed in Lawrence and Harris [1992]. The desired closed-loop response is expressed as a set of feature labels, which are then compared with the labels describing the system's closed-loop step response. This error label vector is passed to the ANN which predicts a change in the PID parameters so that the system's response will be closer to the desired one, as shown in Figure 2.7. A similar approach has also been proposed in Ruano *et al.* [1992],

although this algorithm can use the step responses of both open- and closed-loop systems, and the ANN outputs the PID parameters which are optimal with respect to the Integral of Time multiplied by the Absolute Error (ITAE).

The ability to synthesise a set of PID parameters on-line, using only input/output data, has been investigated for many years [Åström and Hägglund, 1988]. The potential payback from such a system which can increase the robustness of PID controllers is large, and this research area is still in its infancy.

2.3 NEURAL NETWORK STRUCTURE

In the introduction, it was emphasised that the majority of ANNs are model based, and the structure of several such models are now described, compared with each other and contrasted with some truly *model-free* estimators. The learning capabilities of the networks are discussed in Section 2.4, as this is an important topic but is *separate* from the model description. Too often in the past Multi-Layer Perceptrons (MLPs) have been criticised for being slow to converge, when what is really meant is that MLPs *trained* using gradient descent algorithms learn slowly. The model structure *influences* the selection of the training rule, although the learning algorithm does not generally affect the flexibility of the underlying model. To begin this section, the adaptive linear combiner which forms part of most ANNs is described and its properties are discussed.

2.3.1 Linear Combiners

The Adaptive Linear Combiner (ALC) formed part of the two earliest ANNs: the ADALINE [Widrow and Lehr, 1990] and the Perceptron [Rosenblatt, 1961], and it is still used in the many of the present neural networks. The ALC simply forms a weighted sum of the inputs, and this quantity can be thresholded to produce a binary output if the network is used for pattern classification tasks. Let the p-dimensional network input vector be denoted by $\mathbf{x}$ and the weight vector by $\mathbf{w}$, the continuous ALC's output y is given by:

$$y = \sum_{i=1}^{p} a_i w_i = \mathbf{a}^T \mathbf{w} \tag{2.8}$$

where $\mathbf{a} \equiv \mathbf{x}$, and the ALC is shown in Figure 2.8. Generally an augmented term a_0 is set equal to 1 and is known as the *bias*. The thresholded network output is:

$$y = \begin{cases} 1 & \text{if } \mathbf{a}^T \mathbf{w} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

The Perceptron also has an additional layer where the network's input is preprocessed ($\mathbf{x} \rightarrow \mathbf{a}$), and a weighted combination of these modified inputs is taken.

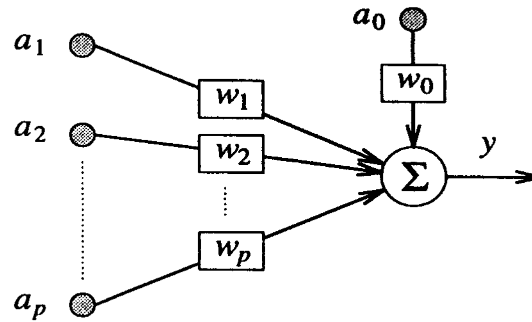

Figure 2.8 A basic adaptive linear combiner.

There are many ways in which this initial nonlinear transformation can be chosen and some of these are discussed in Chapter 3.

The weight vector, $\mathbf{w}$, is adjusted using an error correction learning rule and training ceases when the network's overall behaviour is acceptable. The network only achieves the correct result when the set of (modified) training input examples $\{\mathbf{a}(t)\}_{t=1}^{L}$ is *linearly separable*, and in this case the learning algorithm terminates in a *finite* number of iterations. A set of training examples is linearly separable if there exists a $(p-1)$-dimensional hyperplane which can separate the training inputs in the p-dimensional input space. This hyperplane is formed from the set of weights which satisfy:

$$0 = \mathbf{a}^T \mathbf{w}$$

as this determines the classification boundary for the ALC. Figure 2.9 shows a linearly separable training set and illustrates the fact that, if there exists a single weight vector which can linearly separate the training inputs, there exists an infinite number of such hyperplanes.

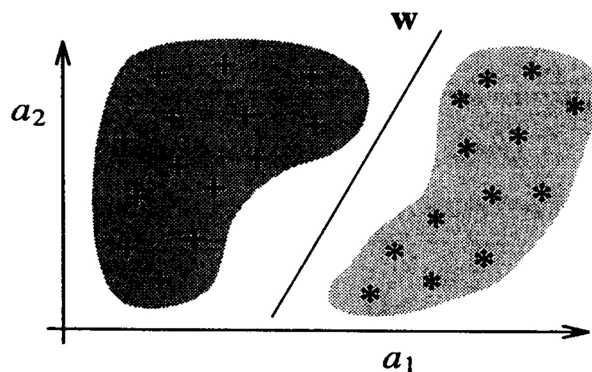

Figure 2.9 A linearly separable training set for a two example classification problem.

Most of the two-dimensional logical functions, AND, OR, NOT can be implemented in the above ALC framework. However, it was shown in Minsky and

Papert's book *Perceptrons* [1969], that the basic two-dimensional XOR logical function cannot be implemented within the standard Perceptron architecture. The result of this publication was effectively to halt much of the research into ANNs during the seventies.

2.3.2 Multi-Layer Perceptrons

Despite the limitations of the ALC, it was well known in the sixties that multi-layered networks could implement exactly the XOR and higher order logical functions, although there appeared to be no natural generalisation of the Perceptron's training algorithm and these results were of theoretical interest only.

A multi-layer ANN is a *feedforward* network where the input signal is propagated forwards through several processing layers before the network output is calculated. Each layer is composed of a number of *nodes*, and each node is (generally) composed of a simple ALC, with an appropriate transfer function which calculates the node's output from the weighted input signal. Each node has input connections with the nodes in the previous layer *only*, and the node's output is transmitted to the nodes in the next layer, as shown in Figure 2.10. Every node has an associated weight vector which *linearly* transforms its input vector.

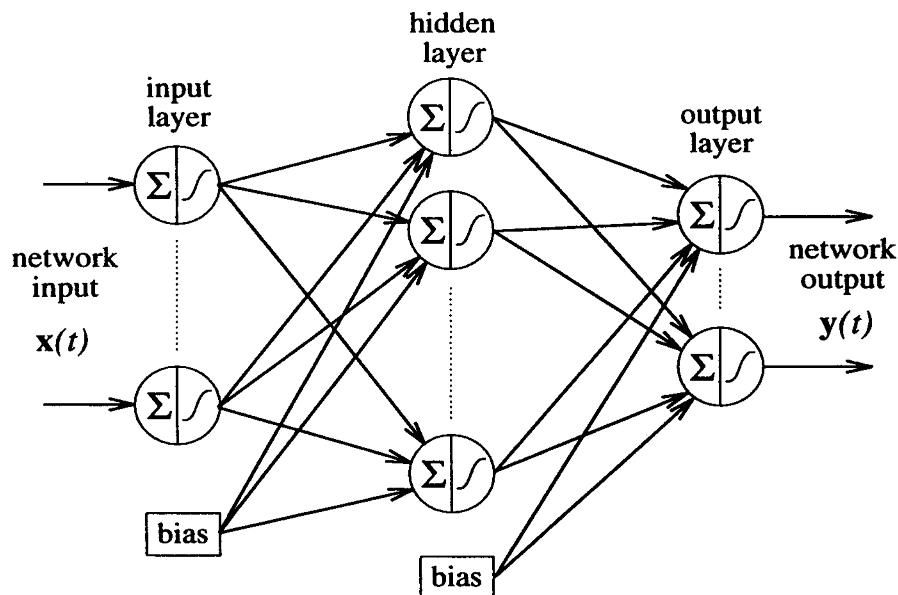

Figure 2.10 A multi-layer feedforward network.

During the recent ANN revival, a number of researchers independently derived a gradient descent algorithm suitable for training these Multi-Layer Perceptron (MLP) networks, [Le Cun, 1985, Parker, 1985, Rumelhart and McClelland, 1986, Werbos, 1974]. The transfer function in each node of these networks is a bounded, *continuously* increasing nonlinearity, rather than a binary threshold. Thus the net-

work output is a continuous (continuously differentiable) function of every weight in the network, enabling it to be trained using gradient descent rules. The availability of such learning algorithms popularised the MLP, and at the time of writing it is probably the most widely used ANN. The model structure does not depend on the learning rule, although the rate of convergence of the learning algorithm depends on the model structure, and the quality of the final model also depends on the learning rule. For the remainder of this section, the MLP structure is discussed *without* reference to a particular the learning algorithm.

