Learning and Generalization in Feed-Forward Neural Networks

Frank J. Śmieja

Submitted for the degree of
Doctor of Philosophy

Department of Physics
University of Edinburgh

1989

To my parents, Joy and Jan,
dziękuję,
bo daliście mi wybór.

Declaration

This thesis has been composed by me and all the work is my own apart from some of the work in chapter 3, which was done in collaboration with Gareth Richards.

Some of the work has been published in [ŚR88, Śmi88, Śmi89].

Acknowledgements

This work has been supported by the Science and Engineering Research Council.

I would like to acknowledge the support and encouragement of my supervisors David Wallace and Alastair Bruce, and thank them for their useful comments on this thesis. Although my former supervisor, Elizabeth Gardner, was unable to see through the completion of my thesis, I would like to acknowledge her encouragement and stimulating discussion in the early stages of this work. We all miss her greatly.

Throughout my time in the physics department I have enjoyed an open and friendly working environment, and I would like to thank my fellow postgraduates for encouraging free flow of their expertise, from which we all benefit. I would especially like to thank Nick Radcliffe, Brian Wylie, Greg Wilson, Steven Booth, Andrew Thornton and Lyndon Clarke for showing me what it means to compute, and Nick Stroud for his friendly and cheerful attitude to the whole business. I would also like to thank Winston Sweatman, John Lunt and Rik Eyre-Todd for putting up with my persistent questioning, and especially Winston with the material concerning the equivalence relations in chapter 5.

For keeping my pecker up when I was down, and thrashing me at snooker, tennis, chess..., when I thought I was up, I say mmmmerci, grazie, danke, dziękuję, ta, to Bruce Forrest (na'at'a'mean!?). Thanks too to the guild of Georges (so cruel, but I love it — reeeally?!), to Steve Hayward and his sobering logic and wit from old London town, to Nick Radcliffe and Gareth Richards, to Martin Simmen, Jeremy Craven and Nigel Wilding (the three Stooges), and to Lydnod Clarke and Stewartie Springham for keeping me up on Tuesdays and convincing me that I was not such a bad potter after all...

Abstract

Aspects of learning and generalization in feed-forward neural networks are studied. The networks are taught using the backpropagation learning algorithm.

The performance of the algorithm is studied using a training set which can be made to have a variable difficulty. Using such a training set the performance is evaluated and improvements and modifications suggested.

A simple classification of problem domain types is made and a particular class is suggested to be the most appropriate for the 3–layer feed-forward network to learn. This class is characterized by underlying regularities among the training set members, such that the mapping required for each pattern in the training set is consistent with all the other required pattern mappings. The suitability of this class of training sets is demonstrated with observation of the emergent properties of the network in actual learning speed and nature, and in the generalization ability displayed after learning an incomplete training set. This behaviour is contrasted with training sets not possessing the underlying properties of this class, from which it is concluded that this type of network is more effectively used for extracting salient information about a training set, given that underlying regularities exist, rather than for other classes of mappings.

The dependence of generalization of the network on such problem domains is studied as a function of hidden layer size. It is shown that in general the number of different solutions available in the algorithm's search space increases rapidly with the hidden layer size. Despite this, it is shown that the generalization performance does not degrade correspondingly, but in fact remains at a steady high level. This observation suggests that the salient information about a training set is more likely to be extracted during learning, as opposed to merely mapping the patterns independently (which form a large set of other possible solutions), and that this information is stored in a distributed manner throughout all the weights of the network.

Contents

Chapter 1

Outline and motivation of the thesis

1.1 Background

One of the fundamental goals of Artificial Intelligence is to develop machines capable of learning about their environment, from their interaction with the environment and the changes in the environment. Perhaps most important is a capability of generalizing from experiences encountered in the environment so as to be able to make predictions and thenceforth plans and strategies in further interaction with the environment. The standard approach to this task is based on the construction of theories and computer programs which embody all the sensory, syntactical and reasoning skills required in the machine's interaction with the particular, well-defined environment. The initial emphasis is generally on stringent and unambiguous definition of machine responses to typical environmental situations. If general rules can be identified with respect to desired forms of interaction then these are also included as *a priori* rules which can be obeyed by the machine. However, the problem of how to enable a system to learn is, as pointed out by McCarthy [McC75], inseparable from that of how to represent the knowledge concerned. Connectionism, or neurally-inspired models, consider correct representation to be the key to successful learning and generalization.

These models are inspired by observation of the processing power and speed achieved in the brain, and attempt to reproduce this through having a basic structure of simple processing units ("neurons") connected together by weights ("synapses"). The speed is to be emulated by the **parallel** operation of sets of units, and the complexity of processing by the mutual interactions of all the units. The emphasis is on trying to evolve non-localized, non-symbolic, robust representations of tasks in the machine by some form of learning from experience, and to use such representations to achieve the above-mentioned goal of generalization.

1.2 The research in this thesis

In this thesis we make a study of a particular type of connectionist model, or *neural network*, and its associated learning algorithm. The neural network is known as **feed-forward** and the learning algorithm **backpropagation.**

Such a study is interesting from the physics point of view, because it involves the operation of complex systems of nonlinear units, from which it is desired to obtain firstly an *optimization* power, and secondly an emergent *generalization* effect. This generalization can be viewed as a synergetic effect of the combined influences of the features extracted from the input space and stored in the weights.

As in many nonlinear systems, especially complex systems such as this, the theoretical work which can be done is limited, and so it is more expedient to perform computer simulations, observe general effects, and from there to make predictions as to the behaviour of other such systems. Nevertheless, if it is possible to reduce the problems to a simpler level, the deductions made will be easier to formalize, and probably more general. This research is performed in just such a reductionist spirit, exemplified by the approach to the rounding problem of chapter 3, which leads to the more general deformation tool, and the reduction of the "natural"[1] problem domains to a level at which we can make the analogy to simple spin systems (chapter 4).

Important features of the algorithm and considerations which should be made in its use are explored. The main thrust of the work involves the study of the

[1]This term is explained in section 4.1.

learning and generalization capabilities of this model when the problem domain used involves underlying **correlations** between its members. We also introduce a method of defining solutions to simpler problem domains discovered by the network, in terms of the representations of patterns in the "hidden layer" (the layer in which the units are not required to have any particular states, as opposed to the input and output layers).

1.3 Organization of the thesis

Chapter 2 introduces the ideas of connectionism and outlines the difference between these types of models and the symbolic program structures also used in Artificial Intelligence to allow learning and generalization. The ideas of learning and generalization in the connectionist framework are explained, and the forerunner of the multi-layer perceptron, the perceptron, is described and discussed, before the multi-layer perceptron itself is introduced. Finally, we compare the processing performed by perceptrons and multi-layer perceptrons with traditional pattern classification techniques, and suggest how the multi-layer perceptron, through learning about the input space rather than using parametric or non-parametric techniques explicitly to fit generating functions to it, can be viewed as a simpler and more general form of classifier.

Chapter 3 presents the details of the feed-forward network model and backpropagation algorithm used, and results of its performance on the rounding problem. Various properties of the net and the basic algorithm are studied, using learning curves, error maps and net scaling. Two major improvements are suggested: the deformation procedure and a parameter changing procedure. The deformation procedure is applied to another problem, noisy digit restoration, in which it is shown how the procedure improves performance, and the parameter changing procedure is employed again in later simulations for efficient minimization.

In chapter 4 we offer a simple classification of problem domains, and then study the suitability of one of them, the "natural" domains, through preliminary experiments with learning protein sequences. Simplifying assumptions as to the underlying nature of this and other such problem domains are made to reduce their basic characteristics to a form such that an analogy can be made to a spin-like model. The learning and generalization of this new training set and two other types of training set, not possessing the characteristics of the natural problem domains, are compared. We observe that the former training set gives rise to faster learning, a far smaller requirement of hidden units to perform the mappings, and clear generalization. We conclude that the network learns about the underlying correlations used in the generation of the pattern in the training set and that it is this which enables the observed emergent properties to arise.

Chapter 5 is involved with the question of the influence of training set content and size and hidden layer size on generalization performance, for the type of network defined in chapter 4. First we introduce the idea of a "solution", defining it in terms of the relative hidden-unit representations of the training set patterns. This idea is used for the estimation of how the number of possible solutions which exist for the parity problem scales with the size of the network, in terms of input and hidden units. The rapid increase in solution number with hidden unit size is suggested to be a cause for concern, since the generalization performance may degrade with the increase in the number of available solutions to the problem. In order to test this, the generalization behaviour of the network on an artificial diagnosis network, derived from similar underlying correlations as the training set in chapter 4, is examined. The results of this examination allay the scaling fears expressed above, and this is explained in terms of the most likely solutions found by the algorithm being those which display the very emergent properties which are desired.

Chapter 2

Introduction

In this chapter the ideas of connectionism will be introduced.

First we describe some "non-neural" ways in which a machine may be taught to learn about the environment, with reference to some well-known examples from Artificial Intelligence (AI). A brief review like this is useful in bringing to light the main difficulties associated with getting a machine to learn and generalize sensibly. It is also interesting to note techniques, such as the optimization procedure which is used in Samuel's checker player, to find appropriate values for parameters in an evaluation function. This operates in its basic form as hill-climbing in the parameter space — the same basic idea as in the network algorithm used in this thesis. However, the ideas behind connectionism differ in the fundamental representational structure of the models, and in the way in which parameters are modified through experience with the environment.

Having made the distinction between these two ways of getting machines to learn, we concentrate on the technical aspects of feed-forward networks, describing the idea of distributed representations, and learning and generalization within this framework. The forerunner of the multi-layer perceptron (MLP), the perceptron with its simple learning rule, is briefly described. Having noted the limitations of this model, we then introduce the multi-layer perceptron, and explain how the complexity of mappings it can perform in principle overcome many of these limitations. The range of functions it is possible for the multi-layer perceptron to realize results from the number of layers of nodes it possesses, combined with the nonlinear response functions of the nodes.

Finally we outline the similarities of the optimizations performed by perceptrons and multi-layer perceptrons with classical pattern classification techniques, and suggest that it is the greater generality of pattern distributions realizable in theory by the neuron-based models, that makes such models both interesting and worthy of study.

2.1 Learning by experience

As has already been mentioned, the most important difference between connectionist models and conventional AI models is the way the knowledge is *represented* in the model. In connectionist models the representation of the knowledge is organized such that the knowledge unavoidably influences the course of processing. In this section we demonstrate this distinction, with a brief review of some well-known symbolic programs for achieving artificial learning. In the next section we shall describe in detail the neuron-based methods.

J. M. Tenenbaum *et al* [TGWW74, TW75] constructed a program which was able to recognize various common objects in grey-scale photographs after a period of instruction which allowed the reallocation of symbols to various structures. As an example, consider the recognition of a telephone.[1] The program does not have a telephone described to it explicitly, but is told that such an object exists in a highlighted field of view (for example a portion of a photograph) and defines necessary constraints on what a telephone looks like itself, by eliminating unnecessary features through a form of indirect questioning of the teacher (operator). The program uses two data structures which represent two types of concepts that it learns about, semantic and iconic. Thus the semantic data structure would contain the information that a telephone is characterized by a black rectangular block of medium size supported by a table which itself supports a black rectangular wedge with a grey square area with small blocks equally spaced in the square

[1]The telephone example used below is taken from [Bod87], where it was used in a slightly different form.

area on its sloping surface. This description is gradually deduced by the program as it becomes more discriminatory, making use of the iconic representation (the visual, pictorial description of the object). Thus the program can learn what an object looks like either by seeing it, or by being told that it resembles something seen before. The operators (symbols) which may be useful for the program to identify particular objects are suggested by the teacher and a semantic and iconic description is built up by the program. The usefulness (whether they are sufficient to define an object) of the operators is then tested by getting the program to isolate areas in the picture satisfying the current requirement for classification as each object, and then further operators are chosen, and so on.

Thus this program dynamically alters its understanding of the environment by interaction with it and the teacher, building more constraining and complex data structures along the way. Here learning can be achieved by the system *only* if it has the ability to construct, analyze and manipulate complex symbols. Thus the knowledge is built up as it is encountered, rather than incorporated into the pre-existing knowledge. The generalization possible here is thus limited by the generality of the object descriptions. Also the system is limited by its dependence on the teacher for guidance on *which* features to use as discriminatory descriptions of images.

P. H. Winston tackles the latter requirement in a program which learns to recognize **structures** (such as arches, bridges, etc.) merely by being shown examples and counterexamples of them [Win75]. All that a "teacher" is required for is to provide information as to what is and is not an example of the structure being considered. The nature of the world is necessarily already part of the program's structure, i.e. the program can manipulate various building-block concepts such as "brick" and "prism", and connecting concepts such as "supported by" and "marries", etc. from the start. Using the examples and counterexamples of various types of object shown to it, it can deduce their essential descriptions in terms of these basic concepts. It can also build on its world-knowledge using the concepts it has defined itself, to new definitions for more complex structures, which may be constructed from several of the structures it now knows about.

This program is interesting in that it searches explicitly for the necessary and sufficient features of a particular structure through progressive information derived from examples and counterexamples of the structure, which is similar to the way connectionist models work. In fact, the representational structures of the objects are in the form of connecting pointers (unit "weights") to descriptive relations between concepts ("nodes") and the network of relative pointers is altered as the new examples are seen. However, this program can not quite be described as a type of neural network, albeit its method of extracting new information from patterns rather than storing complete images is an interesting parallel. The program lacks any appreciation of the problem, in that it considers and stores every feature and does not look for anything in particular in an image. It also cannot but specify each description stringently rather than in a broadly defined way. Conversely, the neural models are designed to extract information in a less constrained manner, thereby allowing the possibility of varying descriptions through the influence of underlying regularities.

One of the more successful learning machines was Samuel's checker (draughts) playing program [Sam63]. Part of the program's mechanism involved learning from mistakes and good moves made during the game playing (a large part of the machine's knowledge also came from "rote" learning, where the values of moves judged by human experts were fed into the machine explicitly). Each node of the tree in the move-searching (minimax procedure) has a value assigned it, indicating the "goodness" of the move which it represents. The evaluation function from which these values are derived consists of a number of parameters marking strategic features of a game. The learning procedure involves *improving* the evaluation decisions, by continually adjusting the weighting of the test parameters involved, according to their success in actual performance. Samuel's program improved with practice to such an extent that it once beat a checkers master — the moves proving to be original after about half-way into the game. Samuel's program learnt in the sense of changing its understanding of the world only in altering the evaluation function, according to the success of the comprising parameters. Thus the machine was used to *optimize* the form of a function whose nature could only be specified *approximately*[2] by the human programmer, by varying its coefficients according to the feedback from the match-play. The actual process of the coefficient-changing is interesting for comparison with neural net learning, in that: "... the entire

[2]By the inclusion of 38 possibly useful board features.

learning process is an attempt to find the highest point in multidimensional scoring space in the presence of many secondary maxima on which the program can become trapped." [Sam63]. Learning is essentially a hill-climbing procedure in the scoring space, with the possibility of extrication from local maxima by manual intervention.

The similarities in Samuel's learning procedure and neural net techniques lies only in the fact that a cost function is being optimized, the representation of information and the basic mechanism being totally different. Generalization is possible, and good, through the unconstrained manner in which information is built into the evaluation function. The only drawback of this model of learning however, is the initial requirement that an evaluation function be defined and suitable terms specified. Connectionist methods may partially eliminate this requirement through the unconstrained extraction of relevant features from a problem domain. In this thesis we explore this basic principle through observation of the generalization performance of one class of connectionist models, the **feed-forward net.**

Not surprisingly, more "intelligent" programs have been written since Samuel's Checkers program, involving deeper knowledge of the actual task required. Some of these programs achieve generalization through the manipulation of new symbolic names in structures which are learnt, rather than extracting information [FN71, Sus75].

The basic idea behind all these programs has been the allocation of symbols to features in the environment, and then manipulating these symbols. The basic mechanism of connectionist models is in the *distributed* representation of features, and their interaction, to produce emergent properties such as generalization.

One of the first examples of a network approach to making decisions was Selfridge's Pandemonium system, in which decisions are made on the basis of "which demon shouts the loudest", with the demons being in a hierarchical layered structure [SN63, Sel59]. The learning was a simple type of hill-climbing. In such a system, however, each demon has to have a function assigned it, as opposed to the distributed, non-localized nature of connectionism.

2.2 Connectionism

2.2.1 Introduction

In many tasks performed by humans, and in the tasks it would also be desirable to have performed by machines, a number of different pieces of information must be kept in mind at once. Each plays a part, constraining others and being constrained by them.

Parallel Distributed Processing (PDP) models assume that information processing takes place through the interaction of a large number of simple processing elements, each sending excitatory and inhibitory signals to other units. These models are composed of many nonlinear computational elements operating in parallel and arranged in patterns reminiscent of biological neural nets. It may be that all or some units represent possible hypotheses or goals and actions, with the connections representing the constraints the system knows to exist between the hypotheses or the relationship of goals to subgoals, to actions and so on. The most robust and non-localized forms of these networks assume no particular representational rôle for some of the units, but merely link aspects or correlations of the information possessed by the network in a distributed and non-fixed way. The computational elements are connected via weights that are adapted during use to improve performance.

For example, a network concerned with the processing of visual data might consist of a set of units whose job it is to process activations from a set of receptive areas on the retina in such a way that the structure of images received as input may be identified and so define a concept for further processing. In order that this might be possible, another set of units, which may not have any particular identification with retinal points, respond to certain characteristic shapes, edges etc., eventually to produce a set of activations somewhere else to indicate the association of a particular concept with the image.

Thus such networks by their very structure process in parallel a number of simultaneous constraints present in a certain input, and are in this way able quickly

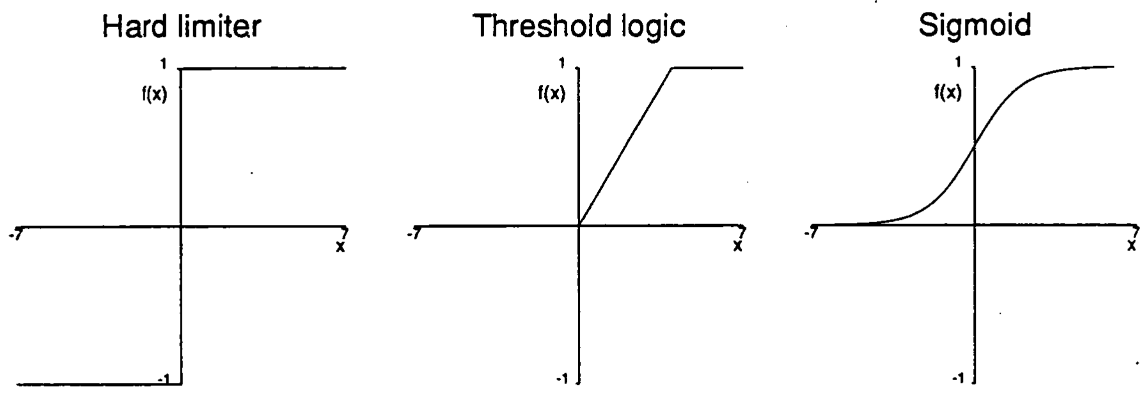


Figure 2.1: Three types of nonlinear response function commonly used in connectionist models.

to link concepts (or actions) with inputs characterized by many contributory features.