XOR solved by Multi-Layer Networks

The two-input XOR problem can be solved exactly by a three-layer MLP (one input, one hidden and one output layer) as shown in Figure 2.11. The hidden layer nonlinearly transforms the inputs into an *alternative* space in which the training

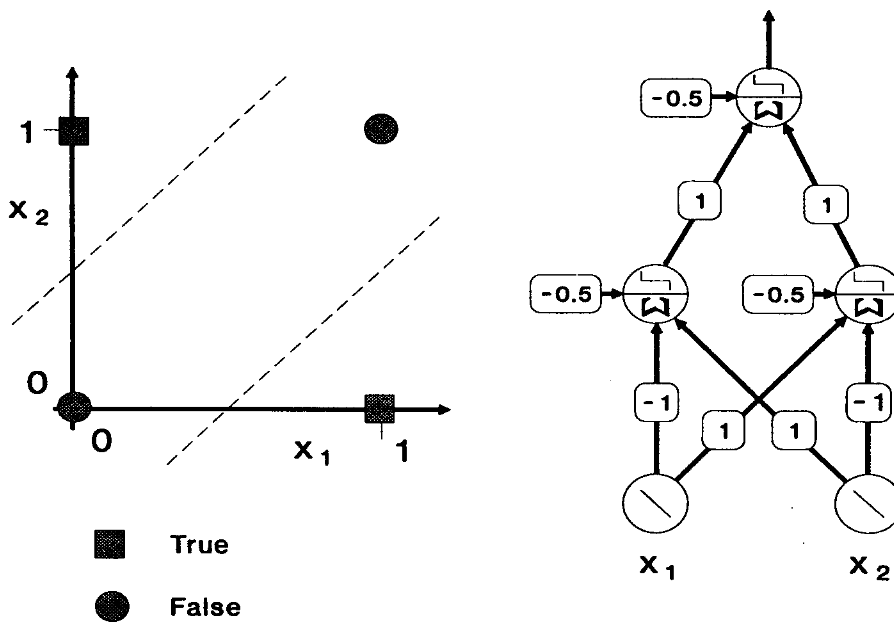

Figure 2.11 A solution for the two-input XOR logical problem using a three-layer MLP with two nodes in the hidden layer.

samples are linearly separable and a correct classification can be achieved. For the network shown, the outputs of the hidden nodes correspond to the logical functions:

(NOT x_1) AND x_2

x_1 AND (NOT x_2)

and if either of these expressions is true (logical OR in the output layer), the MLP output is also true. This construction holds for any finite dimensional log-

ical expression, as it is possible to reduce *any* Boolean function to its equivalent *disjunctive normal form.* Thus any Boolean function can be represented by a three-layer network, where the output layer represents a multi-dimensional OR and the hidden layer nodes form multi-dimensional logical ANDs of the (possibly negated) inputs.

Functional Approximation

Using a continuous transfer function in each node means that the output is continuously dependent on the network's inputs, and there has been a lot of interest in using the MLP for functional approximation rather than classification tasks. It has been established that *any* continuous nonlinear function can be approximated to within an arbitrary accuracy by a three-layer MLP with sufficient nodes in the hidden layer [Hornik *et al.*, 1989]. Therefore the basic structure of the MLP is very flexible and can be employed in a wide variety of modelling and control tasks.

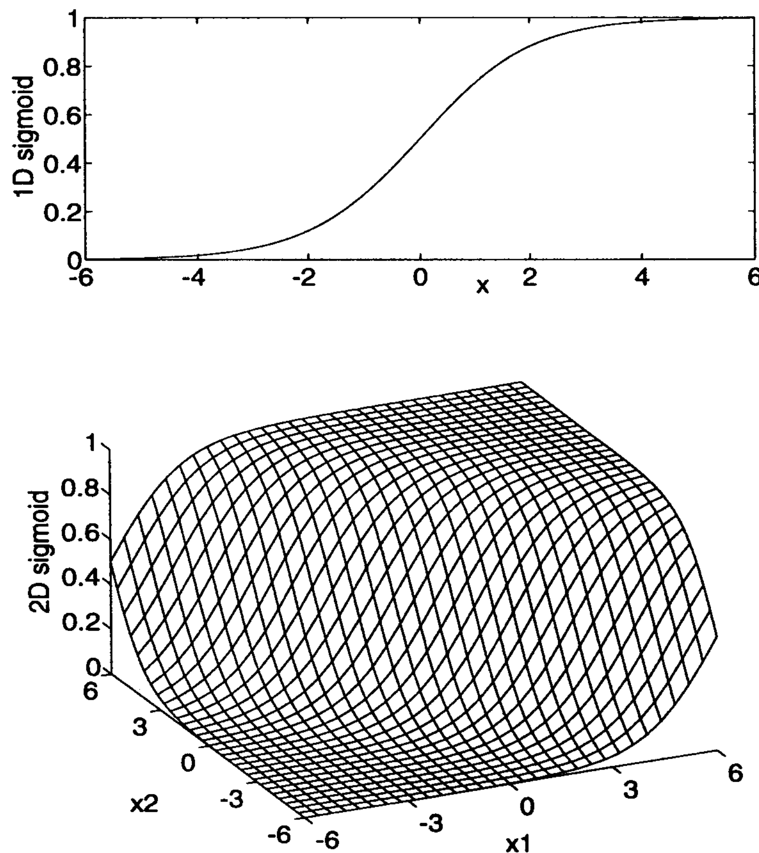

Figure 2.12 The sigmoid's output on a one- and two-dimensional input space.

It is instructive to investigate the nonlinear transformation that occurs in the hidden layer nodes, as this gives an indication of the type of problem for which the

MLP might be used successfully. Consider the commonly used *sigmoidal* transfer function:

$$f(u) = \frac{1}{1 + \exp(-u)} \qquad \in (0, 1) \tag{2.10}$$

where $u = \mathbf{x}^T\mathbf{w}$. This function is bounded and monotonically increasing, tending to 0 as $\mathbf{x}^T\mathbf{w} \to -\infty$, and approaches 1 as the linearly combined input tends to ∞. The output of this function for a one- and two-dimensional input is shown in Figure 2.12, and is constant along the lines (three-dimensional weight space) for which:

$$w_0 + w_1x_1 + w_2x_2 = c$$

for some constant c, and this generalises to n-dimensional input spaces. The output of a sigmoid in the hidden layer is constant along the $(n-1)$-dimensional hyperplanes given by:

$$w_0 + \mathbf{x}^T\mathbf{w} = c$$

Thus the nodes which are composed of an ALC and a sigmoidal-type transfer function are termed *ridge functions* [Mason and Parks, 1992], as the output is *constant* along hyperplanes in their input space. If the desired function can be concisely decomposed into similar ridge functions, MLPs may be suitable models.

Sometimes in modelling and control applications the input data are *redundant* and MLPs can model this relationship by constructing hyperplanes parallel to the redundant inputs and setting the appropriate weights to zero. Thus the network model can deal efficiently with redundant data, and if a suitable network has been constructed and the input space is expanded by introducing a new, redundant input variable, no new nodes in the hidden layer need be introduced. Only a small number of weights are necessary to increase the model's size and these would be all set to zero. Thus the model's structure can incorporate redundant data *efficiently*, although it might not easily learn to recognise this redundant information.

MLPs are generally unsuitable for modelling functions which have significant local variations. The output of all of these hidden layer nodes is generally non-zero, and the resulting optimisation problem can be very complex. The theoretical modelling results guarantee that an MLP can approximate such functions arbitrarily closely, although they provide no indication about the suitability of using ridge functions as opposed to other nonlinearities in the hidden layer. Recently there has been a lot of theoretical interest in using the *Vapnik-Chervonenkis dimension* (VCdim) to investigate the complexity of MLPs [Hush and Horne, 1993], and once this number is known, it can be used to determine the amount of training data necessary for good generalisation. A realistic rule of thumb that came about from this work is that the amount of training data should be approximately ten times the VCdim, or equivalently, the number of weights in an MLP.

In conclusion, MLPs can be successfully applied to high-dimensional functional modelling and classification problems *if* the training data have redundant inputs

and the desired mapping can be approximated by a low number of ridge functions [Wright, 1991].

2.3.3 Functional Link Networks

A functional link ANN [Pao, 1989, Pao *et al.*, 1994] has a similar structure to the three-layer MLP, except that instead of employing ridge functions in the hidden layer, polynomial or trigonometric terms are used and *linear* nodes are used in the input and output layers. The use of such hidden layer transfer functions has a long history in the nonlinear modelling community where a small number of low-order polynomial terms or the dominant terms in a Fourier series have been used to introduce nonlinearities into conventional linear algorithms. The ALC no longer forms part of the nodes in the hidden layer, as shown in Figure 2.13, but is used in the output layer. The use of such nonlinearities produces a very flexible network [Mathews, 1991], although its usefulness for a particular application depends on how well these nodes represent the nonlinear components of the desired function.

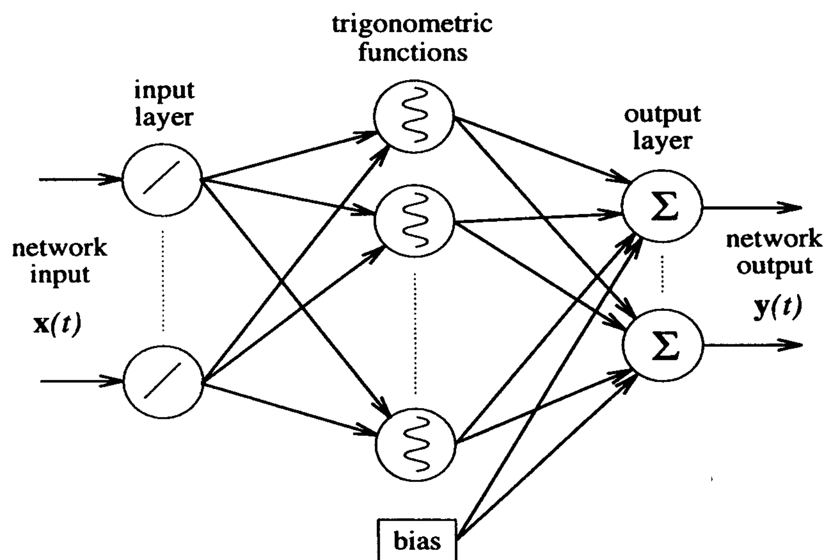

Figure 2.13 A functional link network which has trigonometric terms in the hidden layer.

These networks are universal approximation algorithms; they can approximate a continuous nonlinear function to within an arbitrary accuracy, given a sufficient number of nodes in the hidden layer [Cotter, 1990]. Like the ridge functions, polynomial and sigmoidal terms have a non-zero output over the whole input space, and these are termed *globally generalising basis functions*. A successful application of these networks requires a set of basis functions which can represent the desired function adequately over the input domain, but not over-parameterising the network. Modifying a weight or introducing (or deleting) a new term in the hidden layer affects the network's output globally and so it is not at all clear how the

structure should be chosen, or what type of relationship is stored in the network. There have been many *off-line* algorithms generated for polynomial and trigonometric term selection [Chen and Billings, 1994, Holden, 1994, Ivakhnenko, 1971], and the success of the network depends critically on the representations held in these hidden layer nodes.

2.3.4 Radial Basis Functions

Radial Basis Function (RBF) neural networks can be also be implemented within the standard three-layer network architecture, where the output nodes are simply ALCs, and the hidden layer nodes have a specific structure. RBF networks were first used for high-dimensional interpolation by the functional approximation community and their excellent numerical properties have been extensively investigated by Buhmann and Powell [1990] and Powell [1987]. They were first proposed within an ANN framework by Broomhead and Lowe [1988], and were used for data modelling and least-squares functional approximation. Since this paper was published, the technique has been widely adopted for off-line and on-line modelling and control tasks [Chen and Billings, 1994, Hunt and Sbarbaro-Hofer, 1992].

The (scalar) output y of an RBF network can be expressed as:

$$y = \sum_{i=1}^{p} w_i f_i \left(\|\mathbf{c}_i - \mathbf{x}\|_2 \right)$$

where w_i and $\mathbf{c}_i$ are the weight and centre, respectively, of the ith hidden layer node, and $\|.\|_2$ is the standard Euclidean norm. There are many different ways in which the univariate nonlinear functions, $f_i(.)$, can be selected and some of these are discussed in Section 3.3.4, but one important choice is the *localised* Gaussian function given by:

$$f_i(\|\mathbf{c}_i - \mathbf{x}\|_2) = \exp\left(-\frac{\sum_{j=1}^{n} (c_{ij} - x_j)^2}{2\sigma_i^2} \right) = \prod_{j=1}^{n} \exp\left(-\frac{(c_{ij} - x_j)^2}{2\sigma_i^2} \right)$$

If the training data are contained in a small region of the input space, the nodes in the hidden layer can be distributed within this region and only sparsely populate the remaining area. However, only a local model is formed and if the testing or operational data lie outside this small region, the performance of the network will be poor. Distributing the basis function centres evenly throughout the input space (all the theoretical results and some of the practical applications use this strategy) results in a more complex model, where the number of hidden layer nodes is exponentially dependent on the size of the input space, a property known as the *curse of dimensionality*. Irrelevant inputs cause the number of nodes in the hidden layer to increase dramatically with no corresponding increase in the model's flexibility.

An alternative RBF-type network that can reject irrelevant inputs was proposed by Hartman and Keeler [1991], where instead of the hidden layer nodes taking the *product* of the univariate Gaussian functions, the algebraic *sum* is used:

$$y = \sum_{i=1}^{p} \sum_{j=1}^{n} w_{ij} \exp\left(-\frac{(c_{ij} - x_j)^2}{2\sigma_{ij}^2}\right)$$

The output of the Gaussian bar network is *linearly* dependent on the nonlinear *univariate* Gaussian functions, and the network ignores irrelevant inputs by setting to zero the corresponding weights, w_{ij}. Typical two-dimensional Gaussian and Gaussian bar functions are shown in Figure 2.14. Comparing the outputs of the Gaussian and the Gaussian bar nodes shows that taking the product of the univariate Gaussian functions is similar to forming a multi-dimensional conjunction (AND) whereas summing the individual responses is reminiscent of the logical disjunction (OR). This is also very similar to Kavli's ASMOD algorithm (see Section 8.5) where a B-spline network is composed of the *sum* of several lower dimensional submodels. The Gaussian bar networks form *additive* models which cannot model any cross-product terms, although this restriction is why they are sometimes more successful than the standard algorithm and several "fuzzy" algorithms have been developed that try to produce parsimonious RBF networks with the smallest number of inputs [Tresp *et al.*, 1993].

2.3.5 Lattice Associative Memory Networks

Lattice-based Associative Memory Networks (AMNs) are the main focus of this book and the networks which are members of this class are described and compared in greater detail in Chapter 3. These networks can be mapped onto a three-layer structure with an ALC in the output node. The nodes in the hidden layer have a *localised* response and their output is non-zero in only a *small* part of the input space. In addition, the input space is normalised by an n-dimensional *lattice* and the basis functions are defined on this grid. Similar network inputs activate overlapping regions inside the network, and so these networks store information locally. The nodes in the hidden layer are termed ***basis functions*** and are represented by the p-dimensional vector $\mathbf{a}$. Therefore the output of the network is given by:

$$y = \sum_{i=1}^{p} a_i(\mathbf{x})\, w_i$$

as shown in Figure 2.15.

The modelling capabilities of the network depend on the size, distribution and shape of the basis functions, and the above representation is very general. The simplest lattice AMN is probably an n-dimensional *look-up table*. Associated with each cell in the lattice is a weight and a binary basis function, where the output of the basis function is 1 if the input lies inside the cell and 0 otherwise. Thus within

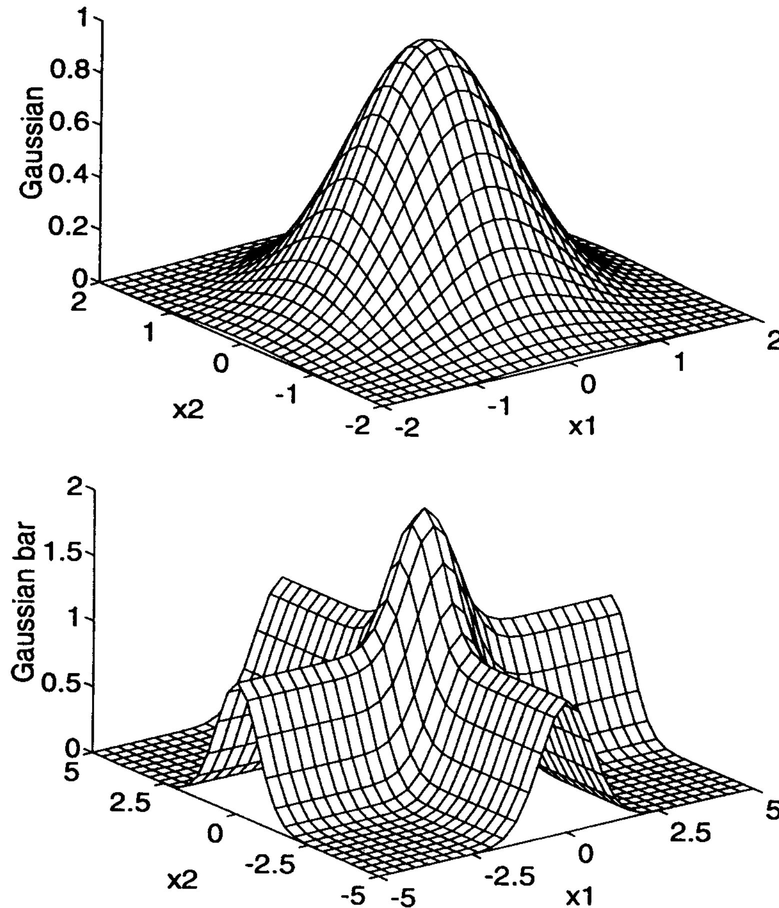

Figure 2.14 Two-dimensional Gaussian (top) and Gaussian bar (bottom) hidden layer nodes.

each lattice cell the network output is simply the corresponding weight and across the whole input domain the network's output is piecewise constant. Information about the stored functional relationship is *not* distributed to neighbouring weights and the memory requirements, p, depend exponentially on the input space dimension, but only a small fraction of the weights is involved in the network's response calculation.

Lattice AMNs partition the input space using *hard* splits (the support of each basis function is well defined), whereas Gaussian RBFs and the hierarchical networks proposed by Jacobs and Jordan [1993] provide a *soft* division of the input space. Gaussian basis functions are greater than some positive number only in a small region of the input space, hence they almost have a compact support, and the hierarchical sigmoidal decision nodes used in Jacob's tree structure (combined with a soft maximum operator) also possess this property. Truly local basis functions have the advantage that only a small region of the network contributes to the output, whereas soft split networks can potentially adapt every parameter at each time instant while still retaining a localised representation. This book concentrates on lattice AMNs, but it should be noted that these soft split networks

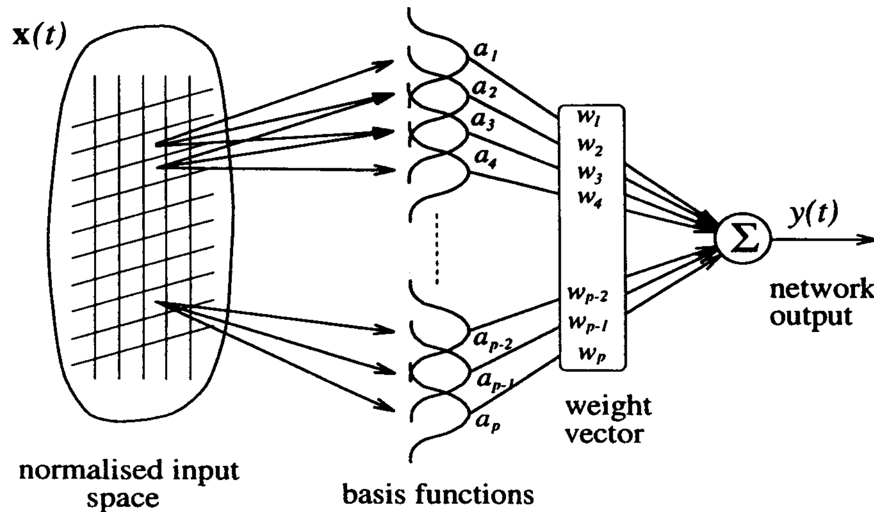

Figure 2.15 The basic architecture of a lattice associative memory network. A lattice is used to partition the input space and a set of basis functions is defined on this structure. The network's output is then formed from a linear combination of the basis functions' outputs.

provide an alternative approach which retains many of the properties of the truly local networks.