Computational elements (nodes) used in these models are nonlinear, typically analogue, and may be slow compared with modern digital circuitry. A simple node sums N weighted inputs and passes the results through a nonlinearity (see figure 2.1). We demonstrate in section 2.2.5 how important this nonlinearity is, as in many areas of physics, for giving rise to more interesting behaviour. The node is characterized by an internal threshold or offset θ and by the type of nonlinearity. The three common types of nonlinearity used are illustrated in the figure. These are the hard limiters, the threshold logic elements, and sigmoidal nonlinearities. The actual network model is specified by the functionality of the node and the learning rule used, and by the nature of the connections. The learning rule specifies how the weights (connections) should be updated during use to improve the net performance.

Neural nets in theory provide a greater degree of robustness or fault tolerance than von Neumann sequential computers, through the possibility of a large number of processing nodes, each processing mostly local connections, or more fundamentally, each representing a small part of a number of pieces of information which are distributed throughout the net. Thus damage to a few nodes or links need not impair overall performance significantly. The possibility of adaptation or learning is one of the major attractions of neural net models, especially in areas such as speech recognition, where training data is limited and new talkers, new words, dialects and phrases are continually encountered. Robustness is also provided by the compensation of minor damage to nodes or weights during further adaptation.

In connectionist models, the knowledge about any individual pattern is not necessarily stored in the connections of a special unit[3] reserved for that pattern, but may be distributed over the connections among a large number of processing units. This allows generalization on underlying pattern trends to take place (chapter 4).

2.2.2 Learning

The representation of the knowledge in a net is set up in such a way that it necessarily influences the course of the processing. Using knowledge in processing does not mean that one has to locate the relevant information in memory and make use of it; the knowledge is intrinsic in the processing itself.

Now, if the knowledge is the strength of the connections, learning must be a matter of finding the right connection strengths so that the right patterns of activation will be produced under the right circumstances. This is a very important possibility — an information processing system which can learn — because then such a system could learn to capture the interdependence between activations that it is exposed to in the course of the processing.

The basic approach of connectionism or PDP models to the question of adaptability is different to traditional symbolic learning techniques in that firstly, the goal of learning is not assumed to be the formulation of explicit rules. The goal is taken to be the acquisition of connection strengths which allow a network of simple units to act *as though* it knew the rules. Secondly, the learning mechanism is not attributed with powerful computational capabilities. Instead one

[3]but may be. See for example section 3.6.3 for the "grandmother cell" idea, and the type of generalization which this may afford.

assumes very simple connection strength modulation mechanisms which adjust the strength of connections between units based on information locally available at the connections.

2.2.3 Generalization

The possibility of some form of generalization arises when the network learns a set of mappings which involve some or all of the weights being used strongly during the presentation of more than one pattern. This can be illustrated with the simple pattern associator [WBLH69, AS87]). The pattern associator consists of two sets of units, with connections from the first (input) set to the second (output) set. A pattern of activation over the input causes a pattern of activation over the output. The simplest way in which such an associator can learn is through the use of the Hebb rule [Heb49] — when units A (input) and B (output) are simultaneously excited, increase the strength of the connection between them. Or, mathematically,

$$\Delta w_{AB} \propto S_A S_B \tag{2.1}$$

where the activations of the units A and B have values $\{1, -1\}$.

Now, if one wishes to learn multiple non-orthogonal patterns in the same set of representations, one may experience two distinct synergetic effects. Firstly, if the set of mappings required fall into some consistent trend depending on some underlying characteristic of the data, each pattern and therefore each weight-change will in some way cause the other patterns on average to produce activations closer to the desired outputs. Using the Hebb rule and the simple associator, the class of mappings for which the effect is positive, due to underlying consistencies, is quite small, but other nets employing continuous-valued node states and weight changes allow significant accumulative learning effects. Secondly, with the information of a particular class of mappings stored in the network weights through all the example patterns used, the network most likely has retained only the relevant **structure** of a pattern that makes it a member of this class, and thus may be reasonably expected to display such general knowledge of the class in deciding

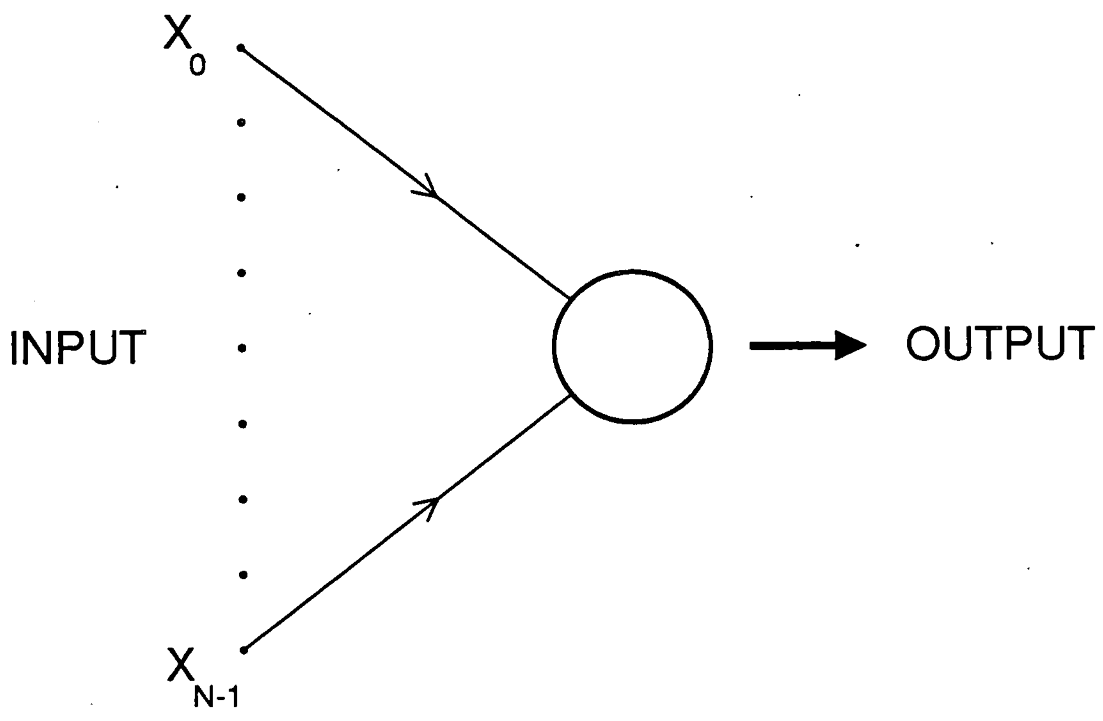


Figure 2.2: A perceptron.

on the classification of a previously unseen pattern. Such an effect is known as generalization, and illustrates what is meant by a network acting *as if* it knew the "rules" but not actually storing them in an explicit way.

2.2.4 Perceptrons

Basically, a perceptron is a device which computes a state y by processing the weighted sum of a set of simultaneous inputs $(x_0, x_1, \ldots, x_{N-1})$ through some function (usually a hard nonlinearity).

Figure 2.2 shows a perceptron that decides whether an input pattern belongs to one of two classes (A or B), depending on whether the computed state y is high or low. The perceptron computes a weighted sum of the input elements, subtracts a threshold (θ) and passes the result through a hard limiting nonlinearity so that the output y is either $+1$ or -1. The decision is taken to be class A if the output is $+1$ and class B if the output is -1.

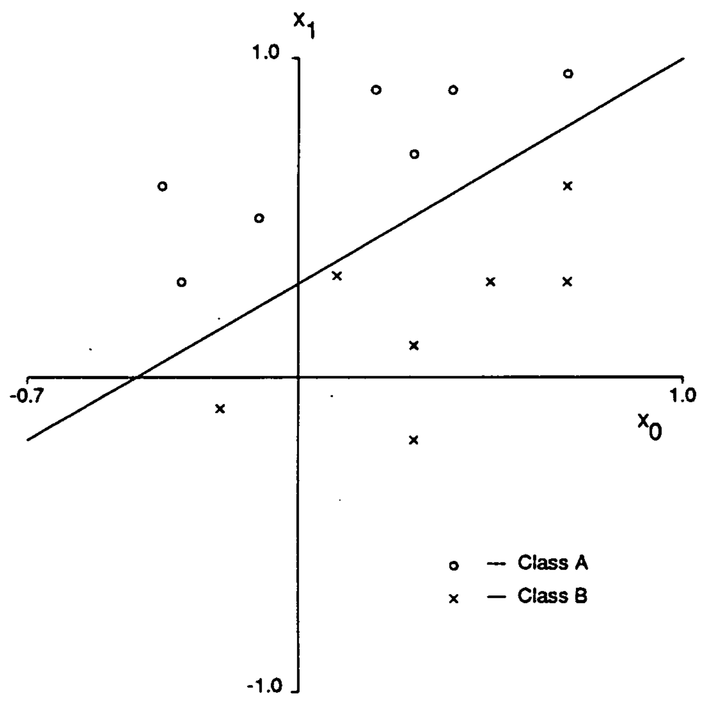


Figure 2.3: A decision boundary to be found by a perceptron, separating the two classes A and B.

For the purposes of explanation of the behaviour of such a device, consider a perceptron with just two inputs. One can then plot a map of the decision regions created in the 2–dimensional space spanned by the input variables. These regions specify the input values which result in a class A and class B response by the perceptron. The decision regions are separated by a hyperplane, which in 2 dimensions is a straight line. Figure 2.3 shows such a decision boundary. The equation of this line is

$$x_1 = -\frac{w_0}{w_1}x_0 + \frac{\theta}{w_1} \tag{2.2}$$

and so its orientation and position is dependent on the connection weights w_0 and w_1 and the threshold θ. In order that the perceptron might distinguish a whole range of classes (rather than those it distinguishes in its initial configuration) it must be *taught* using a learning algorithm. The learning algorithm should adjust the weights and threshold such that the decision boundary is repositioned to classify correctly example patterns chosen from the classes A and B (if this is possible). Rosenblatt [Ros59] developed the original perceptron convergence procedure, which will be described here since it is similar in form, albeit much simpler than, the backpropagation learning algorithm used for the MLPs used in this thesis. This procedure is similar to the fixed increment procedure of pattern recognition [Ull73].

First the connection weights and threshold are initialized to small random non-zero values. Then a new input with N continuous valued elements is applied to the input, and the output (y) is computed. Connection weights are adapted only when an error occurs, using the formula:

$$w_i(n+1) = w_i(n) + \eta\{d(n) - y(n)\}x_i(n) \tag{2.3}$$

$$d(n) = \begin{cases} +1 \text{ if class A} \\ -1 \text{ if class B} \end{cases} \tag{2.4}$$

$$0 < \eta < 1 \tag{2.5}$$

where η is a gain term, controlling the adaptation speed, and the perceptron output state at time n, $y(n)$, is defined by

$$y(n) = F_h(\sum_{i=0}^{N-1} w_i(n)x_i(t) - \theta), \tag{2.6}$$

F_h being the Heaviside function.

The procedure is repeated for each pattern at every time-step n until either all the mappings are reproduced correctly, or the weight vector is seen to cycle repeatedly and fails to improve the performance (this occurs when the classes are not linearly separable).

Associated with Rosenblatt's learning procedure there is a convergence rule, which states that if two classes are linearly separable then the perceptron convergence procedure converges. In other words, if a hyperplane exists which separates the two classes, the perceptron convergence procedure will find it in a finite time.

The perceptron convergence procedure is clearly very simple and numerous modifications can be, and have been, made to the basic idea to allow the perceptron to perform better when the condition of the convergence theorem is not satisfied. One such example is the Widrow-Hoff algorithm [WH60].

It is the very simplicity of the perceptron which allowed the possibility of a convergence theorem, and simple analysis above, which was also responsible for the possibility of a complete mathematical analysis of its fundamental capabilities and limitations. The obvious limitation is its clear inability to separate classes if a hyperplane cannot be drawn between them. This eliminates the vast majority of general mappings. This can be resolved using more layers of perceptrons and combining their decision boundaries to form particular shapes, but then it is (currently) not possible to prove a convergence theorem for whatever learning procedure now has to be used.[4] Using the simple framework of perceptrons, Minsky and Papert [MP69] were able to define the types of mappings which could not be performed and, more importantly, show the generally poor scaling performance that can be expected. (Incidentally, the book by Minsky and Papert is viewed by many to have contributed to the lack of research funding in this area of Artificial Intelligence for the next decade or so [Ola89].)

2.2.5 Multi-layer perceptrons

Multi-layer perceptron (MLP) is a generic name given to feed-forward nets with one or more layers of nodes between input and output layers. The nodes may have any type of nonlinear or linear response function. The layers which are neither input nor output layers are known as **hidden** layers, since their actual states at any particular time are not required to be anything in particular, unlike the input nodes (whose states are fixed by the input pattern) and the output nodes (whose states represent some specific value associated with the input patterns). Figure 2.4 shows a general MLP.

Many of the limitations of perceptrons pointed out in [MP69] can be overcome by nonlinear MLPs, in so far as mapping complexity is concerned. A theorem by Kolmogorov states that any continuous function of N variables can be computed using only linear summations and nonlinear but continuously increasing functions

[4]Apart from stochastic methods, such as the Boltzmann machine [AHS85, Bou86].

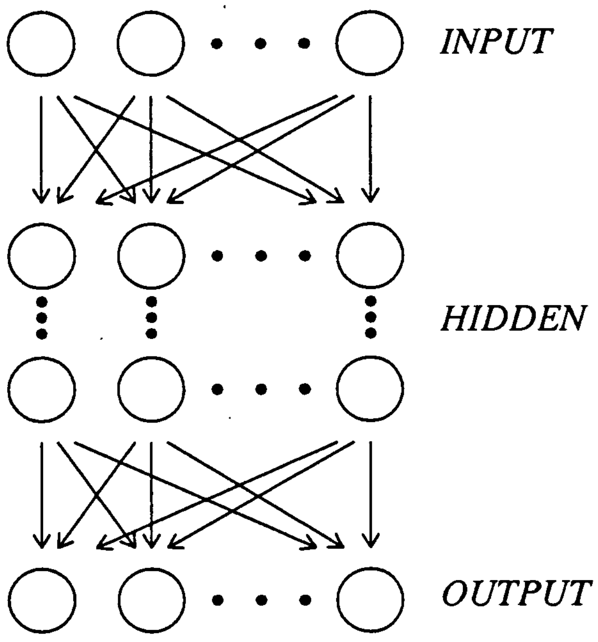


Figure 2.4: A general multi-layer perceptron (feed-forward network).

of only one variable [Lor76]. Effectively it implies that a three layer perceptron with $N(2N+1)$ nodes using continuously increasing nonlinearities can compute any continuous functions of N variables [Lip87]. However, there is the inevitable problem of a satisfactory learning procedure to be solved. It will be demonstrated in chapter 3 how the simple gradient descent search in a multi-dimensional space can be used to alter the weights in such networks, given a global cost function.

The benefits of MLPs are possible directly as a result of the nonlinearities used in the node response functions, combined with the extra layers of processing or mapping. Several layers of *linear* processing units can be shown to be equivalent to using just two layers (input and output), i.e. the linear hidden layers make no difference to the decision region complexity which the network can form. This can easily be seen if we consider each set of weights between two layers l and $l+1$ as an N_l by N_{l+1} matrix $(T_{l+1,l})$, where N_l is the number of nodes in layer l. Thus each matrix of weights $T_{l+1,l}$ performs a transformation on the vector defined by the states of the nodes in layer l, $\underline{v}_l = (v_{l1}, v_{l2}, \ldots, v_{lN_l})$. Thus the states of the layer $l+1$ become:

$$\mathbf{v}_{l+1} = L(T_{l+1,l}\mathbf{v}_l^T). \tag{2.7}$$

Given that L is a linear function, the left-hand side of (2.7) can be rewritten $M_{l,l+1}\mathbf{v}_l^T$ where M is a new matrix given by the product of the matrix T and the diagonal matrix defining the linear function L. Thus the output vector $\mathbf{v}_m$ (states of the output nodes) is given by:

$$\mathbf{v}_m = M_{m,m-1}M_{m-1,m-2}\cdots M_{2,1}\mathbf{v}_1^T, \tag{2.8}$$

where $\mathbf{v}_1$ is the input vector. But the product of matrices M in (2.8) is equivalent to one matrix $S_{m,1}$, and so the hypothetical multi-layer linear network with the weight matrices $T_{2,1}\cdots T_{m,m-1}$ and the linear response functions represented by the diagonal matrices $L_m \cdots L_2$ is simply equivalent to a two-layer network with identity response functions and the weight matrix $T_{2,1} = S_{m,1}$.

To demonstrate the types of decision regions particular MLPs can support, as a measure of the complexity they can manage, consider networks of nodes with hard limiting nonlinearities (Heaviside functions).

Then the 2 layer perceptron can form a decision region in a 2–dimensional input space defined by a single straight line. Thus the 2 classes are given respectively by the region on one side of the line and the region on the other side. With 3 layers of nodes one can consider each of the nodes in the hidden layer as separating the classes with a straight line (i.e. acting as 2–layer perceptrons). The output layer then combines all these boundaries to form a convex open or closed decision region (the region formed by the intersection of a number of straight-line segments). Similarly, a 4–layer perceptron has an extra layer again and thus the output layer combines all the convex open or closed decision regions to which the nodes in the layer above respond. Effectively then, given enough nodes, each node in the layer before the output in a 4-layer MLP can isolate a single point in the input space, and then each node in the output layer can combine any of these. Thus, in principle, the 4-layer perceptron, can form regions of arbitrary complexity, limited only by the number of nodes in the hidden layers.

The difference when a sigmoid response function is used is in the shape of the boundary lines, and effectively modifies the final decision regions into a collection of curved segments rather than line segments. The degree of curvature can be modified through variation of the gain parameter β in the sigmoid equation:

$$y = \frac{1}{1 + e^{-\beta x}} \tag{2.9}$$

The sigmoid nonlinearity is used in general in MLPs because a response function is required for the backpropagation learning algorithm which is differentiable, and the sigmoid has a particularly simple derivative (see chapter 3), while also resembling to a reasonable approximation the desirable hard limiting nature of the step nonlinearity.

Although the limitations on the *complexity* of mappings which can be performed by perceptrons is in principle overcome by nonlinear multi-layer perceptrons, the analyses of Minsky and Papert [MP69] regarding order and coefficient size suggest that various kinds of scaling problems are likely to stand in the way of attempts to exploit their potential. However, s uch obstacles may perhaps be avoided if suitable network architectures and learning rules are employed.

2.2.6 Feed-forward networks and pattern classification

Consider the problem of estimating the conditional probability $P(r|\mathbf{X})$ that, given a pattern $\mathbf{X}$, it is a member of pattern class R_r. Once this is done, the patterns can be classified according to the maximum likelihood decision rule, i.e., an unknown pattern $\mathbf{X}$ should be assigned to the class R_s such that, for all r except $r = s$,

$$P(s|\mathbf{X}) > P(r|\mathbf{X}).$$

A parametric statistical way of doing this is to use Bayes law for conditional probabilities to reduce the problem to determining not $P(r|\mathbf{X})$, but the conditional probability $P(\mathbf{X}|r)$ of a pattern being $\mathbf{X}$, given that it is class R_r. If we assume a 1–dimensional input pattern x, then we can see how a perceptron structure can be used to mimic a *gaussian classifier* [Lip87, Ull73].