Two lattice AMNs which generalise are the Cerebellar Model Articulation Controller (CMAC) (see Chapter 6) and the B-spline network (see Chapter 8). Both networks have basis functions which are defined over more than one cell on the lattice, although in their simplest form both networks also reduce to a look-up table. The B-spline network provides *piecewise polynomial* interpolation and the local definition of the basis functions means that the basis functions can be interpreted as a set of *fuzzy* linguistic variables. The CMAC algorithm provides a coarse coding of the input lattice where the number active of basis functions does not depend on the dimension of the input space. Both networks suffer from the curse of dimensionality (this can be partially overcome using memory hashing techniques, Section 6.2.5, or by decomposing the network into submodels, Section 8.5), although very complex functional relationships can be stored, due to the local representations used in the hidden layer.

2.3.6 Model-Free Estimators

As an alternative to the model-based ANNs which have been described, a number of algorithms have been proposed which do *not* rely on any specific network structure and adapt their internal organisation in response to the training data [Atkeson, 1991, Specht, 1991]. These approaches generally store the training data (or a manageable subset) in a large memory, and use these examples to construct local models about the point of interest. These local models could simply be a low-order polynomial, or the data could be weighted using a (normalised) probability

estimate.

These approaches are similar to some k-nearest neighbour algorithms, and they share the same advantages and disadvantages of this technique. For sparsely distributed data, the techniques generally produce a smooth, global response surface which closely approximates the training samples and when the data are dense, the algorithms are capable of filtering measurement noise and producing a best estimate of the true output. However, any algorithm which is based on remembering data has a high computational burden, because each training sample contributes to the network output. All of these operations can be performed in parallel, but for most practical applications the amount of training data far exceeds the number of available processors. Similarly, these algorithms *learn* by simply remembering training samples; there are no data-forgetting algorithms which could be used to model (or control) a time-varying plant.

2.3.7 Network Generalisation

All the feedforward model-based ANNs considered in this section can be represented using a three-layer network structure which has an ALC node in the output layer. The type of model and the associated generalisation characteristics are therefore *strongly* dependent on the type of nonlinearity incorporated in the hidden layer nodes. Many different sorts of nonlinearities have been described: bounded ridge functions (MLPs), trigonometric and polynomial functions (FLN), Gaussian and Gaussian bar functions (RBF) and piecewise polynomial functions (AMN). It is *not* possible to say that one type of model is always better than another; all that can be achieved is to list their desirable (and undesirable) properties.

The MLP has proved very useful for learning high-dimensional, redundant mappings, as the ridge functions partition the input space using hyperplanes. Using more than one hidden layer allows a more complex partitioning strategy to be developed, but the associated learning problem is generally much harder. The support of the hidden layer nodes (area of the input space for which the node has a non-zero output) is *global*, so adjusting any weight will affect the output of the network for *every* input. The training set should therefore contain a relatively complete coverage of the input space [Barnard and Wessels, 1992], otherwise the learnt network structure will not be of the correct form. In order to make the MLPs more *transparent* and understand the knowledge encoded in the weights, the approximation of these networks using finite polynomial and Volterra series models has been proposed. Any finite model of this form loses some information encoded in the network, where generally the higher order knowledge is lost, although it is precisely these terms which appear to give the MLP its advantages over polynomial/Volterra models. Hence the usefulness of such tests is debatable.

The success of polynomial and trigonometric functional modelling depends on the successful identification of the relevant nonlinear nodes. If any important terms are omitted, the global support of the nodes results in the model producing

a globally biased solution, whereas if too many terms are included, the network tends to overfit the data, and the network output can be extremely oscillatory. However, these types of nodes have a strong theoretical background and many techniques have been developed for choosing a near optimal set of terms.

RBF networks have proved very useful for modelling highly nonlinear data, although their performance depends on the type of nonlinearity used in the hidden layer nodes as well as the distribution of their centres. It has been found that the approximation capabilities of functions with a global support appear to be slightly better than functions with a local support, although more advanced learning algorithms generally have to be used. The Gaussian bar networks, which are not strictly RBF networks, appear to be very useful when dealing with redundant data. Their support is not truly local (but more local than a sigmoid) and not truly global (but more flexible than a standard Gaussian function). The basis functions are generated by using the *addition* rather than the *product* operator to combine the univariate Gaussian functions, but only additive models are formed.

Lattice AMNs have truly local basis functions (hidden layer nodes) which provide an alternative to many of the ANN architectures currently being considered. The local support of the basis functions *forces* the network to generalise in a prespecified manner, and therefore *a priori* knowledge about the desired function can be incorporated into the network's structure, although if the structure is inappropriate, the model will be biased (see Section 2.5.1). The networks are analogous to the piecewise polynomial data-fitting algorithms, in that the piecewise nature of the network prevents unwanted oscillatory behaviour and overfitting. However, the distribution of the basis functions on a lattice means that these networks suffer from the *curse of dimensionality*, which sometimes results in the networks being too flexible (large variance), especially in high-dimensional spaces.

Model-Based verses Model-Free Networks

Model-free algorithms provide an alternative to ANNs, and their modelling (and generalisation) performance demands that further research be performed into the relative merits of these techniques. They generate smooth, global models when the data are sparsely distributed, compared with the lattice AMNs which would produce only local approximations. Also when the training data are dense, very complex models can be produced.

Whether to use a model-based or a model-free algorithm depends on the problem and the available resources. Choosing model-based algorithms allows learning laws to be formulated for the unknown parameters and the network's response calculations do *not* depend on the number of training samples. However, the generalisation depends on the type of nonlinear hidden layer nodes and sometimes this relationship is not explicit, the exception being the lattice AMNs. The response time of a model-free algorithm generally grows linearly with the number of training samples, although no *a priori* knowledge is assumed about the form of the model

and it appears to generalise sensibly.

2.4 TRAINING ALGORITHMS

The ability to *learn* complex mappings from a data set is *the* cornerstone of the recent revival of interest in ANNs. The inability of the Perceptron learning algorithm to be extended to multi-layer networks [Minsky and Papert, 1969] meant that a network could not learn to reproduce the basic XOR logical function, and so these algorithms were not considered suitable for application to more complex real-world problems. However, in the mid-eighties, when a gradient descent algorithm called Back Propagation (BP) was reinvented which enabled MLPs to learn arbitrary functional mappings, it stimulated considerable interest in learning systems.

BP is a *supervised* learning rule; the desired network output is given to the learning algorithm and the difference between this value and the actual output is used to guide the adaptive mechanisms. Many other learning algorithms have also been developed, such as *unsupervised* and *reinforcement* rules, and there are many different algorithms within these broad classifications. This section does not aim to give a complete survey of this field, but it provides a broad overview while concentrating on the rules which are applicable in the neuromodelling and control field.

2.4.1 Unsupervised Learning

Widrow termed unsupervised learning as *open-loop* adaptation, because it does not use any performance *feedback* information to update the network's parameters. This type of learning can be used in a variety of ways:

- **group** the input data into clusters, which can then be labelled in a supervised mode;
- **quantise** the continuous input space in an "optimal" manner;
- **represent** the input data in a lower dimensional space; and
- **extract** a set of features which represents the input signal.

This classification reflects the *usage* of unsupervised learning laws, as many of the above tasks are very similar. For instance, vector quantisation could be considered a subclass of feature extraction, although the latter is generally used for pattern classification, whereas the former can be used to prepare the input data for modelling algorithms.

Unsupervised learning algorithms can be used in a wide variety of networks: training the centres of an RBF network [Chen *et al.*, 1992], providing an "optimal" lattice for the AMNs, although many learning rules are closely tied to particular

system architectures [Grossberg, 1988]. They have also been used for many different applications: speech recognition [Kohonen 1988, 1990], robotics modelling [Ritter *et al.*, 1989], image compression [Nasrabadi and Feng, 1988], etc. The basic unsupervised learning algorithm and the two most important unsupervised network architectures: Kohonen's Self-Organising Map and Grossberg's ART networks are described in this section.

Competitive Learning

Competitive ANNs distribute their nodes across the input space, so that they learn to represent the statistics of the input data which is presented to the network, as illustrated in Figure 2.16. The recall phase of such a network simply involves

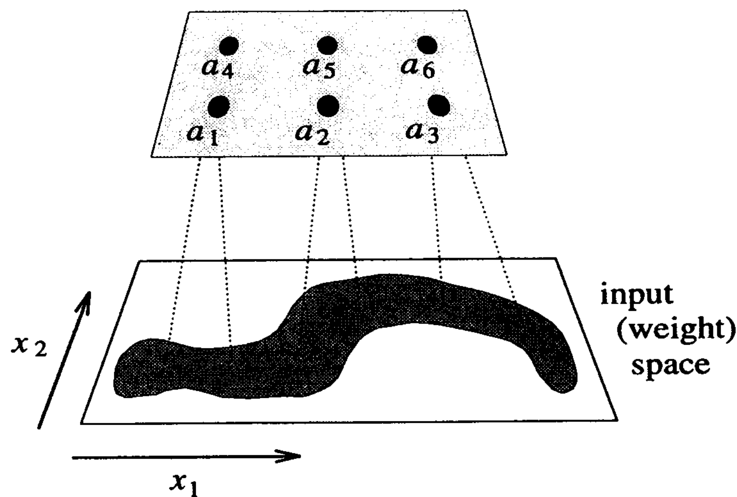

Figure 2.16 An unsupervised learning network showing how the nodes become sensitised to different parts of the input space. The input is assumed to have a uniform probability density function (shown by the shaded area) and the weight vectors (dotted lines) are distributed evenly in the relevant parts of the input space.

finding the node which best matches a particular input pattern and exporting either the index of this node or the associated classification label.