If M_{si} and σ_{si}^2 are the mean and variance of input x_i when the input is from class s, and M_{ti} and σ_{ti}^2 are the mean and variance of input x_i for class t, and $\sigma_i^2 = \sigma_{si}^2 = \sigma_{ti}^2$, then the likelihood values are related to

$$\begin{aligned} L_s &= -\sum_{i=0}^{N-1} \frac{(x_i - M_{si})^2}{\sigma_i^2} && (2.10) \\ &= -\sum \frac{x_i^2}{\sigma_i^2} + 2\sum \frac{M_{si} x_i}{\sigma_i^2} - \sum \frac{M_{si}^2}{\sigma_i^2} && (2.11) \end{aligned}$$

and similarly for class t. The maximum likelihood classifier must calculate L_s and L_t to determine to which class to assign the pattern. The first term in (2.11) is identical for both classes and so can be dropped. It can be seen that a simple perceptron can calculate the difference between the second terms and between the third terms. This can be realized by setting weight w_i in the perceptron equal to

$$w_i = \frac{2(M_{si} - M_{ti})}{\sigma_i^2}$$

and the threshold θ equal to

$$\theta = \sum_{i=0}^{N-1} \frac{M_{si}^2 - M_{ti}^2}{\sigma_i^2}.$$

In general, the elements of the input pattern will be correlated in some way. If we assume the patterns are normally distributed, then we have that

$$P(\mathbf{X}|r) = \frac{\exp[-(1/2)(\mathbf{X} - \mathbf{M}_r)^T \mathbf{C}_r^{-1} (\mathbf{X} - \mathbf{M}_r)]}{(2\pi)^{N/2} (\det\ \mathbf{C}_r)^{1/2}} \qquad (2.12)$$

with $\mathbf{X}$ and $\mathbf{M}_r$ being vectors of the pattern and the means for class R_r. The gaussian classifier will estimate the rth class covariance matrix $\mathbf{C}_r$. This is done by estimates of the average correlation between pattern elements, given the pattern's membership to the class r. By making various simplifications, such as the statistical variability of all pattern elements being equal ($c_{rii} = c_{rjj}$), the patterns can be efficiently classified. A simple example of a *non*-parametric method of pattern classification is the *nearest neighbour* method, which determines simply the distance of an unknown pattern $\mathbf{X}$ from every other pattern in the training set, and finds the training set pattern which is nearest to $\mathbf{X}$. Various metrics may be used to determine this distance.

Various other parametric methods exist for such classification of patterns, but they involve strong assumptions about the underlying distributions.

It can be seen how the task of a neural net may be likened to parametric methods of pattern classification, and it may be argued that all a net is "really" doing is this basic function (one such example is the Boltzmann machine [AHS85]). Even if this were the whole story, we may also note *how* the neural nets do this. It is done through non-explicit assignation of suitable values to the weights, which allow the neural net to reproduce the input distribution. Thus neural net classifiers are non-parametric[5] and make weaker assumptions concerning the shapes of underlying distributions than standard statistical classifiers. In this way they tend to be more robust when distributions are generated by nonlinear processes and are strongly *non*-gaussian. Together with an easily-specifiable learning rule, such nets should be positively viewed as general and adaptable classifiers.

[5]In the sense of an initial assumption of the input space, although it may be argued that the nets act in a parametric way in the ensuing learning, through deciding on a structure for the input space.

Chapter 3

Backpropagation: investigations and improvements

3.1 Introduction

In this chapter the backpropagation learning algorithm is described and various studies are made of its performance on the variable difficulty rounding problem. The first part of the investigation involves the observation of *learning curves* and *error maps*, from which it can be seen how the algorithm fares for different system sizes and different difficulties of problem. The influence of the number of hidden units used in the networks is studied in detail for the rounding problem and the scaling behaviour is discussed. The second part of the chapter is concerned with two methods of improving the algorithm. First we introduce a *deformation* procedure, which also is applied to another problem domain to illustrate its generality. Secondly we develop a method for automatically varying the learning parameters during the course of training, to allow a fast but controlled descent.

3.2 The backpropagation learning algorithm

3.2.1 Notation

Since all applications in this thesis use nets of three layers (input, hidden and output), the notation used can be made more simple, and we will speak of the nodes in "the hidden layer" rather than "layer l" for example. This allows the specification of a node state in any layer through use of a different letter rather than subscript, in this way making the equations easier to understand. Table 3.1 shows the notation used in this chapter.

3.2.2 Derivation of the weight changes

The backpropagation algorithm[1] is a method of adjusting the weights in a feed-forward network so that the output pattern, when pattern p is processed through from input to output, is the same as a target pattern, for all patterns p in the training set. The way the pattern is processed from input to output is as follows. The input pattern p is clamped to the input nodes:

$$I_i^p = v_i^p \qquad \forall i, \tag{3.13}$$

these states are then processed to the hidden nodes:

$$\begin{aligned} H_j^p &= \mathrm{FH}_j(\phi_j^p) && (3.14)\\ \text{where} \quad \phi_j^p &= \sum_{i=0}^{N_I} w_{ij}^I I_i^p. && (3.15)\end{aligned}$$

Note that the summation over i runs from 0 to N_I. The unit zero in the input and hidden layers has constant value ($I_0^p = H_0^p = 1, \quad \forall p$) and thus represents a constant bias connecting to each node, irrespective of the pattern being processed. We will sometimes talk of the hidden unit *threshold*, denoted by θ_j^H, and equal to

[1]We shall use the Rumelhart [RHW86] formulation, although other derivations have been independently discovered [Wer74, lC85, Par85]

Symbol	*Meaning*
I	input unit state
H	hidden unit state
O	output unit state
v	input pattern
t	target pattern
$\delta(o)$	error term for an output unit
$\delta(h)$	error term for a hidden unit
ϕ	potential at a unit
FO	response function of an output unit
FH	response function of a hidden unit
subscript on any of above	labels the unit in a layer
superscript on any of above	labels the pattern being processed
subscript of zero	indicates a fixed node giving the threshold values
prime ($'$)	indicates the differential
w_{ij}^{I}	weight from *input* unit i to *hidden* unit j
w_{ij}^{H}	weight from *hidden* unit i to *output* unit j
N_I	number of input nodes
N_H	number of hidden nodes
N_O	number of output nodes

Table 3.1: Notation.

$w^I_{0j}I_0$ (and similarly for the output unit threshold θ^O_j). Finally, the output pattern emerges as some function of the output states:

$$\begin{aligned} O^p_k &= \mathrm{FO}_k(\phi^p_k) && (3.16)\\ \text{where}\quad \phi^p_k &= \sum_{i=0}^{N_H} w^H_{jk}H^p_j. && (3.17)\end{aligned}$$

If we take the output pattern to be a direct mapping of the output states, then we can define a total error (E) between the net's output and the target output as a sum of squared errors at each of the output nodes for each of the patterns:

$$E := \frac{1}{2}\sum_{p=1}^{N_p}\sum_{k=1}^{N_O}(t^p_k - O^p_k)^2. \tag{3.18}$$

Other measures of the error can be defined (see chapter 4), but we use this one here to illustrate the derivation of the weight changes using gradient descent. The backpropagation algorithm for reducing this error by changing the weights uses gradient descent on the surface E in the space in which it is defined (i.e. the space of the network weights), thus:

$$\Delta w \propto -\frac{\partial E}{\partial w}, \tag{3.19}$$

where w represents a general weight anywhere in the system. The constant of proportionality is taken to be η, and is known as the *step-size*, thus:

$$\Delta w = -\eta\frac{\partial E}{\partial w}. \tag{3.20}$$

The error E is an implicit function of all the weights, but the form of Δw for all the weights between the same two layers will be the same. The weight changes for the weights from hidden to output are:

$$\Delta w^H_{lk} = -\frac{\eta}{2}\sum_p\sum_j\frac{\partial(t^p_j - O^p_j)^2}{\partial w^H_{lk}} \tag{3.21}$$

$$= \eta \sum_p \sum_j \frac{\partial O_j^p}{\partial w_{lk}^H}(t_j^p - O_j^p) \tag{3.22}$$

$$= \eta \sum_p \sum_j (t_j^p - O_j^p) \frac{\partial(\sum_i w_{ij}^H H_i^p)}{\partial w_{lk}^H} \mathrm{FO}_j^{p\prime}, \tag{3.23}$$

where $\mathrm{FO}_j^{p\prime} \equiv \mathrm{FO}_j'(\phi_j^p)$, and represents the derivative of the output response function. The partial derivative in (3.23) is clearly zero for all but the $i = l, j = k$ term in the $\{ij\}$ summation, since all the weights vary independently, thus we have:

$$\Delta w_{lk}^H = \eta \sum_p (t_k^p - O_k^p) \mathrm{FO}_k^{p\prime} H_l^p \tag{3.24}$$

$$\Delta w_{lk}^H = \sum_p \eta \delta(o)_k^p H_l^p \tag{3.25}$$

$$\text{where} \quad \delta(o)_k^p := (t_k^p - O_k^p) \mathrm{FO}_k^{p\prime}. \tag{3.26}$$

For the input to hidden weights we have a similar derivation until (3.23):

$$\Delta w_{lk}^I = \eta \sum_p \sum_j (t_j^p - O_j^p) \frac{\partial \left(\sum_i w_{ij}^H H_i^p\right)}{\partial w_{lk}^I} \mathrm{FO}_j^{p\prime}, \tag{3.27}$$

where now the hidden states H are implicit functions of the input to hidden weights, thus:

$$\Delta w_{lk}^I = \eta \sum_p \sum_j (t_j^p - O_j^p) \mathrm{FO}_j^{p\prime} \sum_i w_{ij}^H \mathrm{FH}_i^{p\prime} \frac{\partial \left(\sum_m w_{mi}^I I_m^p\right)}{\partial w_{lk}^I}. \tag{3.28}$$

Again here the only term in the $\{im\}$ summation which survives is the $i = l, m = k$ term:

$$\Delta w_{lk}^I = \eta \sum_p \sum_j (t_j^p - O_j^p) \mathrm{FO}_j^{p\prime} w_{lj}^H \mathrm{FH}_l^{p\prime} I_k^p \tag{3.29}$$

$$= \sum_p \eta \sum_j \delta(o)_j^p w_{lj}^H \mathrm{FH}_l^{p\prime} I_k^p, \tag{3.30}$$

$$\Delta w_{lk}^I = \sum_p \eta \delta(h)_l^p I_k^p \tag{3.31}$$

$$\text{if we define} \quad \delta(h)_l^p := \mathrm{FH}_l^{p\prime} \sum_j \delta(o)_j^p w_{lj}^H. \tag{3.32}$$

So we see from equations (3.25) and (3.31) that all the weight changes are computed from the product of the state of the unit from the which the weight originates, and an error term associated with the unit the weight influences directly, summed over all the patterns. Since the error term for the hidden units involves a weighted summation over the error terms for the output units, this procedure is known as *back propagating the errors*. Notice how the processing of the δ's in the backpropagation phase is almost the same (apart from the nonlinear response function) as the processing of activations in the forward direction.

One further point is that if sigmoidal response functions are being used, then it is not possible for the state of the units to reach the limiting values of zero or one. Thus it is normally decided to consider outputs within some tolerance *tol* of the actual targets as sufficiently well learnt.

The basic algorithm is thus given by equations (3.25) and (3.31). However, a slight modification pointed out in [RHW86] theoretically gives a more stable and faster descent, by adding on a fraction of the last weight change at time $n-1$ in calculating the new weight change at time n:

$$\Delta w(n) = -\eta \frac{\partial E}{\partial w} + \alpha \Delta w(n-1), \tag{3.33}$$

the idea being that if the system is progressing down a long gentle slope the weight changes will be in the same direction and therefore additive, hence speeding up the descent, while if the system is continually crossing from one side of a valley to the other, the weight changes will be damped into an average downward direction. α controls the fraction of the weight to be added on each time, and is known as the "momentum". Its actual usefulness is discussed later.

Thus the learning schedule for a network commences in the following way. The weights are initialized with small random values in order to break the symmetry, as described in [RHW86]. At each epoch of learning the complete set of patterns is presented to the network and the gradients for each weight accumulated, using the

Input	Output
$[0.5 + r, 1.0]$	1.0
$[0.0, 0.5 - r]$	0.0

Table 3.2: The mapping required to be learnt between an input unit and its corresponding output unit. The parameter r defines the difficulty of the task.

backpropagation procedure. After all the patterns have been presented the weights are changed according to equation (3.33). This is continued until the output values for each pattern are within tolerance for each pattern, or learning is abandoned due to the network getting stuck in a *local minimum* (see section 3.5.3). There are other schemes for updating the weights, which approximate to the gradient descent procedure for small values of the step-size, but we shall not consider them here. The type of updating scheme described above is known as "batch learning" [Wal87a].

3.3 The rounding problem and its training set

The learning algorithm was applied to the following task. The network is required, when trained, to be able to round-off a set of numbers applied to its input to zeroes and ones at the output. There is a corresponding output unit for each input unit. The numbers applied to the input lie in the interval [0,1], and outside the range $(0.5 - r, 0.5 + r)$, where r is the parameter defining the *difficulty* of the task. Thus the problem domain to be learnt consists of the mappings in table 3.2 for a certain task difficulty r.

Thus the network can be trained to perform tasks which require differing levels of discernment. The easiest task takes the form of a one-to-one mapping of binary patterns input to output ($r = 0.5$), and the greater difficulties are found when numbers either side of 0.5 are very similar and yet have to be mapped to different extremes. So the closer r is to zero, the harder it should be for the network to adjust its weights to achieve the required function.

Since the nature of this problem is such that each element forming an input picture is totally independent of the other elements (i.e. this is an order 1 problem, in the terminology of Minsky and Papert [MP69]), the elements in the output picture should correspondingly be independent. The only dependence between input and output is between elements corresponding to the same positions in input and output pictures. With this restriction it is clear that the network should tend to alter its weights such that it forms *large* weights for non-intersecting routes from the input elements through the hidden layer to the corresponding output elements, and *negligible* weights for the weight paths which would interfere with these routes.

This problem is linearly separable, so it is not necessary to use a hidden layer to perform the mapping. However, for the purposes of illustrating the performance of the backpropagation algorithm in learning the mappings when there are hidden units present, since in general they will be necessary, we perform these experiments solely on networks with one hidden layer. Given that hidden units are present, we must also ensure that we have at least as many hidden units as input/output pairs. The reason for this is that *as* $r \rightarrow 0$, the numbers leading from the input nodes become extremely small and so unless there exist paths which are independent of the other input values, the output will not be able to discriminate between $0.5 + r$ and $0.5 - r$ values at the input. The input layer contained up to seven units, the output layer had the same number of units as the input layer, and the hidden layer could contain up to 25 units. If there are more hidden units in the hidden layer than are *required* to find a solution, then there are expected to be a larger number of possible solutions, and one of these will have to be chosen by the system. The choice can depend only on the initial random weights. Thus the system can descend into different global minima of the error surface, by starting off at different points on the surface. The spectrum of global minima includes solutions where routes between input/output pair involve varying numbers of hidden units, and also the cases where some hidden units are not used at all. We are not concerned with whether the network can generalize on this problem, although a form of simple generalization is possible, which is instructive to examine, before more complicated generalizations are examined in later chapters. It can be demonstrated using the decision regions mentioned in chapter 2. If we take a network with two input nodes (and thus two output nodes), and draw

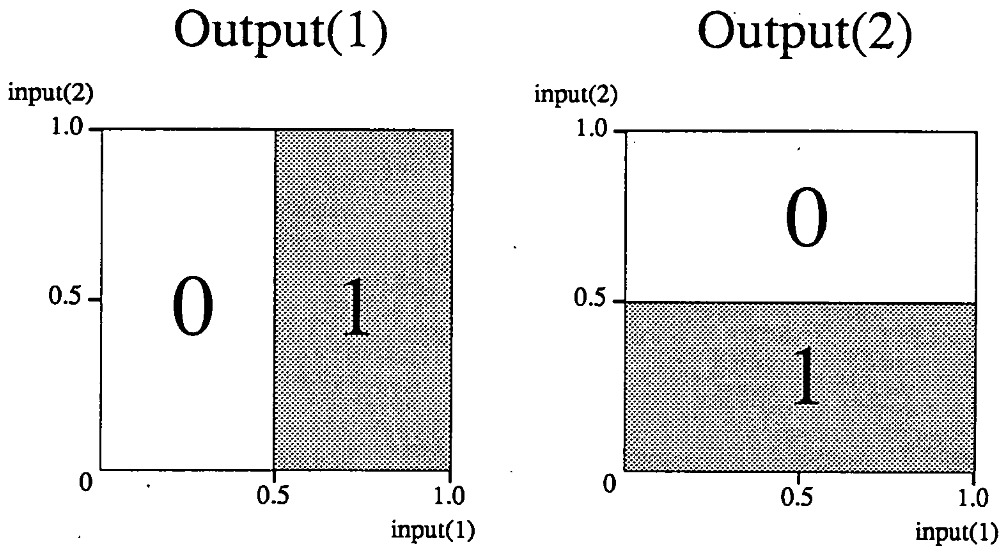


Figure 3.1: The two types of decision regions for output nodes 1 and 2, when there are two input nodes input(1) and input(2), in the rounding problem.

the desired decision regions in figure 3.1 for each of the output nodes, we notice that the decisions of the respective output nodes (we consider them to be binary classifiers) require an independence of the other input node, and a sharp dividing line at the value 0.5 on the relevant input unit. For any particular difficulty of problem if all sample points have been learnt then the decision line will exist in the region constrained by the learnt points on either side of the perfect decision line. Figure 3.2 shows the types of lines which may arise after the points shown have been learnt. It may be that the perfect line is found straightaway, although this is unlikely, nevertheless it is certain that some points either side of the line will be mapped very well, thanks to the learning of particular hard numbers in the training set, and so generalization to a larger or smaller extent can be achieved. It can be seen that for certain values of input(1) the decision line crosses the perfect decision line, which may also be regarded as some form of (weak) generalization. Thus generalization is limited by, but maybe better than, the hardest examples in the training set.

The simple preliminary study above actually gives some guidance on what patterns to include in the training set for the most efficient weight changes and economical

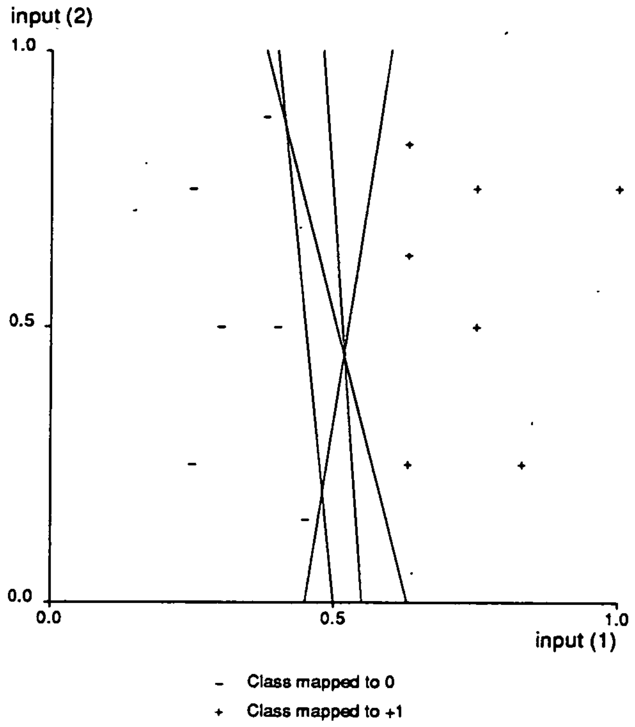


Figure 3.2: Possible decision lines which output(1) node may form on having successfully learnt the data points shown, which indicate the inputs in terms of the point (input(1), input(2)).

pattern numbers. Clearly we should organize the patterns such that the two basic properties of the decision lines are obvious: firstly, independence of decisions with the other input units (i.e. those not corresponding to the output in question), and secondly, inclusion only of the hardest examples for a particular problem difficulty. The task for a particular range r is that the network should learn to round off all numbers (N) in the range

$$(0.5 + r) \leq N \leq 1.0 \quad \text{and} \quad 0.0 \leq N \leq (0.5 - r) . \tag{3.34}$$

In order to ensure this then, writing $R_+ = (0.5 + r)$, and $R_- = (0.5 - r)$, the pictures at the input units are taken to be all the permutations of 0, 1, R_+ and R_-, with R's only present at one of the inputs per picture. An example of this is given in table 3.3. It can be seen that such a training set satisfies the conditions mentioned above, for optimal constraining of the decision boundary. Notice too that for c input units the number of pictures required (for the guarantee of reproducing the whole set of possible inputs and their combinations correctly) will be

unit 1	unit 2	unit 3
R_+	1.0	1.0
R_-	1.0	1.0
R_+	0.0	1.0
R_-	0.0	1.0
R_+	1.0	0.0
R_-	1.0	0.0
R_+	0.0	0.0
R_-	0.0	0.0

Table 3.3: Part of the training set for a three input unit network. The rest of the set is obtained by permutations of the columns.

$$N(c) = c2^c, \tag{3.35}$$

thus the number of pictures required in the training set scales worse than exponentially with the number of input units. Such problems with scaling are not surprising. Although, as mentioned above, this an order 1 problem, the strong independence of the input elements can only be ensured by at least this set of patterns. These patterns define the boundary exemplars [AT88] of the training set. In fact, the main task asked of the network, is to learn the independence in the input/output pairs.