A network which is trained using the unsupervised competitive learning algorithm is composed of a group of nodes $\{a_i\}_{i=1}^{p}$, and associated with each node is a *normalised* weight vector $\mathbf{w}_i$ (each element should also be normalised so that it assumes equal importance). A normalised input, $\mathbf{x}$, is presented to the network and a match is then calculated between the input and the weight vector corresponding to each node, whose activation value is given by:

$$a_i = \sum_{j=1}^{n} w_{ij} x_j = \mathbf{x}^T \mathbf{w}_i \qquad (2.11)$$

The network then determines the node a_k such that $a_i < a_k \; \forall i \neq k$ and the output is either this index or the classification label stored at the kth node. Selecting the node with the maximal output (closest match) can also be implemented as a neural network, as described by Lippmann [1987].

The weight vector associated with the best match node is updated according to the rule:

$$\mathbf{w}_k(t) = \frac{\mathbf{w}_k(t-1) + \delta\left(\mathbf{x}(t) - \mathbf{w}_k(t-1)\right)}{\|\mathbf{w}_k(t-1) + \delta\left(\mathbf{x}(t) - \mathbf{w}_k(t-1)\right)\|} \tag{2.12}$$

and every other weight vector retains its current value.

If the input data are not normalised, the match is implemented using the Euclidean distance measure and the kth node is selected such that $\|\mathbf{x} - \mathbf{w}_k\|_2 < \|\mathbf{x} - \mathbf{w}_i\|_2 \; \forall \, i \neq k$. The learning rule is then modified to:

$$\mathbf{w}_k(t) = \mathbf{w}_k(t-1) + \delta\left(\mathbf{x}(t) - \mathbf{w}_k(t-1)\right)$$

These learning rules $(k = 1, 2, \ldots, p)$ can easily be shown to form a set of p MSE gradient descent training laws, and the weight vectors converge in the limit to their optimal values with respect to the network input MSE.

The networks and the learning rules can be used as simple classification systems, where a label is associated with each node, or for vector quantisation where a value (weight) is stored at each node and the network output is simply this weight. Generalised competitive networks allow more than one node to be active at any one time (similar to k-nearest neighbour networks) and interpolate locally the information stored at each node. Initial learning is also generally faster for these networks, as more than one weight vector is updated for each input.

The Self-Organising Map

Kohonen's Self-Organising Map (KSOM) [Kohonen, 1990] is a competitive network in which the nodes are *ordered*, and its aim is to produce a low-dimensional, topology conserving representation of the input space. The nodes are regularly placed in an m-dimensional space, (generally $m < n$) and the learning algorithm attempts to formulate a mapping such that, if two inputs $\mathbf{x}(1)$ and $\mathbf{x}(2)$ are close in the input space, the two activated nodes a_i and a_k are also close in the m-dimensional user-defined space. Generally the nodes are placed on an m-dimensional *lattice* and an appropriate *kernel* function is associated with each node which tends to zero as the input moves away from the activated node. At time t, the weight vectors are updated according to the following rule:

$$\mathbf{w}_i(t) = \mathbf{w}_i(t-1) + \delta(t)h(i,k)\left(\mathbf{x}(t) - \mathbf{w}_i(t-1)\right) \tag{2.13}$$

where the kth unit is activated at time t, and $h(i,k)$ is the kernel function and $\delta(t)$ is the time-varying learning rate. If $h(i,k) = 1$ when $i = k$ and 0, otherwise

the standard competitive learning algorithm is obtained. However if $h(i,k) = 1$ when $i = k$ and it tends to 0 in the locality of k, the nodes surrounding a_k receive similar training information and the weight vectors become sensitised to similar input regions. The mapping then retains the topological features of the original input space. Many different kernel functions can be used in the above algorithm: Mexican hat functions, Gaussian functions, etc., and the algorithm appears to be reasonably robust with respect to different selections.

For modelling and control applications, this algorithm has the potential to map a high-dimensional input space to a low-dimensional representation which preserves the topological ordering of the original inputs. In high-dimensional space, most of the relevant input data are contained in a much smaller subspace, and the KSOM has the ability to extract this information automatically in a computationally efficient manner [Walter and Schulten, 1993].

Adaptive Resonance Theory

Since the late sixties, Grossberg and his research group have been investigating and developing (amongst others) biologically plausible neural pattern classification architectures and the associated learning rules. The basic design question which this research addresses is development of systems that can be "designed to remain plastic, or adaptive, in response to significant events and yet remain stable in response to irrelevant events" [Carpenter and Grossberg, 1988]. This was termed the *stability-plasticity* dilemma and is a *fundamental* problem for any on-line adaptive algorithm. Grossberg proposed an Adaptive Resonance Theory (ART) for neural networks, which is designed to overcome this problem and also possesses three important properties: the input activity is normalised (similar to the competitive networks), contrast enhancement of input patterns and distinction between Short-Term Memory (STM) and Long-Term Memory (LTM) [Grossberg, 1988]. The three ART architectures proposed to date can deal with binary input signals (ART1), real-valued input signals (ART2) [Carpenter *et al.*, 1991a] and ART3 incorporates a hierarchical structure [Carpenter and Grossberg, 1990]. Other proposed systems have included a fuzzy ART [Carpenter *et al.*, 1991b], and vector associative maps for unsupervised learning and control of movement trajectories [Grossberg *et al.*, 1993].

The basic ART1 architecture is shown in Figure 2.17 with the bottom-up and top-down weight arrays ($\mathbf{W}^b$ and $\mathbf{W}^t$, respectively) playing a major part in preventing learned memories from being overwritten by new information. A bottom-up weight vector encodes the competitive element of the ART architectures, and the node with the best match is selected. The top-down weight vector associated with this node is then compared with the input pattern and if it is a good match, this categorisation is accepted, if not the node is disabled and a new best match is calculated. Once the input has been either categorised or a new node has been allocated to store this input, the LTM weights are changed.

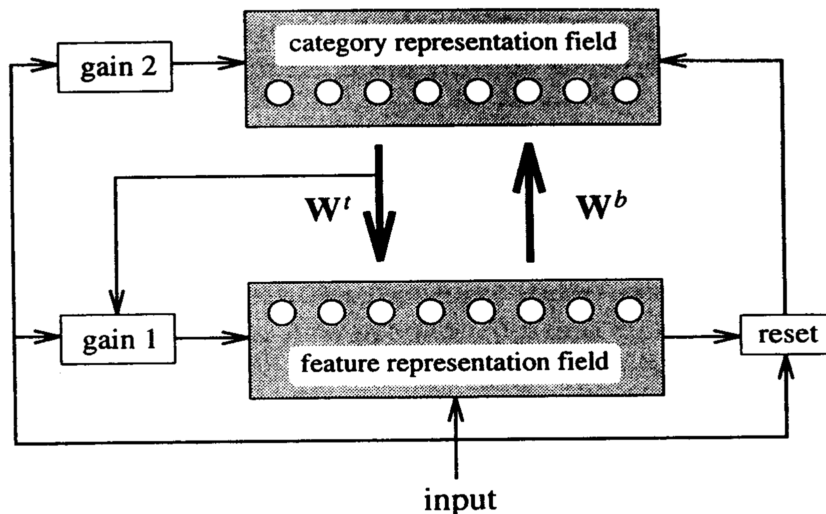

Figure 2.17 The basic ART1 architecture.

The ideas encoded in the ART architectures are *directly* relevant to on-line learning modelling and control algorithms, and any proposed system should have the ability to store new information without affecting unrelated stored data. Many learning systems do *not* possess this property, although the lattice AMNs considered in this book have this feature, as new information is stored *locally*.

2.4.2 Supervised Learning

Supervised learning rules differ from unsupervised training algorithms because the network's desired response (output) needs to be presented to the network for each input. Denoting the desired network output at time t by $\widehat{y}(t)$, the *instantaneous* performance of the network can be inferred from:

$$\epsilon_y(t) = \widehat{y}(t) - y(t) \tag{2.14}$$

and these output errors are used to update the network's weights.

Parameter convergence is the aim of this optimisation strategy, as the network is only able to *generalise* correctly if the parameters are close to their optimal values. Generalisation is *essential* if the network is trained off-line from a finite training set, and this illustrates the problem of determining *when* to stop training. If this decision is based on the size of the output error only, a low value does not always imply that the weights are correctly initialised, and this problem is discussed further in Sections 4.6 and 8.3.2.

Supervised learning is probably the most widely used training mechanism and gradient descent adaptation is probably the best known supervised learning rule. This is due to a variety of reasons:

- Widrow's original LMS learning rules are instantaneous gradient descent training algorithms.

- A large number of theoretical results have been derived, which can establish parameter convergence and estimate the rate of convergence for certain network structures.
- The resulting learning algorithms have low memory requirements and a low computational cost.

Therefore this section concentrates on gradient descent learning algorithms (batch and instantaneous), although other training rules are discussed.

Batch Learning

Supervised learning algorithms can be divided into two distinct approaches: ***batch*** and *instantaneous* training rules. Instantaneous training rules use only the information provided by a single training example $\{\mathbf{x}(t), \widehat{y}(t)\}$, when the weight vector is updated, whereas batch learning laws generally use *all* the training data to adapt the weights. Under certain conditions, the instantaneous training rules are approximations of their batch counterparts, and so the latter are described first.

For a training set given by $\{\mathbf{x}(t), \widehat{y}(t)\}_{t=1}^{L}$, a cost function which measures the current performance of the network needs to be specified. In most applications this is the MSE defined by:

$$J = E\left(\epsilon_y^2(t)\right) = \frac{1}{L}\sum_{t=1}^{L}\epsilon_y^2(t) \tag{2.15}$$

Other cost functions may improve the network's performance, although using the MSE produces computationally efficient learning algorithms and good final models.

Gradient descent learning algorithms adapt the weight vector in the direction of the *negative* gradient of the performance function:

$$\Delta\mathbf{w} = -\frac{\delta}{2}\frac{\partial J}{\partial \mathbf{w}} \tag{2.16}$$

where δ is the *learning rate* which determines the stability and the rate of convergence of the learning algorithm and $\Delta\mathbf{w}$ is the weight vector update. Applying the chain rule to the MSE cost function gives:

$$\Delta\mathbf{w} = -\frac{\delta}{2}\frac{\partial y}{\partial \mathbf{w}}\frac{\partial J}{\partial y} = \delta E\left(\frac{\partial y(t)}{\partial \mathbf{w}}\epsilon_y(t)\right) \tag{2.17}$$

Therefore the gradient update contains information about the network Jacobian $(\partial y/\partial \mathbf{w})$, and the performance of the network $\partial J/\partial y$. For the MSE cost function, the latter quantity is simply the output error, and the *efficiency* of this algorithm will depend on the structure of the network (information held in the Jacobian). If the network is linearly dependent on the weight vector $(y(t) = \mathbf{a}^T(t)\,\mathbf{w})$, the above expression simplifies to:

$$\Delta \mathbf{w} = \delta E\left(\epsilon_y(t)\mathbf{a}(t)\right)$$

and the computational simplicity of this rule is obvious.

For linear networks, it is shown in Chapter 4 that the weights trained using this learning algorithm converge to their optimal values, and the rate of convergence can be estimated. The gradient descent learning rules can also be extended for nonlinear optimisation (to train MLPs for instance, see Section 4.5), although the theoretical convergence results do not generally apply. This is illustrated in Figure 2.18, where it is shown how gradient descent rules can become trapped by local minima and plateau areas. Even the rate of convergence of linear networks depends on the *condition* of the model and slow convergence occurs when the model is poorly conditioned.

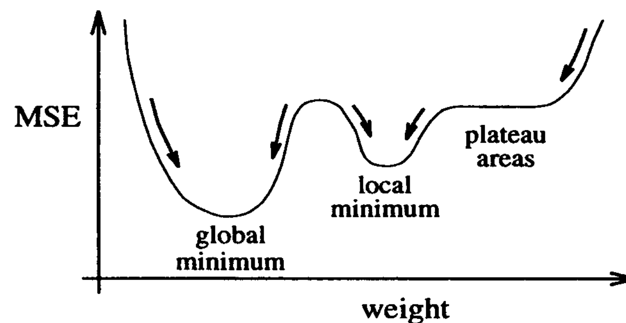

Figure 2.18 The nonlinear relationship between a weight and the mean square output error. Gradient descent training algorithms can become trapped by local minima and plateau areas.

This discussion has considered *first-order* gradient descent rules which use only the first-derivative information. Second and higher-order training algorithms can be derived which make use of second (and higher) -order derivatives [Widrow and Stearns, 1985]. Probably the best known of these is Newton's method which uses second-order curvature information and this can be expressed as:

$$\Delta \mathbf{w} = -\mathbf{D}\frac{\partial J}{\partial \mathbf{w}}$$

where $\mathbf{D}$ is (an approximation to) the inverted second-derivative matrix (the Hessian) [Gill *et al.*, 1981, Mills, 1992]. Equation 2.16 is therefore an approximation to this more complex learning rule, and they are equivalent when the cost function has constant second derivatives and there is no cross-coupling with respect to each weight, i.e. $\mathbf{D} = \text{diag}\{\delta, \delta, \ldots, \delta\}$.

Instantaneous Learning

Instantaneous learning rules generally use only a single piece of training information when the weights are updated. This is closer to human learning processes, which do not use all the available information. However, this fact should not be used to

justify the use of these adaptation rules in learning control applications, but should inspire their development and their performance should be *rigorously* assessed. Batch learning rules have the disadvantage that all the training information is required before the weights are updated and, although recursive algorithms are available, the large and complex structure of the networks limits the feasibility of these techniques.

Instantaneous learning algorithms generally attempt to minimise an instantaneous measure of the network's performance, and for the MSE cost function this is given by:

$$J(t) = \epsilon_y^2(t) \tag{2.18}$$

Using a gradient descent rule, the weight update is:

$$\Delta \mathbf{w} = -\frac{\delta}{2}\frac{\partial J(t)}{\partial \mathbf{w}} = \delta \epsilon_y(t) \frac{\partial y(t)}{\partial \mathbf{w}} \tag{2.19}$$

which is simply an (unbiased) estimate of the derivative of the true cost function. When the network is linear this reduces to:

$$\Delta \mathbf{w} = \delta \epsilon_y(t) \mathbf{a}(t)$$

This rule can be interpreted as updating the weights in proportion to the amount that they contributed to the output. If the vector $\mathbf{a}$ is *sparse* (many zero elements) and the non-zero elements are approximately of the same magnitude, the learning algorithm works well, as is shown in the following example and in Chapter 5.

In Section 5.3.1, it is shown that if no mismatch (modelling error or measurement noise) exists, then a weight vector trained using the above rule converges to a value that can exactly reproduce the training data. However, when modelling error exists, the noise associated with the gradient estimate means that the weight vector no longer converges to an optimal value, but lies in a domain which surrounds this quantity. The size and shape of this *minimal capture zone* (see Section 5.4.3) depends on the network's structure and the distribution of the training data, and this can be regarded as part of the cost of using an instantaneous estimate rather than the true gradient. If the learning rate is allowed to decay to zero rather than remaining constant, the gradient noise is filtered out and the weights converge to their optimal values [Luo, 1991].

These learning rules can also be used for networks which depend nonlinearly on their weights, and the original BP algorithm for training MLPs was an instantaneous learning rule [Rumelhart and McClelland, 1986]. Also higher-order learning rules, which use the past L training samples, can be derived and these generally reduce the size of the minimal capture zones and increase the rate of convergence, at a cost of increasing the computational complexity.

Example: Look-Up Table

A look-up table has an extremely simple structure, although it can provide considerable insight into the suitability of training a network, using an instantaneous learning law. It has an n-dimensional lattice defined on the input space and the output of the network is simply the ith weight, if the input lies in the ith lattice cell. The transformed vector **a** has $(p-1)$ zero elements and the ith element is unity, so it is extremely sparse.

The weight vector is updated using an instantaneous *stochastic* learning rule:

$$\Delta w_j(t-1) = \delta_j(t)\, \epsilon_y(t)\, a_j(t)$$

where $\delta_j(t)$ is the learning rate associated with the jth node at time t. Only the ith weight is updated at time t, so the learning algorithm is computationally efficient, and the time-varying learning rate is given by:

$$\delta_i(t) = \frac{1}{\sum_{k=1}^{t} a_i(k)}$$

therefore the *a posteriori* weight value at time t is:

$$\begin{aligned} w_i(t) &= w_i(t-1) + \frac{1}{\sum_{k=1}^{t} a_i(k)}(\widehat{y}(t) - w_i(t-1))a_i(t) \\ &= \frac{\widehat{y}(t) + w_i(t-1)\sum_{k=1}^{t-1} a_i(k)}{\sum_{k=1}^{t} a_i(k)} \\ &= \frac{\sum_{k=1}^{t} \widehat{y}(k) a_i(k)}{\sum_{k=1}^{t} a_i(k)} \end{aligned}$$

This last quantity is the optimal estimate of the weight (at time t), and so the instantaneous and batch learning laws are equivalent for a look-up table.

The instantaneous learning algorithm works well because the transformed input vector **a** is sparse and all the non-zero elements are equal in value. If a network is to be successfully trained using instantaneous learning rules, it should partially satisfy these two conditions, which is true for the lattice AMNs analysed in this book.

Alternative Strategies

This section has concentrated mainly on gradient descent learning rules because of their popularity and the theoretical results which can be derived. However, they are not suitable for every optimisation problem and for every network structure. Two other supervised learning algorithms which have been used for training ANNs are *stochastic* and *genetic* learning rules.

Stochastic training algorithms introduce a random element into the search for the optimal weight vector. They can be used when the network's output is *not*

a continuously differentiable function with respect to weight vector. In the most general form, a random search can be performed in weight space and the weight vector which generates the lowest cost is selected, or else a stochastic element can be introduced into other learning rules (such as gradient descent) in order to reduce the *chance of becoming stuck in local minima. This technique should only be used if the* cost function is highly complex as they use little or no information about its shape, and gradient descent rules generally perform better [Monzingo and Miller, 1980].

Genetic algorithms have gained in popularity over the past few years, although the seminal text on the subject was written nearly twenty years ago [Holland, 1975]. The weights are represented using binary strings and the strings are concatenated to form one large string. A *population* is then created from several large, different strings and these are combined using various genetically inspired techniques such as cross-over (cutting strings and swapping the respective components) and mutation (randomly changing bits in the string). The fitness of the each new string is evaluated and a new population is created which consists of some of the new and some of the old strings, and the fittest strings have a highest probability of being members of the new population. The technique combines elements of directed (using cross-over) and random (mutation) search algorithms and has been shown to perform well in a wide variety of off-line tasks [McGregor *et al.*, 1992, Renders and Hanus, 1992], which involve highly complex optimisation calculations.