3.4 Unit response functions

The response functions FO and FH are normally taken to be the same for all units (although this is not necessary). As mentioned in chapter 2, the sigmoid function is normally used, because of its properties of being similar in approximation to the perceptron step nonlinearity, and confining the response to lie in a fixed range. The form of the sigmoid function is given in equation (2.9), which we rewrite here for convenience:

$$y = \frac{1}{1 + e^{-\beta x}}. \tag{3.36}$$

The differential of the sigmoid function is

$$\frac{dy}{dx} = y' = -\beta\frac{e^{-\beta x}}{(1+e^{-\beta x})^2} \tag{3.37}$$

$$= -\beta y(y-1), \tag{3.38}$$

and so we see also that another nice feature of this function is that the differential involves only the value of the function and not its argument, which is convenient for efficient computer implementation.

So it seems that the sigmoid function would be an appropriate one to choose for the node responses. However, there is still the question as to whether the nodes should be confined to positive states in the range [0, 1], or allowed to use the entire range [−1, 1]. This may depend on the type of problem studied, but we shall demonstrate how in the problem studied in this chapter the full range is the more appropriate.

Note first that the only state ranges which are important are the ones in use by the hidden units, since the input units have their values clamped by the input patterns and the output units can be converted easily to any range using a linear post-processing stage. The range of values which can be adopted by hidden units will in general affect the size of the space which can be used to represent the patterns (chapter 5 discusses this concept more fully).

We can understand the effect of using the ranges [−1, 1] and [0, 1] in the hidden layer of units by considering the effect on an output unit as the values of the inputs are varied. We will discover that the maximum difference to be obtained is about a factor of two in the learning speed, given the most suitable application. The [0, 1] range can only cause positive state values to be transmitted down the weights, while the [−1, 1] range allows the full range of positive and negative values. But the value of an output unit is given by:

$$O_i = \mathrm{FO}\,(\phi_i) \tag{3.39}$$

$$= y(\sum_{j=1}^{N_H} w_{ij}^H \mathrm{FH}_j + \theta_i). \tag{3.40}$$

The quantity in parentheses has the same range ($\mathcal{R}_1$) whatever the range of values FH_j can adopt, because the parameters w_{ij}^H and θ_i are able to take on all real values. The only benefit to be derived from allowing the FH_j to be of the form $2y_j - 1$ is to provide it with a symmetry about zero, thus enabling the network to learn automatically half the mappings required (in the optimal case of the rounding problem),[4] or equivalently, if all mappings are in the training set, provide a reinforced update for the weights by summing the reinforcing weight changes. This doubles the learning speed which would be achieved using the $[0,1]$ hidden unit ranges.

For a network with the same number of hidden nodes as input and output nodes, then if we assume that the optimal solution (consisting of "paths" from input node to corresponding output node) is reached, the inputs ϕ to a hidden node will be opposite in sign for values of the input node at the top of the path symmetrically about 0.5, and in order for the output unit at the end of the path to respond correctly, the weight from hidden unit j to output unit i and the threshold of the output unit must satisfy:

$$w_{ij}^{H}\ \mathrm{FH}_j(\phi_j) + \theta_i = -(w_{ij}^{H}\ \mathrm{FH}_j(-\phi_j) + \theta_i) \tag{3.41}$$

Thus for the $[-1,1]$ response function ($\mathrm{FH} = 2y - 1$) this requires that

$$\theta_i = 0 \quad \forall i, \tag{3.42}$$

while for the $[0,1]$ response function ($\mathrm{FH} = y$) it requires that

$$w_{ij}^{H} = -2\theta_i \quad \forall i, \tag{3.43}$$

where w_{ij}^{H} represents the large weight making the path from hidden unit to output unit. In both the derivations above we assume that all the other weights not involved in the paths are zero.

The actual effect of this requirement can be demonstrated if we monitor the values of the quantities w_{ij}^{H} and θ_i above, when the two different response functions are

[4]This reasoning assumes the "heavy route" solution is the one which is always found.

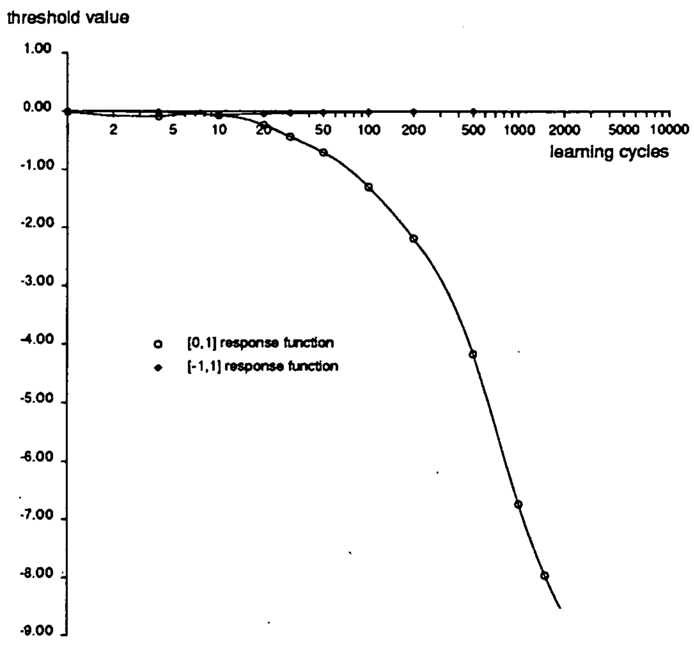


Figure 3.3: Thresholds as function of learning cycle for a 3–3–3 network, $r = 0.01$.

used. Figure 3.3 shows how the threshold θ is much more stable in the $[-1,1]$ case, hardly moving from its optimal value of zero, while for the $[0,1]$ case the threshold increases negatively all the time, being forced to follow the value of the large weight making the path (as required in (3.43)). This is undesirable since the dependence of the two quantities upon one another will make the descent less stable (because both the variables are iterating to values which depend on each other). Also in figure 3.4 we see that the learning speed is approximately double for the $[-1,1]$ case all the time (note the logarithmic scaling on the abscissa).

The response functions used therefore in this chapter are:

$$\mathrm{FO} = \frac{1}{1+e^{-\beta\phi}} \tag{3.44}$$

and

$$\mathrm{FH} = \frac{2}{1+e^{-\beta\phi}} - 1, \tag{3.45}$$

which give weight changes specified by the following δ's (from equations (3.26) and (3.32)):

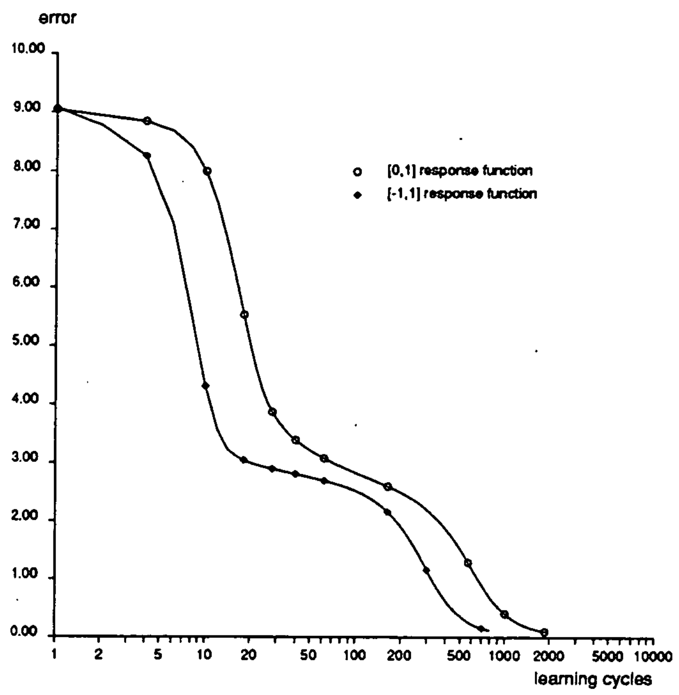


Figure 3.4: Learning curves for the 3–3–3 network, learning down to range $r = 0.01$.

$$\delta(o)^p_k = \beta(t^p_k - O^p_k)O^p_k(1 - O^p_k) \tag{3.46}$$

$$\tag{3.47}$$

$$\delta(h)^p_l = \frac{\beta}{2}(H^p_l + 1)(1 - H^p_l)\sum_j \delta(o)^p_j w^H_{lj}. \tag{3.48}$$

These δ's can then be substituted in equations (3.25) and (3.31) to get the weight changes.

3.5 Evaluation of the basic algorithm's performance

3.5.1 The success rate of the basic algorithm

A further note about the notation: N_1–N_2–N_3 denotes a network of N_1 input units, N_2 hidden units and N_3 output units.

r	*% nodes correct*	*% patterns correct*
0.5	100	100
0.1	100	100
0.01	88.8	43.8
0.008	89.0	45.0
0.006	83.0	15.0
0.004	77.3	9.38
0.002	77.1	11.3
0.001	80.0	0.00
0.0001	80.0	0.00

Table 3.4: Comparison of the percentage of output nodes and patterns correct, for the system 5–5–5, using the basic algorithm, for various difficulties r.

The performance of the algorithm in learning different levels of problem difficulty, for different sizes of network, is summarized in figure 3.5 and table 3.4, where we have used networks with just the necessary number of hidden units to perform the mapping. Figure 3.5 shows how the performance of the algorithm, monitored by the total number of output nodes correct,[5] decreases as the problem difficulty is increased (represented by a decreasing r), for all the system sizes, there being a sharp change in the number of nodes correct at a particular value of r (usually somewhere about 0.002, but it can be seen that it was sooner for the largest system. In table 3.4, we compare the percentage of output nodes correct with the corresponding percentage of *complete patterns* which were correct, for the 5–5–5 network. Note how the network can get all the patterns wrong although most of the nodes are actually correct. Closer examination of these numbers suggests that most if not all the incorrect patterns were actually a result of the nodes responsible being "flipped", causing the outcome that nodes are mapped either very well or very badly.

The nature of the learning can be studied using the *learning curves* — plots of the progress of the total error at the output units as a function of the training cycle. This is shown for systems 2–2–2, 3–3–3 and 4–4–4 in figure 3.6.

[5]If the network was having difficulty learning, i.e. it was in a *local minimum* (see section 3.5.3), the learning was terminated.

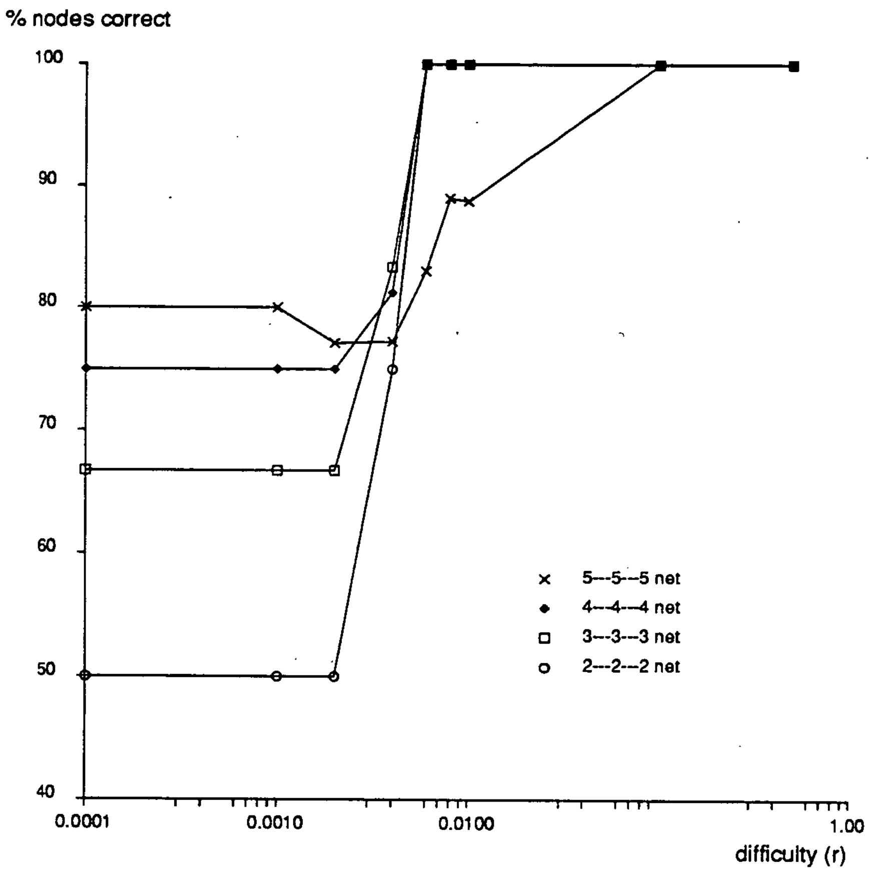


Figure 3.5: Performance of the algorithm for various problem sizes and difficulties.

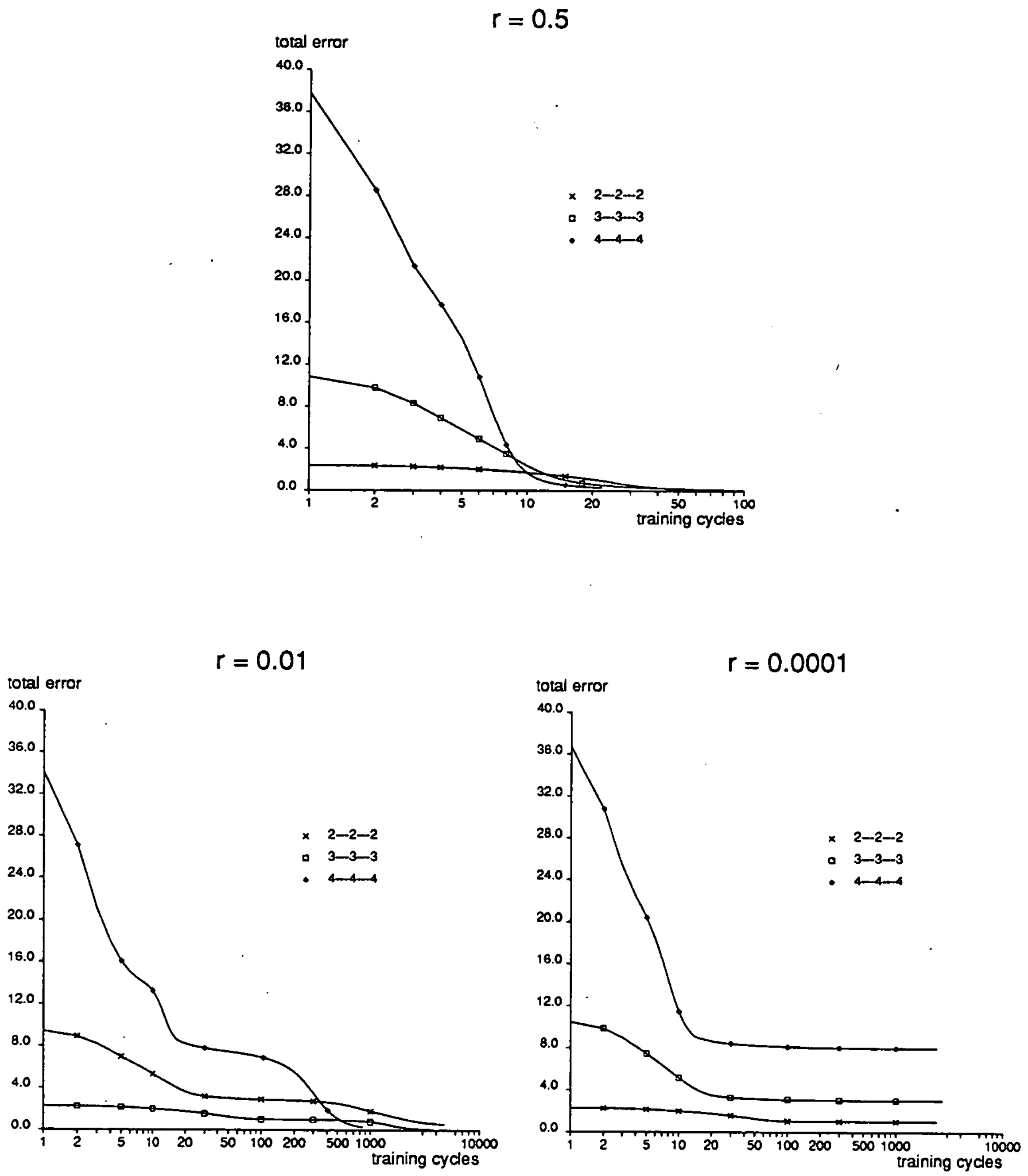


Figure 3.6: Learning curves for the basic algorithm at various net sizes and problem difficulties.

On each graph is plotted the progress of total error with training cycle for each system for a certain difficulty r (0.5, 0.01, and 0.0001), with $\eta = 0.1$ and $\alpha = 0.6$. It can be seen how, for a difficulty of 0.5, all the systems manage to locate a global minimum within a reasonable period. The descent is marked in all the systems by a relatively steep descent for the first 10 to 100 epochs, followed by a region of low gradient until the end. For the second difficulty, 0.01, only two systems manage to locate a global minimum. The descent is marked again by a steep fall in error during the first 10 to 100 epochs, but this time the almost level descent which follows is terminated by another relatively steep drop at 100 to 1000 epochs. The 4–4–4 curve is characterized by a rapid drop at the end, indicating the location of a sudden steeper descent, leading ultimately to a solution. It can be seen that the 3–3–3 system, however, does not locate a similar feature, and is destined to remain stuck on a plateau-like surface. With the third difficulty (0.0001) none of the systems manage to find a solution. The relatively steep initial descents are terminated at 10 to 100 epochs by a very flat portion, which shows no sign of ending.