2.4.3 Reinforcement Learning

Reinforcement learning algorithms use a reduced form of performance feedback information in their updating rules. This performance information is generally binary and denotes whether the sequence of control actions has been successful. The seminal paper on this subject was written by Barto *et al.* [1983] where an inverted pendulum was trained using a reinforcement learning algorithm. The learning ANNs which implement the ASE and ACE blocks, described in Section 2.2.4, are generally similar to look-up tables, although recently more complex input space quantisation strategies have been proposed [Zhang and Grant, 1992], and more flexible networks have been applied [Anderson, 1989]. The reduced performance feedback information means that the networks which initially learn quickly perform well and this was confirmed when the CMAC and fuzzy networks were applied to the pole-balancing reinforcement learning task [Berenji and Khedkar, 1992, Lin and Kim, 1991].

These reinforcement learning rules were motivated by the adaptive automata algorithms [Barto and Anandan, 1985, Narendra and Thathachar, 1974], where they are used to update the probability vectors associated with each possible input state. A good description of the weight update equations is provided by Barto *et al.* [1983] and Millington and Baker [1990]. The ACE reinforcement weights are updated in proportion to the weighted sum of the local reinforcement signals and the ASE weights are updated in proportion to the internal reinforcement signal at time t, multiplied by a term which measures the eligibility of that particular

state. The continuous reward/punish signal from the ACE means that the ASE is able to learn, even when the system does not fail, and the performance of this algorithm over a system which only uses the external reinforcement signal has been demonstrated many times.

2.5 VALIDATION OF A NEURAL MODEL

The most important part of any learning systems design procedure is the verification and validation phase. During these tests, the designer should be able to assess how well the network has learnt the training data and how successfully it is able to generalise (interpolate and extrapolate) to unforeseen cases. Most learning algorithms can successfully learn a set of training examples given a sufficiently flexible model structure or an appropriate learning algorithm, although the question of whether they possess the ability to generalise *correctly* (or sufficiently accurately) is still unresolved [Barnard and Wessels, 1992]. In Lau and Widrow [1990], it is said that "it is necessary to develop quantitative techniques to evaluate neural network's performance with real-world data ... Rigorous mathematical foundations must be developed to determine the characteristics of the training set and the network's ability to generalise from the training data". For safety and certification reasons the performance of the trained network must be completely understood; it cannot be simply regarded as a black box about which nothing can be proven. It has been argued that most of the ANNs are model-based algorithms and so tests are required to determine if the structure is sufficiently flexible and to establish if the network's input representation contains enough information. From a learning viewpoint, it is desirable to have the *smallest* acceptable network and input vector, as this forces the network to generalise sensibly.

There are many different ways in which the performance of a trained network can be assessed, the simplest (and probably the most biased and abused [Weiss and Kulikowski, 1991]) method is by simply assessing the network's performance in reproducing the training data. A better approach is to split the available data into a training and a test set, and to use the testing data to evaluate the final model. There are many different ways in which the available data can be divided, and some of these are reviewed in Section 2.5.2. This test only provides very general performance information; if the model is poor, it does not give any reasons about why this occurs. During the eighties, a number of correlation and statistical based tests were devised for validating the NARMAX models developed by Billings and his collegues [1983, 1986, 1989, 1992]. These tests are equally applicable to neurofuzzy modelling and control algorithms, and the model validation tests are reviewed in Section 2.5.3. The topic of network *transparency* is then examined, as this is one of the most desirable features of the networks discussed in this book. Network transparency refers to the manner in which information is stored, and a network is said to be transparent to the designer if the relationship between a

weight and the network surface is *easily* understood, otherwise the network is said to be *opaque*. Networks with this property are described in Section 2.5.4, and the validity of assessing opaque networks by mapping them onto a transparent model is discussed.

All of these evaluation methods provide information about the generalisation ability of the trained model, and it is anticipated that additional performance tests will evolve in the future.

2.5.1 Bias Variance Dilemma

Artificial neural networks have been called *model-free* estimators, as they are flexible enough to approximate *any* smooth nonlinear function to a prespecified degree of accuracy, given sufficient resources. This is a *necessary* theoretical result, which shows that there may be a large number of modelling problems which neural algorithms could be applied to, although it does not guarantee success. Irrespective of the learning algorithm used to train the parameters, the model's structure should be appropriate. The model should be flexible enough to learn the desired mapping described by the training data, but it should not be *over-parameterised*, as this causes the model to fit the noise which is inherent in most data sets. The problem of estimating how flexible a model should be (how many nodes in each layer, number of layers, order of the splines, etc.) is well developed in the statistical modelling community where it is termed the bias/variance dilemma [Geman *et al.*, 1992]. The MSE performance measure for a particular input can be decomposed into two components which reflect the *bias* of the network error (where the average is taken over all possible training sets), and the *variance* of the estimates of the trained networks. Following the notation developed by Geman *et al.* [1992], let D denote a training set, $y(\mathbf{x}, D)$ be the output of a network trained using the data contained in D and $E_D(.)$ denote the expectation operator taken over all possible training sets, then the output error is given by:

$$\epsilon_y(\mathbf{x}) = \widehat{y}(\mathbf{x}) - y(\mathbf{x}, D)$$

The suitability or effectiveness of this network as a predictor of $\widehat{y}(\mathbf{x})$ can be measured by calculating the MSE for all possible training sets D, giving:

$$\begin{aligned} E_D\left(\epsilon_y^2(\mathbf{x})\right) &= E_D\left((\widehat{y}(\mathbf{x}) - y(\mathbf{x}, D))^2\right) \\ &= (\widehat{y}(\mathbf{x}) - E_D(y(\mathbf{x}, D)))^2 + E_D\left((\widehat{y}(\mathbf{x}) - y(\mathbf{x}, D))^2\right) \end{aligned} \tag{2.20}$$

The first term on the right-hand side is called the ***bias***, as it measures the *average* modelling error, whereas the second estimates the ***variance*** of the network approximations. Even if an "average" network is able to interpolate the data and make the bias zero, a high variance still causes poor performance. A large variance occurs when the performance of the network is very sensitive to the training data (if the network is too flexible) which results in a poor MSE performance.

In general, a network should be flexible enough to ensure that the modelling error (bias) is small, although the model should not be over-parameterised, as this causes its performance to strongly depend on a *particular* training set (high variance).

2.5.2 Output Error Tests

Many of the network training rules are based on minimising an MSE cost function and therefore it is natural that the same test should be applied first when the performance of the network is evaluated. The Root Mean Square (RMS) output error is generally a *negative* test; it only gives information about the training being inadequate (a high RMS), and little can be deduced from a low RMS value. This can be easily illustrated by considering a nearest neighbour data storage algorithm which simply remembers the training data. After one training cycle, the network's recall is perfect and the RMS is zero, but this testing procedure gives no information about its ability to generalise. Alternatively, the reduction in the RMS after one learning cycle for an MLP trained using a gradient descent algorithm is small and the only information provided is that the learning procedure should not be stopped.

It is important to obtain a good estimate of the true performance of the ANN, as this could be used to determine if the structure is flexible enough for a particular data set. If the modelling capabilities of the network were increased and this resulted in a lower estimate of the true RMS value, it would indicate that the original network structure was insufficient. Similarly, if the true RMS value was higher for a more complex model, this would indicate that the ANN was starting to model the disturbances present in the data and a simpler network structure would be preferred, as this forces the network to generalise. The RMS test provides no estimate of the variance of the network, and this is illustrated in Figure 2.19.

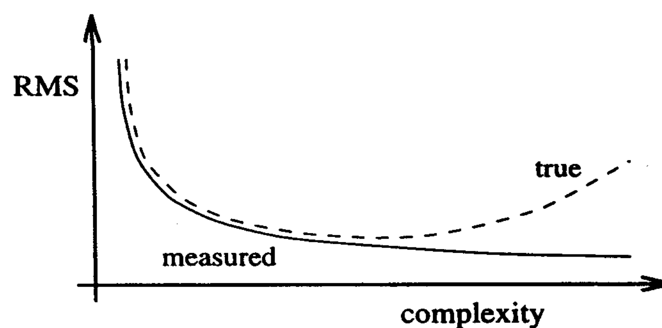

Figure 2.19 An illustration of the trade-off between network complexity and the true (dashed line) and measured (solid line) RMS.

Despite the fact that this performance measure can easily be abused, several statistical and information theory-based validation tests, see Section 8.5.4, have been developed which attempt to reproduce the true RMS, by combining the measured RMS with terms that estimate the network's complexity. These measures

weigh the RMS against the number of parameters in the network and the amount of training data, producing parsimonious networks that are able to model the data adequately.

Test Set Construction

Any data set presented to a learning algorithm must be split into a part which is used to train the network, and another used for evaluating its performance by testing its ability to generalise correctly. For off-line design, where a fixed training set is presented to the network, there is generally a problem due to lack of data and so an obvious problem is to determine how much should be used for training and how much should be used for testing. One heuristic which works well in practice is to use two-thirds for training and one-third for testing. This approach is reasonable because it is not possible for a learning algorithm to perform well if there are insufficient training data, but the performance cannot be assessed accurately if the test set is too small. Random subsampling can be used to obtain a better estimate of the true RMS, and this proceeds by randomly generating different test and training sets, training a network on each of these different test sets and averaging the RMS values after each training session. These measures produce estimates for both the network's bias and its variance which can be then used to assess the model's true performance. A special case of this technique is the *leave-one-out* strategy where only one piece of data is used for testing and the remainder is used for training. The learning and testing procedures are carried out enough times so that each data pair is used as the test case just once, and the RMS value (total variance) can be calculated by averaging over each RMS estimate [Weiss and Kulikowski, 1991].

2.5.3 Correlation and Statistical Tests

The validation of a trained ANN using a test set is the first of many testing procedures which should be applied to the network, and for the nonlinear NARMAX models (see Section 2.2.1) many other evaluation algorithms have been developed. These tests can highlight deficiencies occurring in the input data and in the network's structure, so they can be used to assess the order of the nonlinear models, measurement errors that may have occurred when the data were collected and to produce parsimonious models which generally perform well.

Correlation Tests

Correlation-based evaluation tests for NARMAX models were developed after it became apparent that the linear covariance tests [Ljung and Soderstrom, 1983] were

not sufficient when they are applied to nonlinear systems [Billings and Voon, 1986]. A further set of correlation-based tests was developed and any well-trained ANN model should satisfy the following five conditions:

$$\begin{aligned}
\phi_{\epsilon\epsilon}(\tau) &= E\left(\epsilon(t-\tau)\epsilon(t)\right) = \delta(\tau) \\
\phi_{u\epsilon}(\tau) &= E\left(u(t-\tau)\epsilon(t)\right) = 0 \qquad \forall\tau \\
\phi_{u^{2\prime}\epsilon}(\tau) &= E\left(\left(u^2(t-\tau) - E\left(u(t)\right)^2\right)\epsilon(t)\right) = 0 \qquad \forall\tau \qquad (2.21) \\
\phi_{u^{2\prime}\epsilon^2}(\tau) &= E\left(\left(u^2(t-\tau) - E\left(u(t)\right)^2\right)\epsilon^2(t)\right) = 0 \qquad \forall\tau \qquad (2.22) \\
\phi_{\epsilon(\epsilon u)}(\tau) &= E\left(\epsilon(t)\epsilon(t-1-\tau)u(t-1-\tau)\right) = 0 \qquad \tau \geq 0 \qquad (2.23)
\end{aligned}$$

where $\epsilon(t) = \epsilon_y(t)$ is the current network output error. To generate meaningful results, *normalised* correlation functions are generally calculated and these are defined by:

$$\widehat{\phi}_{\psi_1\psi_2}(\tau) = \frac{\sum_{t=1}^{L-\tau} \psi_i(t)\psi_2(t+\tau)}{\left(\sum_{t=1}^{L} \psi_1^2(t) \sum_{t=1}^{L} \psi_2^2(t)\right)^{0.5}} \qquad (2.24)$$

The normalised correlation functions lie in the interval $[-1, 1]$, and a 95% confidence band ($= \pm 1.96/\sqrt{L}$) indicates if the calculated correlations are significant.

It is not known whether or not these tests are sufficient, but they form a powerful set of evaluation methods for process modelling, and have proved very useful for a wide variety of different ANNs. They provide information about the structure of the network, as well as determining if the input vector is sufficiently rich.

Chi-Squared Tests

Another evaluation algorithm which has been used for nonlinear model validation is the chi-squared statistical test [Bohlin, 1978, Leontaritis and, Billings, 1987], which indicates whether or not a model is biased. This performance measure is calculated by defining an η-dimensional vector $\Omega(t)$ as:

$$\Omega(t) = [\omega(t), \omega(t-1), \ldots, \omega(t-\eta+1)]^T$$

where $\omega(t)$ is a function of the previous system inputs, outputs and prediction errors. The chi-squared statistic is then given by:

$$\zeta = L\mu^T \left(\Gamma^T\Gamma\right)^{-1} \mu \qquad (2.25)$$

where

$$\begin{aligned}
\mu &= L^{-1} \sum_{t=1}^{L} \frac{\Omega(t)\epsilon_y(t)}{\sigma_\epsilon} \\
\Gamma^T\Gamma &= L^{-1} \sum_{t=1}^{L} \Omega(t)\Omega^T(t)
\end{aligned}$$

where σ_ϵ^2 is the variance of $\epsilon_y(t)$. The model is generally regarded as adequate if for several different vectors $\Omega(t)$, the corresponding values of ζ lie within a 95% acceptance region:

$$\zeta < k_\alpha(s)$$

where $k_\alpha(s)$ is the critical value of the chi-square distribution with s degrees of freedom and a level of significance $\alpha = 0.05$.

2.5.4 Network Transparency

In validating a trained ANN, it would be desirable to have an understanding of the influence of each weight on the network output. This enables the designer to search for regions where the network has been insufficiently trained and aid the evaluation process as the magnitude of the parameters would be directly related to the size of the network output. However, many ANNs are *opaque*; the knowledge which is stored in the network is distributed across many parameters in a complex manner which cannot easily be understood by the designer. Some attempts have been made to interpret MLPs (for example) using a truncated Taylor series or finite Volterra model [Soloway and Bialasiewicz, 1992], although neglecting the higher order terms of a sigmoid expansion loses the fine detail, and it is conjectured that these higher order terms provide the representational advantage of globally generalising nodes for *smooth* functional approximation tasks.

One notable exception to this problem is the B-spline lattice AMN, and to a lesser extent the higher order CMAC networks. Their internal structure can be interpreted as a set of fuzzy production rules (see Section 9.2.1 and Chapter 10) such as:

IF (x_1 *is positive small* AND x_2 *is almost zero*)
THEN (y *is negative small*)

This provides the AMNs with some degree of transparency, although it is generally only true for small-dimensional spaces, as the number of rules is exponentially dependent on the number of inputs. More complex input space quantisation strategies and network architectures [Kavli, 1994, Zhang and Grant, 1992] can make these techniques applicable to higher dimensional modelling problems although, unless the redundancy is explicitly represented, the fuzzy rule base may be overly complex.

2.6 DISCUSSION

The majority of the currently used ANNs are model based. The structure of the network remains fixed during training, and only the weight vector is adapted. In

this context, these algorithms should be subject to a rigorous analysis in order to determine their advantages and disadvantages when compared with other learning model based algorithms. A learning algorithm can be classified according to its basic modelling flexibility, its learning capabilities and the model transparency (the ease with which the final model can be understood by the designer). This chapter has reflected that philosophy. Human learning provided the inspiration for many of the algorithms described in this section, but these techniques must be properly understood if they are to be applied successfully. The argument that, because these algorithms are based on human thinking and human thinking cannot be analysed, *therefore* results cannot be proved about these techniques is incorrect. Many of the so-called neural learning algorithms have their counterparts in the adaptive control and signal processing literature: LMS $\equiv$ MIT rule, and NLMS $\equiv$ Kaczmarz's algorithm (Åström and Wittenmark [1989], Kaczmarz [1937] and Chapter 5), and many theoretical results are now known about their performance. The same amount of rigour should be applied to the neuromodelling and control research, otherwise results will be *strictly* problem dependent.

The biological motivations and implications of these algorithms have been deliberately understated in this chapter and in the remainder of the book. Biological research should stimulate and motivate this research, but it does not have to guide the work blindly. In the introduction it was stated that it may be another fifty years before a first-order understanding of the brain's functionality is achieved, yet control engineers want these results *today*, with provable convergence and stability properties.

The modelling capabilities of an ANN should be flexible enough to achieve the desired objectives, but if a model is too flexible, it can overfit the data and the generalisation is poor. Learning from a data set can be posed as a functional approximation task, as it is not enough to simply learn to reproduce the training data, the underlying structure of the function must also be learnt from the samples, otherwise the network is not able to generalise correctly to neighbouring inputs. The adaptive ANN should also be well conditioned, as generalisation *requires* both a good model structure and parameter convergence. If the network is badly conditioned, the designer may be fooled into thinking that it is performing well when the prediction errors are small. However, parameter convergence in a badly conditioned network (of the correct structure) occurs *slowly*, and prematurely stopping the learning procedure means that the network is unable to generalise appropriately.

One of the original objectives in the ANN field was to develop algorithms which were capable of learning information *on-line* [Narendra and Mukhopadhyay, 1992, Narendra and Parthasarathy, 1990]. Many neuromodelling and control applications have tended to ignore this and the ANNs are used solely for their nonlinear approximation ability. To exploit the full potential of this area, *on-line* adaptation is *essential* in order to cope with time-varying plants operating in dynamic environments. Many of the learning rules currently developed are based on gradient descent, as originally occurred in the adaptive control field. How-

ever, it was found that under certain conditions (for instance, if the learning rates were set too high), the resulting closed-loop systems could become unstable. This led to stability-based learning laws being developed during the sixties and it is only recently that similar adaptive rules have been proposed for neural networks [Sanner and Slotine, 1992, Tzirkel-Hancock and Fallside, 1991]. The present stability theories are directly applicable to some ANNs, although it would be hoped that new results and algorithms could be developed which exploit network specific properties.

Future research must be aimed at understanding the modelling properties of the various ANNs, and proposing new architectures to improve their suitability for modelling and control applications. The inspiration for such improvements could be biologically based or come from the related fields of adaptive control and signal processing. Possibly, a more important topic is to understand and derive new learning laws which are applicable to various ANNs. The ability to prove convergence and stability is *crucial* if these learning algorithms are to be applied for on-line adaptive control, and for certain classes of networks which are *linear* in their weight vector (RBFs, lattice AMNs), the theories developed for adaptive linear systems can be modified and applied to these networks [Narendra and Annaswamy, 1989]. These algorithms should be compared with other nonlinear adaptive control strategies and the advantages in using these networks must be clearly quantified. This can only be achieved when theoretical results are available to compare the functional approximation and learning convergence of the different algorithms. Finally, the problems associated with applying these techniques to real-world tasks should not be underestimated. Algorithms need to be developed and improved for identifying the relevant network inputs, monitoring the adaptive networks and verifying and validating the knowledge stored in the network. For most practical applications these are non-trivial problems, but are essential for a successful implementation.