Finally, from figure 3.7 the speed with which the algorithm finds a solution starting at various (random) points on the error surface can be seen to centre quite closely about 250 cycles for the majority of the runs, although there are a number of runs (14%) which get stuck in local minima and fail to find solutions at all, and also a number that take much longer times, and do not form part of the main distribution. These runs fell victim to the "crack" problem, discussed in section 3.5.5. In general, however, if a solution is going to be found quickly, the point in weight space at which the iteration is started can be expected to cause about a 40 – 50% difference in run time.

3.5.2 Error maps in weight space

A useful, but limited, probe into the terrain of the error surface is the error map, or two-dimensional *cross-section* of the error surface in weight space. The technique (described in [PNH86]) is to work out the current weight-change vector $\Delta \mathbf{w}$, and, from this the unit vector $\frac{\Delta \mathbf{w}}{||\Delta \mathbf{w}||}$ giving the direction in weight space of the system's next step. The error is then plotted in an appropriate range of values of step size

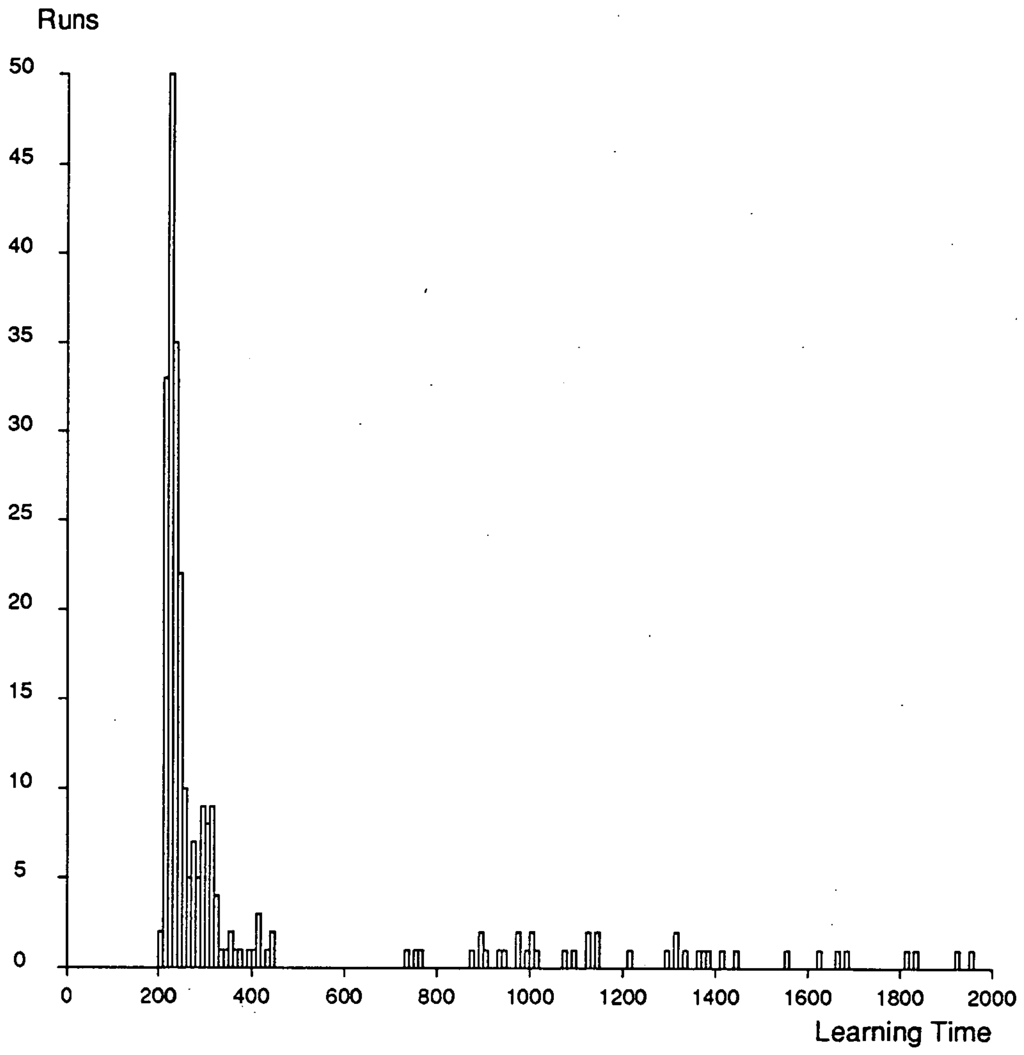


Figure 3.7: Histogram showing the distribution of learning times for a 4–4–4 network at a difficulty $r = 0.01$, for the basic algorithm.

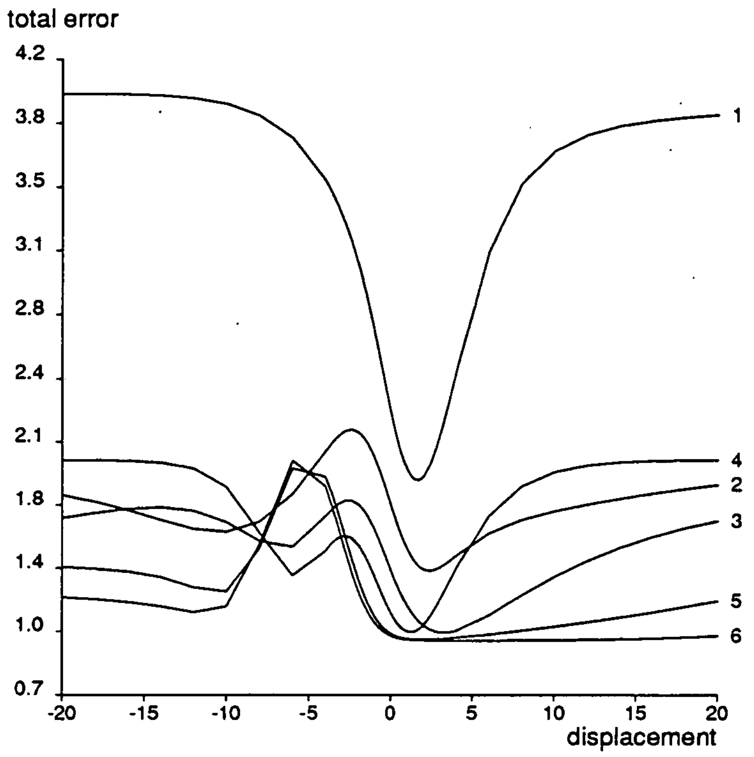


Figure 3.8: Descent into a local minimum.

about the system's current position (say $[-2\|\Delta\mathbf{w}\|, 2\|\Delta\mathbf{w}\|]$), to give an idea of the kind of terrain down which the system is progressing.

On some of these graphs an asterisk indicates the present position of the system, and a vertical line indicates the destination point.

3.5.3 Local minima

When the learning curve remains essentially flat for a relatively long time (compared to the rest of the descent) it is assumed that the algorithm is unable to converge to a global minimum, and has settled into a local minimum. It is not easy to "escape" from such local minima by taking perhaps a large step in a random direction, since if a lower basin exists somewhere else, the chances of reaching it in this manner are very slim, and especially as the dimension of weight space increases, the time needed for a reasonably thorough search is prohibitive.

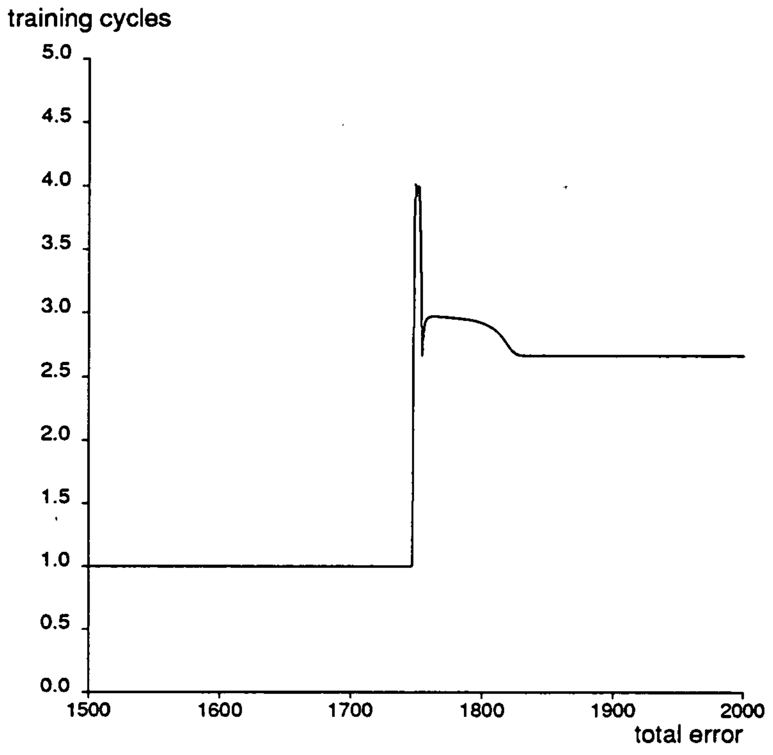


Figure 3.9: Attempted "escape" from a local minimum at cycle 1750.

As an example, consider figure 3.8, which shows a 2–2–2 system learning a mapping of difficulty 0.0001. The initial descent, shown in figure 3.6, is reasonably quick, but before long the error becomes quite stationary. The graph maps the terrain as the system clearly begins to iterate to the bottom of a local minimum. Imagine the system at the point on the curve given by zero displacement, and the rest of the curve as being the surrounding terrain in the direction in which the system is about to move. We can see how as the system settles into the local minimum the steepest gradient decreases, and the minimum is characterized by a steep wall in one direction and a plane in the other. It is possible to make the system climb out of this by giving it a large step size. However there is no reason why the direction it takes (current steepest *descent* in this case) should be one which brings the system to the brow of a hill, indeed the normal scenario is for the system to climb up to another local minimum, or a plateau, and stay there, as in figure 3.9 (a large increase in the step for one cycle at 1750 cycles brought the system out of the current minimum, but unfortunately left it on a higher one).

Why should it be the case that a local minimum has such a shape described above rather than the more intuitive "basin" shape? The reason is that the response functions used by (in particular) the output units are of such a form that if the potential becomes greater than an upper limit, or lower than a lower limit, the actual state of the unit changes very little. With semi-linear threshold functions there is an explicit cut-off when the state of the unit reaches one or zero, such that it remains there should the potential become higher or lower respectively, and it might be expected that the very slowly sloping plateaus here would be perfectly flat in such a case. Thus the contrasting flatness in one direction and steepness in the opposite direction indicate that all the output units are bound very closely to zero or one for each pattern, and increasing all the weights in the current direction will merely serve to push the units closer to the extreme values of the nodes at the output (and possibly at the hidden layer too). If the weights are increased in the opposite direction the opposite should happen, with some of the output units being pulled back towards mid-range values.

Now we are in a position to explain the shape of at least the last two curves in figure 3.8. The height of the flat portion is given by the number of output nodes which are bound to the *wrong* extreme, the gradient of the flat portion has already been explained as a result of the low gradients at the top and bottom of the sigmoid function, and the mountainous region to the left is characterized by an overall increase in error due to the larger number of otherwise correctly mapped outputs being moved to intermediate values countering the beneficial movement away from the extremes of the smaller number of incorrectly mapped outputs. As we move further away from the current position of the system the error falls again, as some outputs again become correctly mapped, while others get pushed to incorrect extremes. It could be that the error increases overall; this depends on the actual weight changes. Finally, at far left we see another flat portion again being located. Figure 3.9 can also be explained in this way. The large step size given to the system pushes it onto a plateau because it was large enough not only to change the weights sufficiently to leave the current local minimum, but also to locate another area of weight space characterized by outputs being bound to extreme values, this time with more outputs wrong than before.

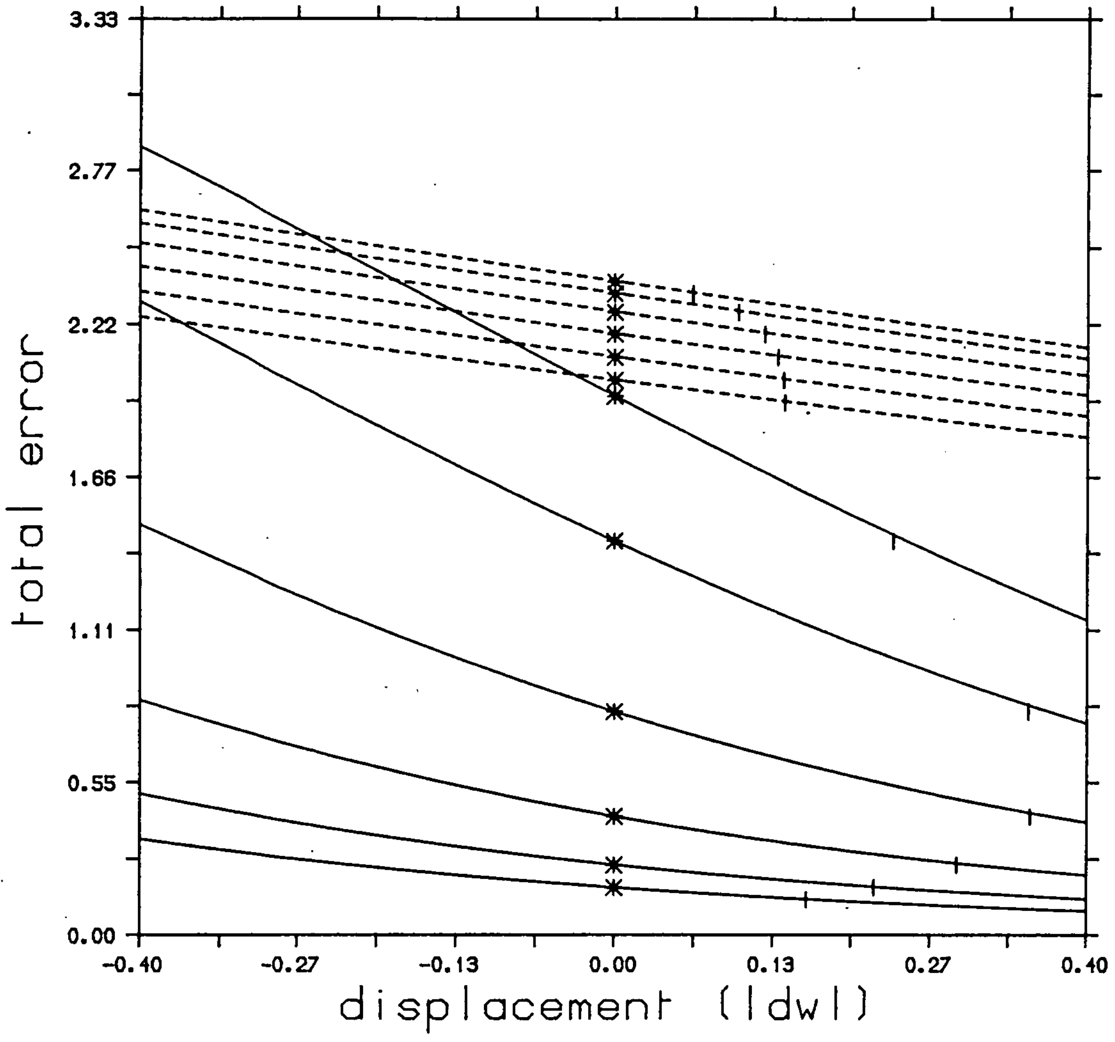


Figure 3.10: Comparison of the initial gradients for the 2-2-2 and the 2-25-2 systems, at $r = 0.5$.

3.5.4 The influence of hidden units

Gradient of early descent

The descent for the simplest system (2–N–2 with $r = 0.5$) is shown for the first six training cycles in figure 3.10. The terrain for the system with two hidden units is compared with that for the system with 25 hidden units. It can be seen how much steeper the descent is when there are a large number of hidden units.

Observation of gradients for the 2–N–2 system indicated that the gradient increases uniformly with hidden unit number. To obtain an approximate scaling law, assume that all the weights in a network are of equal importance in the early stages of learning, so that each weight is made to change in the learning algorithm such as to reduce the error, by about the same magnitude δp, and if this change is small, then since

$$\frac{\partial E}{\partial w_{ij}} \propto \|\delta \mathbf{w}_{ij}\|, \tag{3.49}$$

and the gradient is given by

$$\left\|\frac{dE}{d\mathbf{w}}\right\| = \left\|\sum_{ij} \frac{\partial E}{\partial w_{ij}} \delta \mathbf{w}_{ij}\right\| \tag{3.50}$$

we have

$$\left\|\frac{dE}{d\mathbf{w}}\right\| \propto \left\|\sum_{ij} \delta p \; \delta \mathbf{w}_{ij}\right\|. \tag{3.51}$$

But if $\delta \mathbf{w}_{ij} = \delta p \; \hat{\mathbf{e}}_{ij}$, where $\hat{\mathbf{e}}$ is a unit vector in the direction $\mathbf{w}_{ij}$, so if there are N_W weights in the system,

$$\begin{aligned}
\left\|\frac{dE}{d\mathbf{w}}\right\| &\propto \left\{\left(\sum_{ij} (\delta p)^2 \hat{\mathbf{e}}_{ij}\right) \cdot \left(\sum_{ij} (\delta p)^2 \hat{\mathbf{e}}_{ij}\right)\right\}^{\frac{1}{2}} && (3.52) \\
&\propto (\delta p)^2 (N_W)^{\frac{1}{2}} && (3.53)
\end{aligned}$$

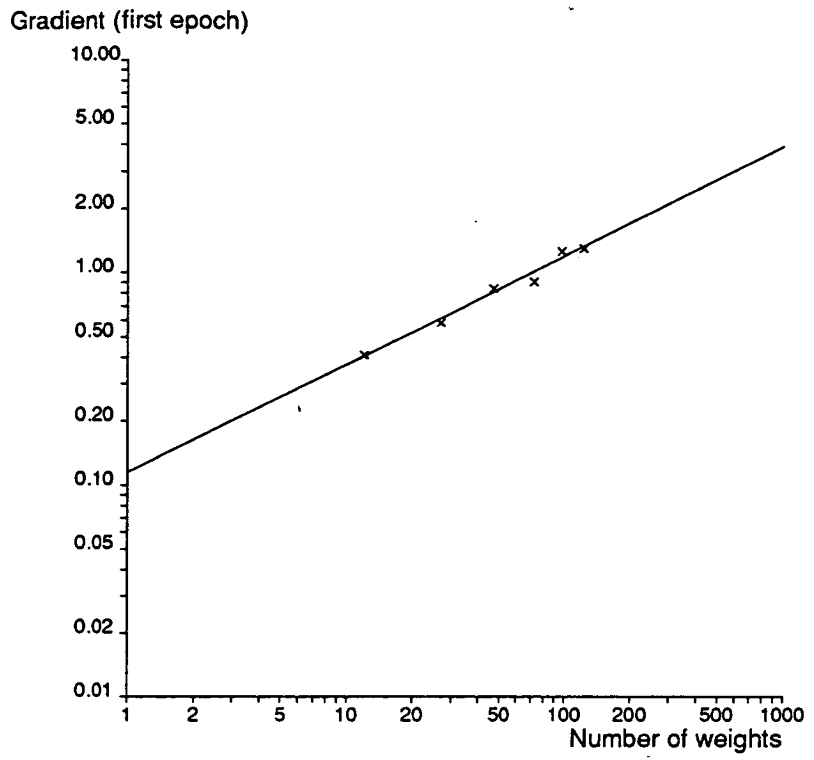
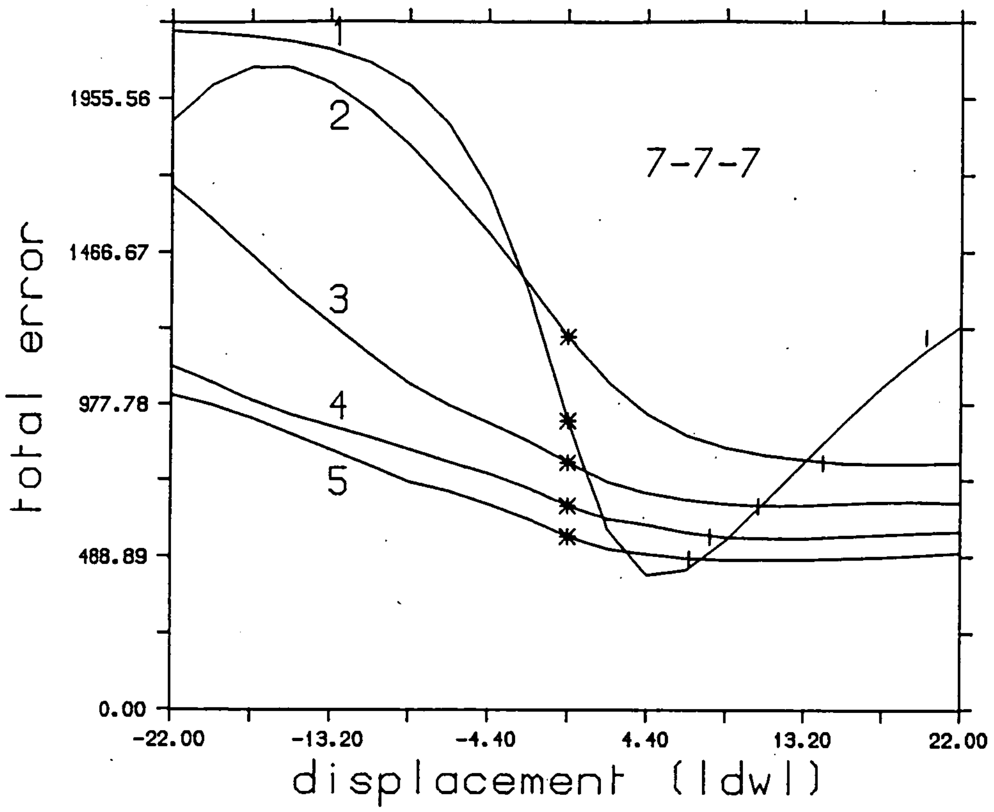


Figure 3.11: Scaling of the gradient with the number of weights in a network. Note the logarithmic scaling of the axes.

since the $\hat{e}_{ij}$ are orthogonal. Thus we should expect the gradients to scale linearly as a function of the square root of the number of weights used in the system. In figure 3.11 we plot the logarithm of the gradients at $r = 0.01$ against the logarithm of the number of weights in the networks. A least-squared fit to these points produces a line of gradient 0.51 ± 0.04, in accordance with the simple argument above.

Figure 3.12 shows the same situation as figure 3.10 for a larger network. However, here the surface is so much more mountainous anyway that the beneficial effect of the extra hidden units is best seen by noticing the cliff-like terrain of the 7–25–7 network, as opposed to the valley-like terrain of the 7–7–7 network. The cliff-like descent is much quicker and more penetrating.

The addition of hidden units clearly consistently increases the size of the gradient, together with stability of the descent, and thus seems to have a consistently beneficial effect (although it must be remembered that the actual computing time increases as we increase the hidden units in the network).



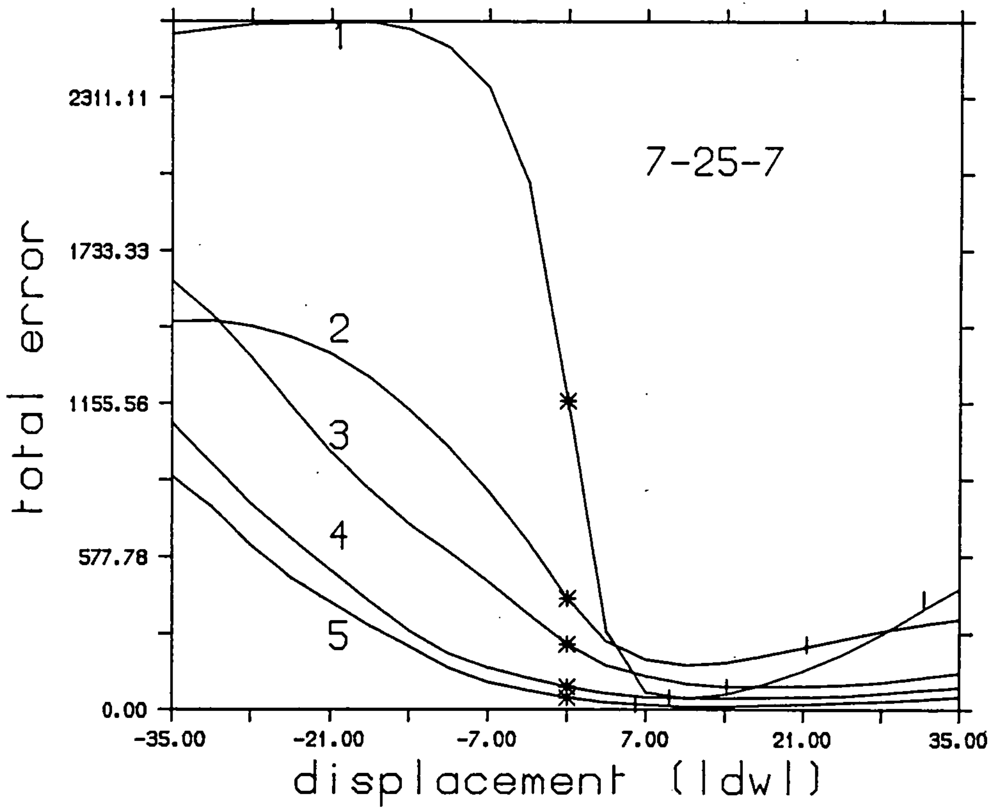


Figure 3.12: Comparison of the initial error surface for the 7–7–7 and 7–25–7 systems, for $r = 0.5$.

Redundancy

With the addition of more hidden units, it becomes difficult to analyze the patterns of heavily weighted routes from input to output, since this typically includes more than one path for each input/output pair. Thus it is useful to represent the system graphically, to provide an indication of the routes taken. The intensities were normalized to the weight with the greatest (absolute) value. Weights of negligible size compared with the larger weights have negligible intensities. Analyses of various sizes of system showed that one frequently obtained hidden units with negligible weights to and from all output and input units. An example is shown in figure 3.13, a graphics screen dump of a 5–15–5 network which has learnt down to a range[6] of 10^{-4}. Similar patterns are observed in other networks with large numbers of hidden units. It appeared that such occurrences were the results of competition between two or more input/output routes of similar strength resulting in a draw, with the weights concerned subsequently becoming negligible compared with weights in other routes, and the routes themselves thenceforth abandoned.

Scaling of learning time

In figure 3.14 we observe the effect of extra hidden units on the learning, at difficulty 0.5, for different network sizes. The graphs show the number of epochs to solution for each system size, averaged over 50 – 100 runs with different random starts. The error bars give some idea of the variation in learning time depending on a particular starting point on the error surface. It can be seen that for each of the networks shown, there is a definite trend for a quicker descent as the number of extra hidden units increases. Also there is possibly a trend for the addition of one or two hidden units producing a more dramatic effect as the network size increases. In all cases the addition of more hidden units has less effect as the total number of units in the hidden layer increases.

From the results of section 3.5.4 it might be expected that the scaling has some kind of dependency on a power of the number of weights in the network, or

[6]Learning down to such a low range for a network of this size was only possible after the improvements described in sections 3.6 and 3.7 were implemented.

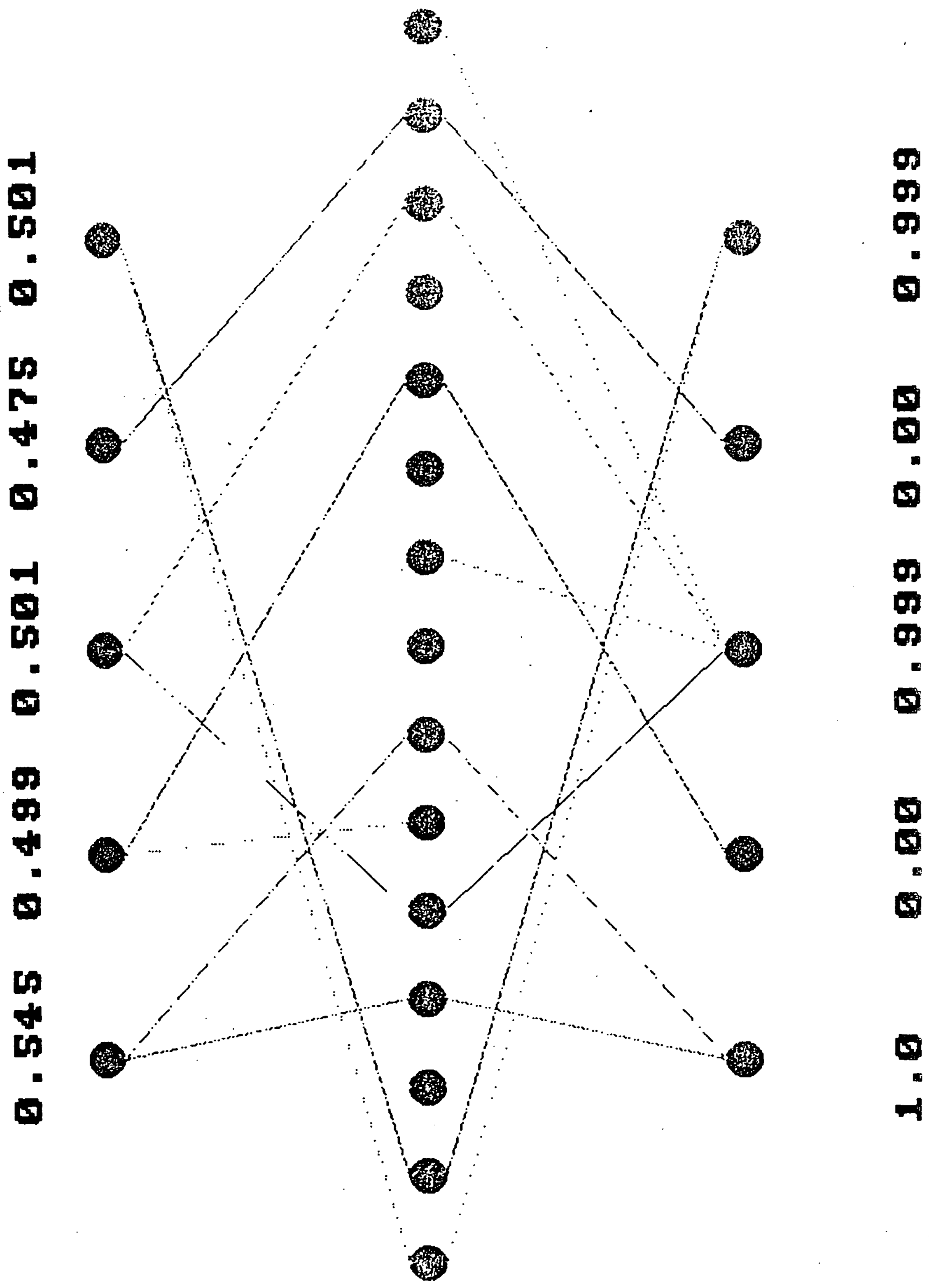


Figure 3.13: Graphics screen dump of the network (redundancy).

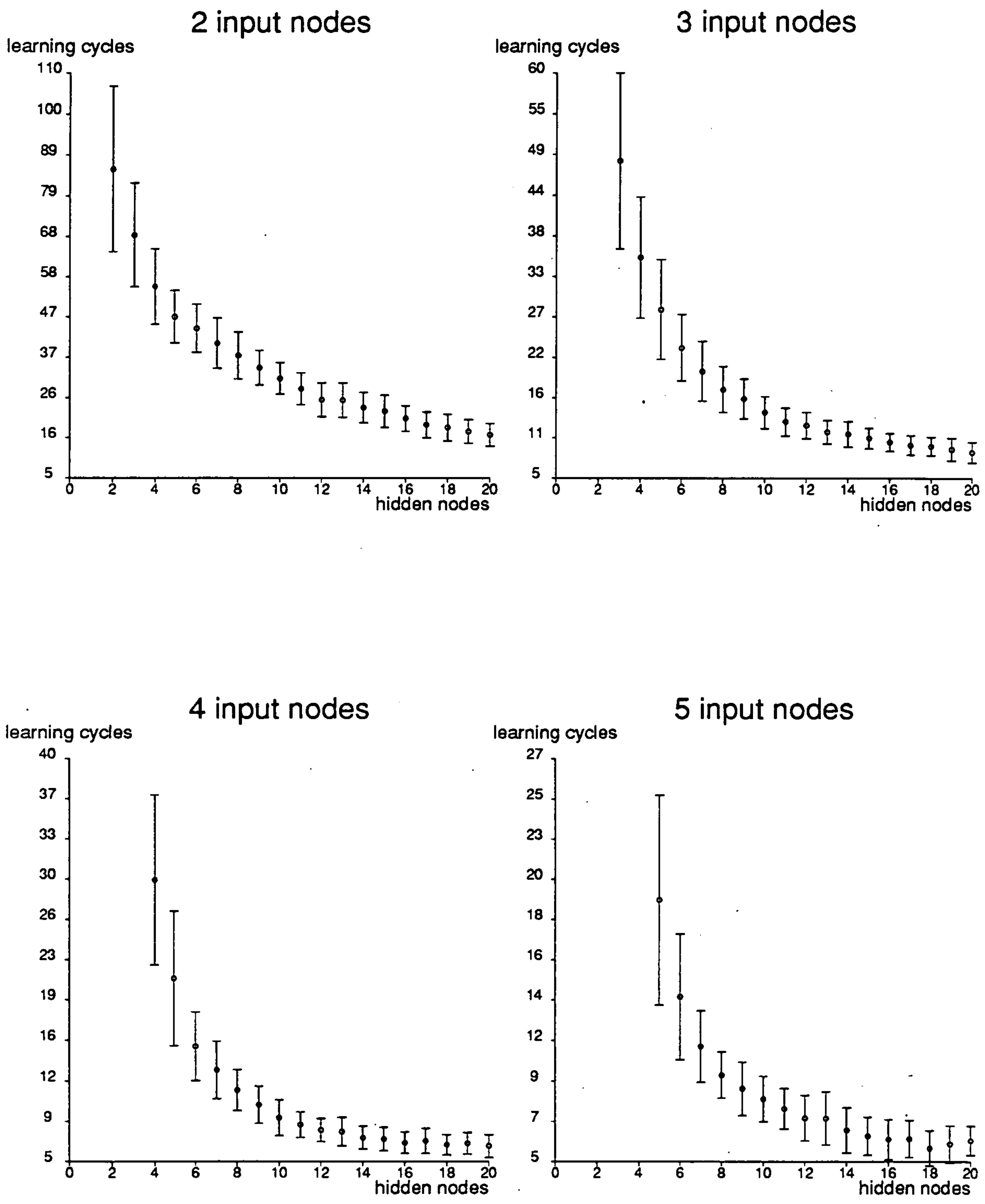


Figure 3.14: Scaling of learning times for $r = 0.5$ with the number of hidden units.

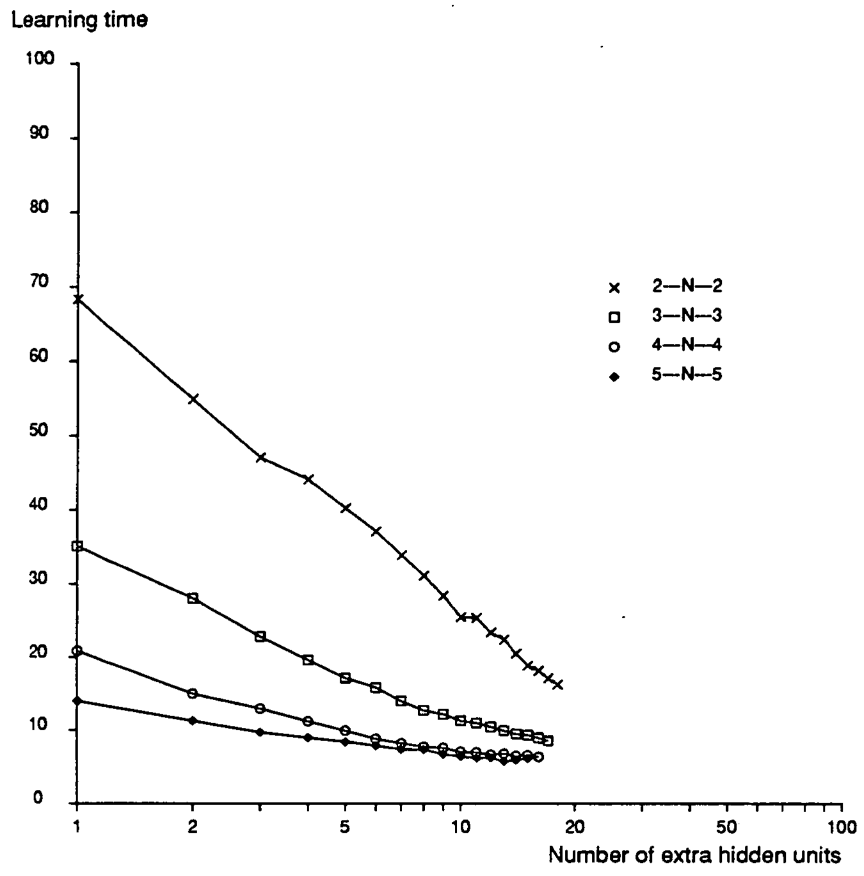


Figure 3.15: Scaling of learning time versus logarithm of extra hidden units.

more specifically, on the number of *extra* hidden nodes. By *extra* is meant the number over and above the number required to solve the problem, i.e. $N_H - N_I$. If we assume a power law, then plotting log(learning time) vs. $\log(N_H - N_I)$ should give a straight line of gradient given by the power. However, due to the rapid levelling off at higher numbers of hidden units, it is more likely that it is a logarithmic relationship. In figure 3.15 is plotted the log of the number of *extra* hidden units along the abscissa, and the training time as before on the ordinate. The first portions of the lines show approximate agreement with the scaling relation $T \propto \log(N_H - N_I)$, with the constant of proportionality being a function of the number of input units (note how the gradient of these lines decreases with the number of input units). The latter portions of the lines veer away from the logarithmic dependency. This is suspected to be due in part to the redundancy effect described in the last section: as the number of extra hidden nodes increases, the probability of more of them being redundant increases, and so the learning time will not decrease so rapidly with increasing hidden units, or ultimately produce a levelling off. It is also possible that yet more hidden units may *increase* learning time, because the node redundancy is a result of competition, and this competition may slow down the algorithm's descent.

3.5.5 The danger of α

The influence of the momentum parameter (α) is sometimes very important for finding a solution. For example, it was pointed out in [PNH86] how an initially large value of the momentum parameter can cause unstable descent, due to the large weight changes this causes at early stages, when the error surface is steep.

For instance, on a run with a 7–25–7 system with zero momentum a solution was reached after only 6 training cycles ($r = 0.5$), whereas when the system was trained with a momentum of 0.6 a solution was not found until more than 50 epochs. The descent during the initial few epochs was quite similar for the two cases, however, at an error of about 3.5 the second system landed on a plateau. The following 40 or so epochs were taken up by slow progress along this plateau, until the cliff-like edge was found. This descent was interesting, and so the surrounding terrain for the relevant epochs was mapped out in figure 3.16. The graphs show how flat the plateau is and how steep the plunge at the end is. Following the sequence of graphs from left to right, top to bottom, it can be seen how the system slowly moves toward the cliff-like drop (slowly because the gradient is small) and rolls down the cliff quickly. This can be explained in the same vein as the graphs in section 3.5.3. The flat portion this time however has an end in sight, but the surprise is how sudden the drop in error is. The actual drop is approximately 3.5, and if we assume that this change in error is due entirely to incorrect outputs being suddenly switched from the wrong extreme to the right one, then each incorrect output would contribute an error of 1/2 to the overall error, and so all 7 incorrect outputs are suddenly rectified (the rest of the descent merely involves minor improvements). The fact that they are all rectified together implies that the source of the error was a single weight to which a particular output node was very sensitive, and happened to result in all mappings of a particular type being wrong.

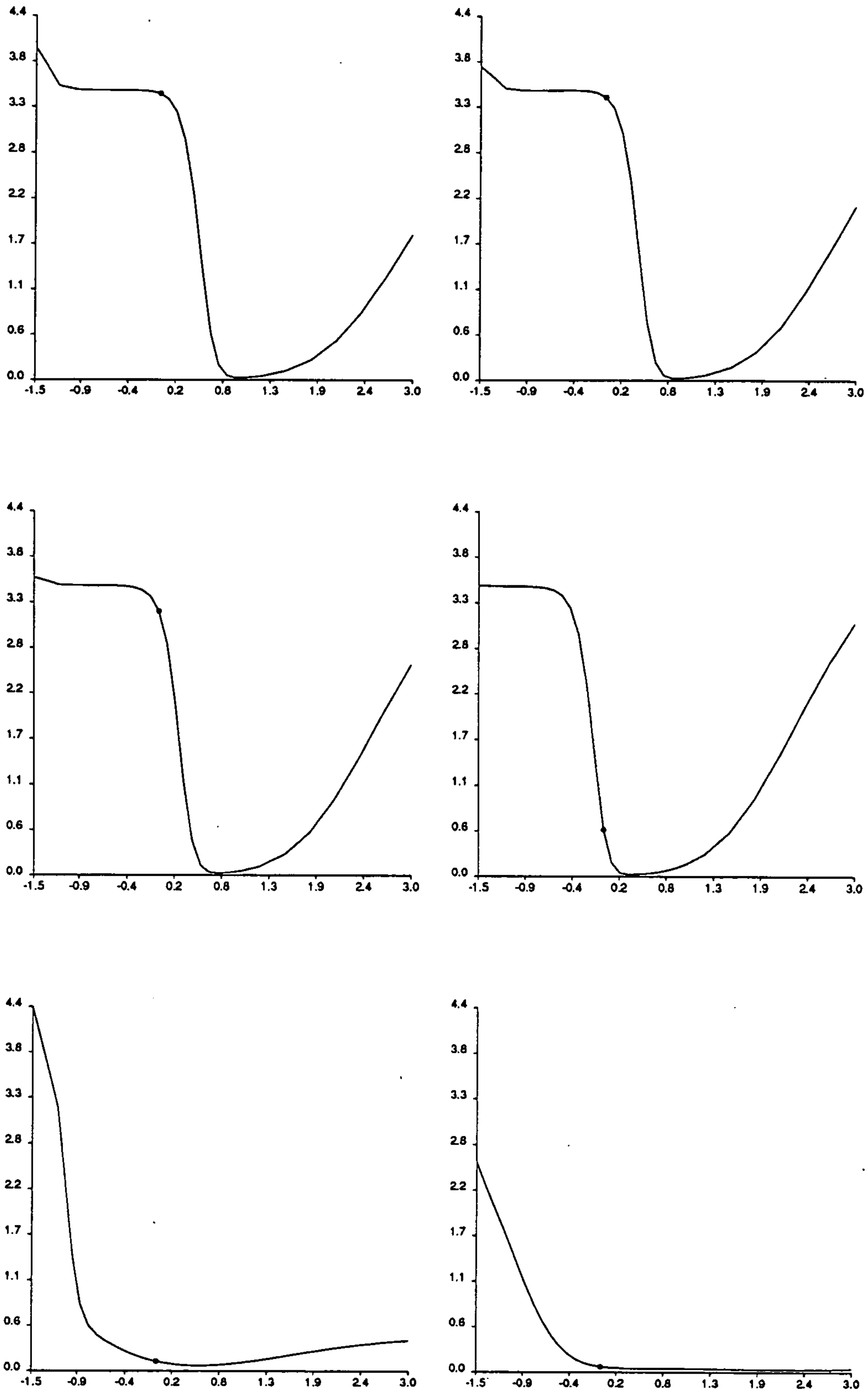


Figure 3.16: Location of a cliff-like crack in the error surface. The small circle indicates the position of the system on the error surface (read left to right, top to bottom).

Comparison of the scale of these graphs with the 7–25–7 descent in figure 3.12 shows how small a crack was actually found by the system. It seems that the presence of the momentum caused the system, by chance, to jump onto the plateau rather than continue with a reasonably comfortable descent. That was presumably when the sensitive weight mentioned above was increased just too much that it caused binding of the incorrect output nodes to the extreme values of the sigmoid function. However, it is this kind of unpredictable behaviour in solving tasks such as this one which makes the bare gradient descent learning algorithm unattractive.

Sometimes the system was helped by interactively altering step size and momentum at various stages in the learning. However, this was not considered to be a very satisfactory way of pursuing a better learning procedure, since the error surface could not be predicted for an arbitrary network at any particular point in the learning. Below we introduce two ways of improving the performance of the algorithm.

3.6 The deformation procedure

3.6.1 Motivation and description

We have seen in the preceding sections how inability to learn is marked by the system finishing up in a local minimum, characterized by a certain number of outputs being incorrectly mapped, and bound to extreme values of the sigmoid function. It is also clear that the harder the problem becomes, the greater the number of outputs that are incorrectly mapped (see the higher minima in figure 3.6). Since it is clear that unless outputs are kept very close to the correct extremes of the response function they will very probably be "flipped" over to the incorrect extreme from which the learning algorithm is unable to extricate them, the sensible thing to do would be to ensure that this condition is met. To do this, and to learn greater difficulties, the answer is clearly to use easier mappings of the same class to push the outputs to the correct extremes, and then gradually to make the problem harder, keeping the outputs from flipping. We call this method *deformation of the error surface.*[7]

Thus the idea of deformation is to start the system off by training it to learn a relatively straightforward mapping task, and gradually to increase the difficulty of the mapping, in such a way that the task is eventually *deformed* into the task of the desired difficulty. This process can be viewed as a "topological" deformation of a problem which can be represented as a simple shape in some space, into a more difficult problem whose extra difficulty is represented by the same topological surface, forming a more complex shape in the same space. Alternatively, one can imagine a task of classifying articles of clothing. A basic picture of an article of clothing might be shown, followed by a set of progressively more unusual or highly decorated versions of such an article. In learning to classify or recognize a whole range of clothing, the basic object is understood first, in its essence, rather than presenting the whole set all at once and expecting the net to organize sensibly from the start.

The entire problem is completely defined by the error surface in multi-dimensional weight space. The harder the problem one requires the network to solve, the more treacherous will be the terrain of the error surface, and the harder it will be for the system successfully to descend into one of the global minima of the surface. Thus one can picture the deformation procedure as moulding the error surface about the point occupied by the system, as the system descends towards the point of the final global minimum. In this way the system is able to avoid a lot of the treacherous terrain it would have to descend were it started off at a random position on the final error surface. This technique does not guarantee descent to a global minimum; the difficulty used at the beginning, and the parameters used to vary the deformation, need to be such that the surface can be gently deformed, with the task required to be learnt to vary smoothly at each deformation.

Deformation can be compared with the simulated annealing technique [KGV83] in which the system is eased into a global minimum of the surface defined by the cost function by reducing the noise of the system down to the value it has in the actual

[7]Wieland [WL88] has suggested a similar technique for the gradual learning of a classification, involving learning exemplars far from the decision boundary first, and then working towards the boundary.

problem. The idea here is that by starting the surface descent at a high noise value (or high temperature), the system will tend to locate the global minimum from the beginning, and as temperature is reduced will remain within the basin of attraction of the global minimum. The difference between the two is that the annealing prevents the system from becoming trapped in local minima, while the deformation removes the need for the system to descend a hazardous surface, by moulding the surface around the system.

3.6.2 Deformation and the rounding problem

The rounding problem is clearly an ideal candidate for the deformation method. The deformation parameter is r, which is to be decreased in stages from 0.5 to a final value r_0.

The difficulty can be varied continuously in the rounding problem, so it is necessary to determine the change in r required as a function of r. For a problem with a discrete set of difficulty levels it may be a simpler matter to determine such a schedule. Initially r was changed by a constant factor (0.99) each time. It was found that the factor was required to be closer to unity as the r decreased in order that the system remained near enough to the bottom of a "ravine-like" structure in the error surface that it did not break out of it into some local minimum (i.e. "flip"). Thus it is necessary to find some way of getting the r-change to cause an alteration in the error surface which is sufficiently small that the new error is not significantly different from that attained after completion of learning for the old r.

The expression derived below gives the change in position of the system on the error surface (E) after it has been deformed due to a change in r.
The error defined in (3.18) is rewritten as

$$E = \sum_p \sum_i E_{ip} \tag{3.54}$$

$$\text{where} \quad E_{ip} := \frac{1}{2}\left(t_{ip} - o_{ip}\right)^2 . \tag{3.55}$$

After a difficulty has been learnt the system is able to round numbers outside that r to zero or one respectively (within a tolerance *tol*). Thus for a single output unit and a single picture the maximum error at the end of a deformation cycle is given by

$$E_{max} = \frac{1}{2}(tol)^2. \tag{3.56}$$

It is necessary to control the change in r such that the error at this output unit increases by the same (tolerable) amount each time. The assumption is that the error at the other units, and for other pictures, will behave similarly, or at least no worse than that at the output unit with the maximum error.

In deriving the expression for $\frac{\partial E_{ip}}{\partial r}$ for a three-layer network we use the same notation as in section 3.2. Thus

$$E_{ip} = E_i(O_i) \tag{3.57}$$

dropping the subscript p, and so

$$\delta E_i = \frac{\partial E_i}{\partial O_i}\delta O_i, \tag{3.58}$$

where

$$\delta O_i = \sum_j \frac{\partial O_i}{\partial H_j}\delta H_j \tag{3.59}$$

for fixed values of the weights. Similarly

$$\delta H_j = \sum_n \frac{\partial H_j}{\partial I_n}\delta I_n. \tag{3.60}$$

Using equations (3.44) and (3.45), equations (3.59) and (3.60) become

$$\delta O_i = \sum_j \left\{ w_{ij}^H O_i (1 - O_i)\,\delta H_j \right\} \tag{3.61}$$

$$\delta H_j = \sum_n \left\{ w_{jn}^I \frac{(H_j + 1)(1 - H_j)}{2}\delta I_n \right\}. \tag{3.62}$$

From the definition of E_i we have that

$$\begin{aligned} \delta E_i &= -(t_i - O_i)\,\delta O_i & (3.63)\\ &= -\sqrt{2E_i}\,\delta O_i. & (3.64)\end{aligned}$$

Due to the pictures that are presented to the system, δI_n is only non-zero for one of the input units per input picture, and always has the values:

$$\begin{aligned} \delta I_n &= +2r \quad \text{for} \quad t_{i=n} = 1 & (3.65)\\ \delta I_n &= -2r \quad \text{for} \quad t_{i=n} = 0. & (3.66)\end{aligned}$$

Now, substituting (3.61) and (3.62) in equation (3.64) we find

$$\begin{aligned} \delta E_i &= -\sqrt{2E_i}\,O_i(1-O_i)\sum_j \left\{ w^H_{ij}\frac{(1+H_j)(1-H_j)}{2}\sum_n w^I_{jn}\delta I_n \right\} & (3.67)\\ &= -2E_i\left(1-\sqrt{2E_i}\right)\sum_j \left\{(1+H_j)(1-H_j)w^H_{ij}w^I_{jn}\delta_{ni}\right\}\delta r, & (3.68)\end{aligned}$$

where δ_{ni} is here the Kronecker delta. Hence the change in error with r is given by:

$$\frac{\partial E_i}{\partial r} = -2E_i\left(1-\sqrt{2E_i}\right)\sum_j \left\{\left(1-H_j^2\right)w^H_{ij}w^I_{jn}\delta_{ni}\right\}. \qquad (3.69)$$

Equation (3.69) was used in determining the amount by which r should be changed after each stage in the deformation had been successfully learnt. The maximum tolerable error change (δE) was taken to be 0.0005 for all simulations (with tol = 0.1), however this value is not critical.[8]

3.6.3 Learning digits in the presence of noise

The schedule for deformation in the last section was somewhat complicated. This was mainly because it was desired, and it was possible, to obtain such a schedule that perfect learning could be ensured. In most real-world problems however, this

[8]It is important for δE not to exceed an upper limit (so that the system stays in the ravine), while an optimum value is determined by the minimum number of cycles required to learn at the new r.

is not viable. So as an example of another problem in which deformation can be used to improve significantly on the bare learning procedure, the deformation procedure being any reasonable one, we consider the learning of digits in the presence of noise. The deformation parameter here is the degree of noise in binary images.

The network used has a 45–10–45 architecture. The input and output layers are to be viewed as 5 by 9 arrays of pixels. The input units themselves take on only the binary values 1 or 0. The training set consists of a set of noisy images of digits which are to be mapped to their corresponding clean images at the output. The difficulty of the problem is a function of the amount of noise present in the inputs, since the greater the noise the less the basic structure of the digit is seen. Thus one can imagine the error surface becoming very hazardous at various points, especially when the training set contains digits already very highly correlated without noise.

The training schedule is clear: teach the network first of all the clean images (i.e. N–H–N encoding), and then introduce noise at the input patterns, until the desired noise value is obtained. That is, the final operation of the net is to be one in which for any digit corrupted by noise of value less than or equal to $n\%$, a clean image of a digit will be produced at the output. For relatively large noise values it may be the case that the noisy image of a particular digit is "closer" (in terms of a distance measure the net is using) to another digit. In this case the net should produce as output the second digit. The network can be viewed as a device (characterized by a particular noise tolerance n) which cleans up noisy images by producing at output the digit which is closest, in terms of general structure, to the input image.

It is clear that the error surface for such a functionality is necessarily very highly structured and will contain many crevices and steep descents.

First we observe the performance of the basic algorithm on the 5%, 10%, 15% and 20% noise domains. Each training set consists of ten examples of each digit, i.e. 100 patterns in all. The learning parameters used here and in all subsequent runs are $\alpha = 0.9$, $\eta = 0.1$, $tol = 0.15$.

% noise	*final error*	*% patterns correct*
5	20.018	80
10	20.021	80
15	20.027	80
20	20.032	80

Table 3.5: Performance of the basic algorithm (training set).

The network was not able to achieve 100% success in any of the noise categories. (A run was terminated after 10,000 epochs, when the rate of change of error was slower than one part in a thousand per epoch, indicating a local minimum of the type in section 3.5.3 had been located.) Table 3.5 shows the performance of the network in terms of the percentage patterns correct.

We show in figure 3.17 typical ways in which the network got stuck. The pictures show the input, hidden and output unit states for a particular pattern in the training set. In one case all the mappings were correct apart from all the "1"s with one pixel wrong, and all the "2"s with the same three pixels wrong. This type of error is characteristic of the "flipping" in the last section, and confirms suspicions that the network had reached local minima. Clearly, certain patterns are very similar to each other, and the net is most likely to descend into a local minimum giving rise to mixture states. The minimum the net is required to reach probably becomes either narrower or further away (or both), the more noise that is present.

% noise	*final error*	*accumulated epochs*	*% patterns correct*
0	0.467	1051	100
10	1.021	1430	100
20	0.608	2351	100

Table 3.6: Deformation schedule 1 (clean $\rightarrow$ 10% $\rightarrow$ 20%).

Next three deformation procedures were tried. The first involves the sequence clean $\rightarrow$ 10% $\rightarrow$ 20%, the second the sequence clean $\rightarrow$ 5% $\rightarrow$ 10% $\rightarrow$ 15% $\rightarrow$ 20%, and the third clean $\rightarrow$ 20%. It was not attempted to find an optimal defor-

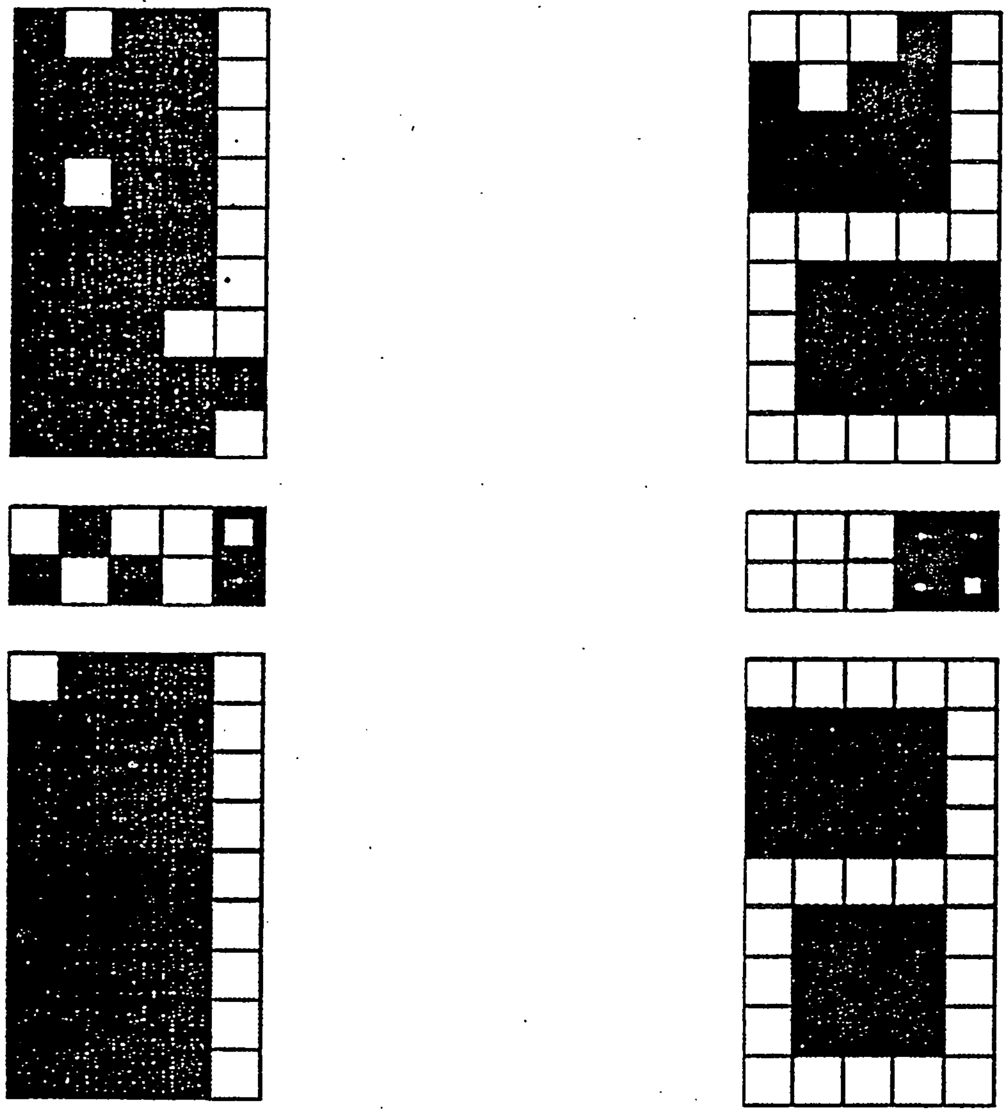


Figure 3.17: Typical incorrect mappings learnt by the bare algorithm.

% noise	*final error*	*accumulated epochs*	*% patterns correct*
0	0.467	1051	100
5	1.122	1300	100
10	1.021	1562	100
15	0.302	3361	100
20	1.505	15,000	99

Table 3.7: Deformation schedule 2 (clean $\rightarrow$ 5 $\rightarrow$ 10 $\rightarrow$ 15 $\rightarrow$ 20).

% noise	*final error*	*accumulated epochs*	*% patterns correct*
0	0.344	1160	100
20	1.563	5000	99

Table 3.8: Deformation schedule 3 (clean $\rightarrow$ 20).

mation schedule for learning up to the 20% noise training set. These experiments were done to demonstrate the suitability of the deformation procedure for this type of problem. It is not even necessary to use such a hard problem; as was suggested above the idea is more to build on current more general knowledge in a sensible way. We show below that deformation enables the network to find very good minima in a hazardous error surface. Deformation may help even when a global minimum may not exist (i.e. in the cases when there are conflicting members present in the training set), by keeping track of the optimal minimum using previous knowledge.

Tables 3.6, 3.7 and 3.8 show the performance of the net for each deformation schedule. Using the first schedule the network was able to learn successfully all the training sets. Typical mappings for the 20% noise network are shown in figure 3.18. Using the second or third schedules the net was not able to complete the learning, but the local minima in which it got stuck are much lower than for the basic net. Actually nearly all the patterns were correct. We show an example of an incorrect mapping in figure 3.19. The optimum deformation schedule lies somewhere between the second and third schedules tried above.

Turning to figure 3.18 again, it can be seen how the net performs the mapping of apparently quite different noisy images of the same digit, by responding to the features in the image which are most typical of the digit. This can be seen in the activations in the hidden layer for patterns in the same digit class. This representation in the hidden layer is then used to reproduce the digit at the output layer. This is a more general example of the **grandmother cell** mechanism, in which a certain hidden unit (or units) is responsible solely for the recognition of a particular family of features, or patterns. In this case, the same units have approximately the same states for noisy versions of a particular digit. Thus, instead of assigning a single hidden unit for the recognition of a particular digit, the network assigns certain *vectors* in hidden-unit space (section 5.2 explains this idea more fully) to be the encoded representation of the family of noisy versions of the same digit. The generalization afforded in this way (unseen noisy versions can be recognized correctly, so long as the noise is small enough) is of a a content-addressability nature, as opposed to that produce through indirect learning of trends typical of the entire training set. Without the two-level processing capability allowed by the layer of hidden units, networks would not be able to perform most interesting tasks involving extraction of the relevant information from the activations at the input. The net recognizes the typical patterns which are sufficient to identify a particular digit. These are by no means obvious, looking at figure 3.18, but clearly the inputs have enough in common to warrant the similar hidden-unit representations. The common properties of this class of patterns are known as the *minimal microfeatures*, or minimal information in the input pattern sufficient to distinguish it from other patterns in the group and therefore to classify it correctly [KL89]. The representations achieved in the hidden layer is in general an interesting and important property of the network studied in this thesis, and chapters 4 and 5 contain further discussions on the subject.

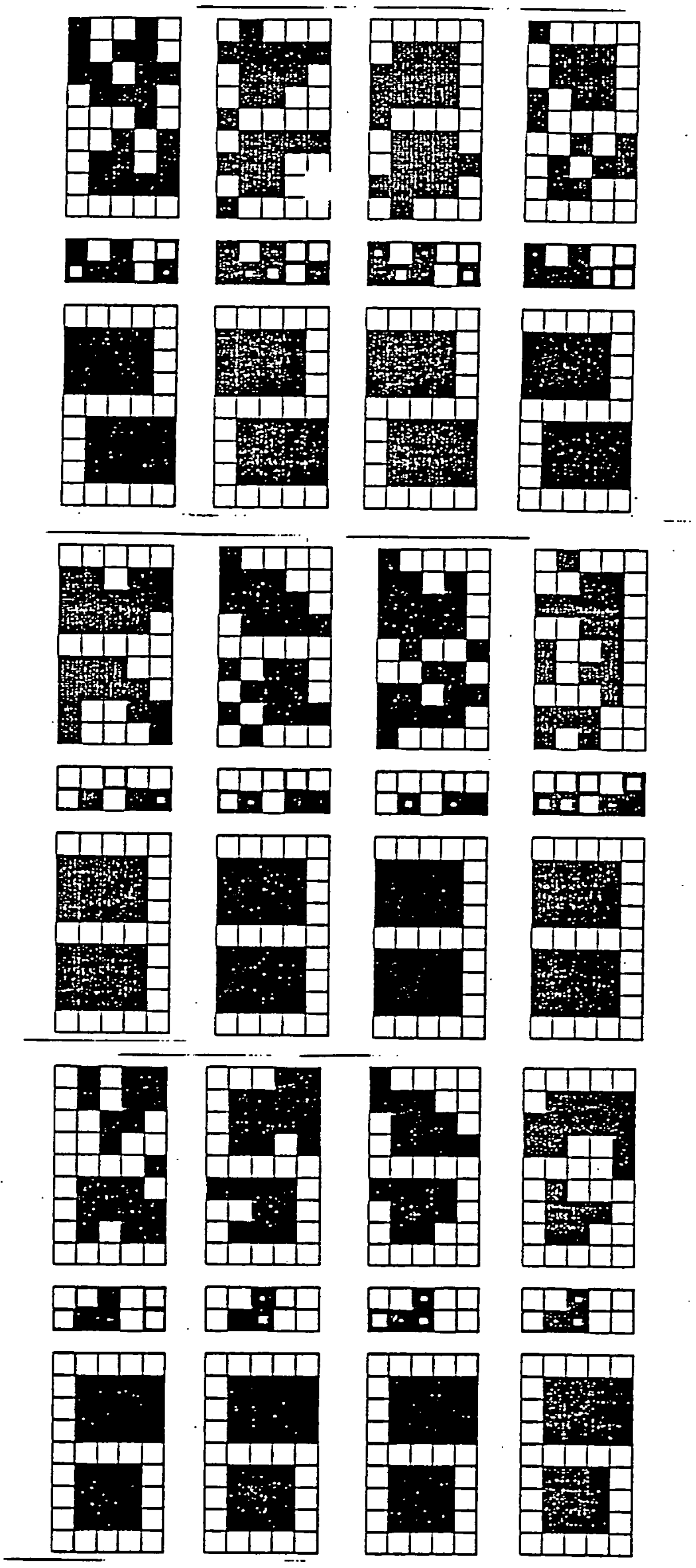


Figure 3.18: Examples of successful hard mappings which the deformation procedure found. Note the similarities in the hidden-unit representations of identical letters.

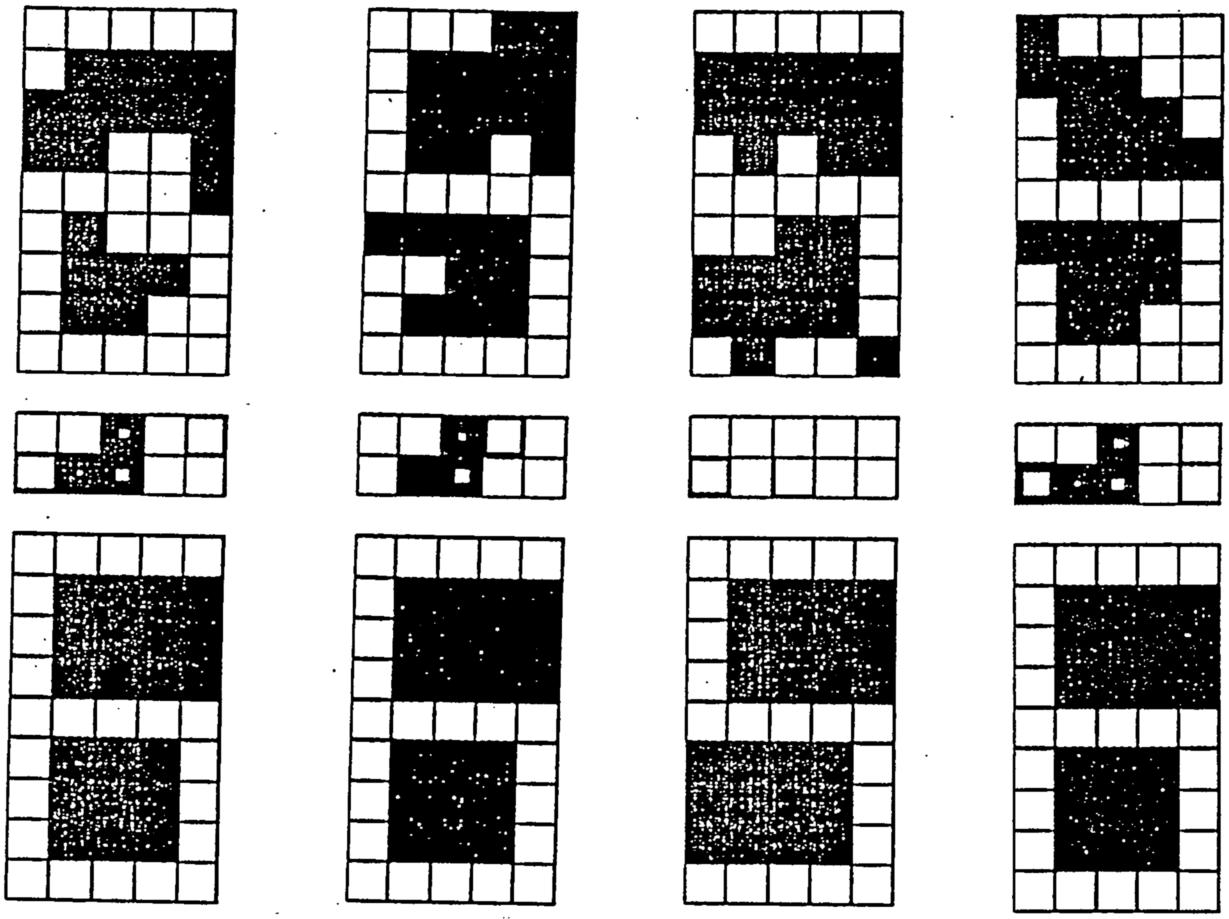


Figure 3.19: An example of a case in which the deformation procedure got stuck. Notice how at such a high noise level the network actually *mistakes* the digit for another digit, which it can be seen is a plausible alternative.

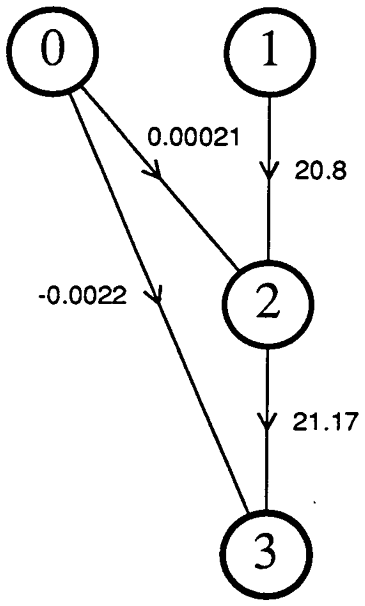


Figure 3.20: Example of the weight situation in a 1–1–1 system at a particular point in the learning. The input is at the top and output at the bottom. The numbers on the lines indicate weight values and those in the circles label units.

3.7 Varying the learning parameters

3.7.1 Critical slowing-down

The deformation process was very successful in the rounding problem in allowing networks to solve tasks of much greater difficulty ($r_0 = 0.0001$), however it became clear that the updating procedure for the weights became more inefficient as r was decreased. This can be demonstrated with a simple example where the network has one unit in each of its three layers (see figure 3.20). The system has just learned at the deformation stage of 0.005, and is about to start error propagation at the next r of 0.004881. The numbers which are presented are: 0.504881 (R_+) and 0.495119 (R_-), with the state of the threshold unit always at 1.0. The total error at the output unit at the end of the last r is 0.01. Tables 3.9 and 3.10 show the δ's and gradients at this point in the training of the system.

It can be seen how inefficiently the large weights are updated. The reason for this

input	δ^{out}	δ^{hid}
R_+	0.00988	0.002445
R_-	-0.00988	-0.002446

Table 3.9: Values of δ for $r = 0.011$ for the system in figure 3.20.

weight	*gradient*
w_{20}	-0.000001
w_{21}	0.0000477
w_{30}	+1.0E-08
w_{32}	0.00200

Table 3.10: Gradients for the weights in the system in figure 3.20 ($r = 0.011$).

small update, despite the comparatively large δ's, is contained in the expression for the gradients

$$\frac{\partial E}{\partial w_{ij}} = -\sum_p \delta_{ip} g_{jp}. \tag{3.70}$$

The values of the δ's remain similar as r is decreased, since the deformation ensures the system remains close to the tolerance error, while the values g_{ip} *decrease* with decrease in r. Thus the time (in learning cycles) taken to learn each new r unavoidably increases as the system learns to round numbers closer to 0.5. This is to be expected, since the values of the (heavy) weights required also scales inversely proportionately with r.

3.7.2 The problem of "valley ascent"

Steps were taken to try to speed up the learning, and it was found that the acceleration provided by the momentum parameter was very effective — *provided the acceleration was suitably controlled.*

With acceleration turned off, the following fate often befell a system. Figure 3.21 shows how the system climbs *up* a valley using gradient *descent*, bouncing from one wall to the other (the initial point is the lowest, with each successive point joined).

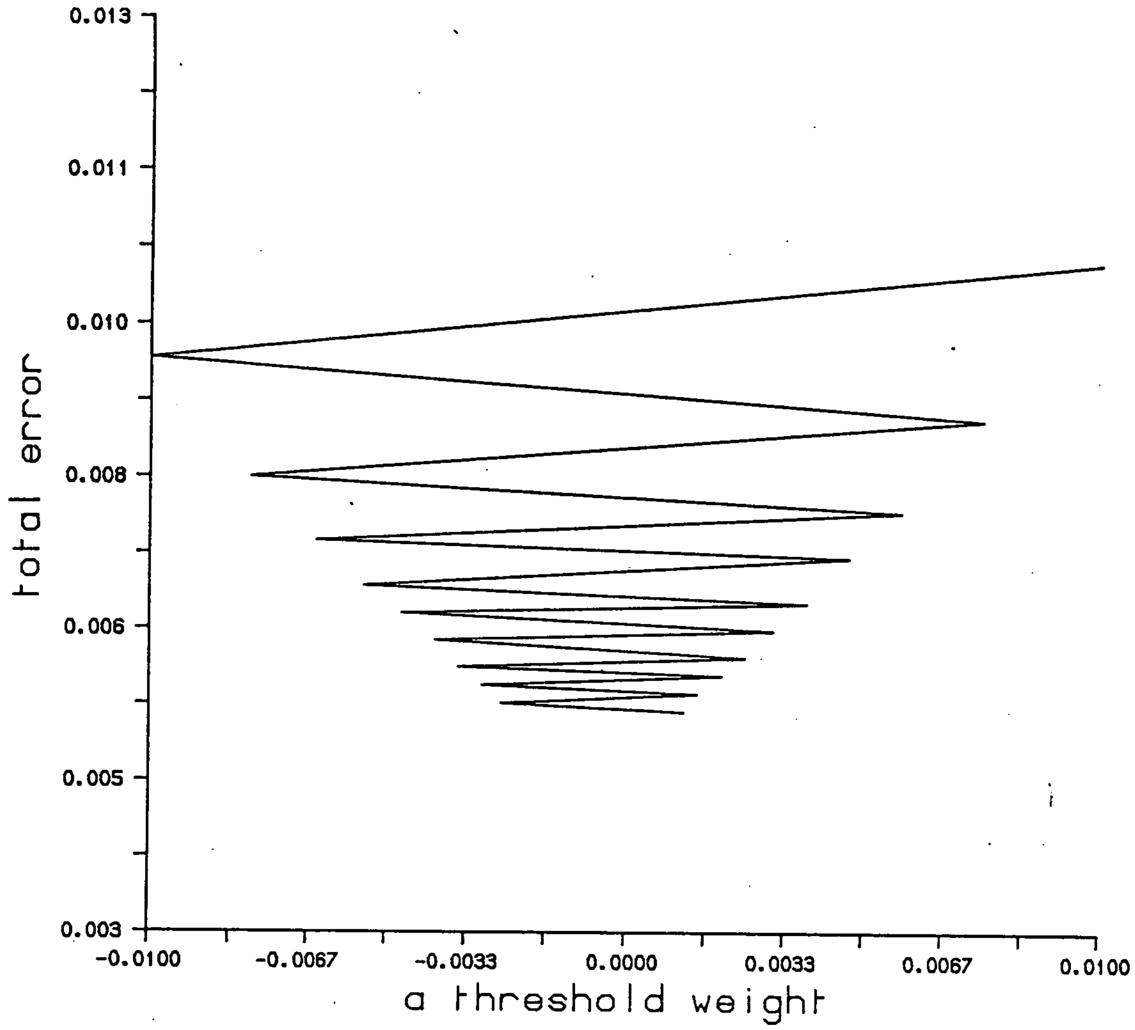


Figure 3.21: "Valley ascent". The system starts at the lowest point and each gradient calculation sends it higher up the valley.

The explanation for this effect, which ultimately leads to the system hanging on a flat region outside the valley (again the "flipping" mentioned above), is that the step size at the first (lowest) point is just too big at that point on the valley wall to produce a weight change which will send the system down the valley. Thus the system finishes at a point higher up on the opposite valley wall. It might be expected that with its next step the system would have rectified this, there being less chance of the weight change being so large that the same occurrence is repeated. However, this is hardly ever the case, due to the effect of deformation on the *shape* of the valley. This is best illustrated when we observe the alteration in the error surface, taking a cross-section in the direction of the threshold weights (θ_i) about the value threshold $= 0$, as the value of r is decreased to very small values. This error map is shown in figure 3.22, and it is clear how due to the great steepness of the valley walls, which increases as the valley is climbed, the system is squeezed out of the valley, with no chance of getting back in. Such an occurrence is much more likely when deformation is used, because the system is guaranteed to remain in such a narrow valley, at these difficulties, while without deformation the valley would either never be found, due to the surrounding plateaux, or if found would only be descended a short way if at all. Note also from figure 3.22 how the topological shaping performed by the deformation is nicely illustrated — the system would be kept at some point near the bottom of the valleys, with the folding of the error surface happening harmlessly above.

3.7.3 Learning parameter variation (method A)

The valley ascent above became a recurring problem for all system sizes below a certain value of r. The method of combating this was to reduce η at the point this behaviour was detected. The onset of the valley ascent is marked by two weight changes in opposite directions, the second of which has a greater magnitude than the first. When this is detected η^I_{ij}/η^H_{ij} for that weight is reduced by the amount:

$$\Delta\eta^{I,H}_{ij} = \left|\frac{\Delta w^{I,H}_{ij}(n-1)}{\Delta w^{I,H}_{ij}(n)}\right| \eta^{I,H}_{ij}, \tag{3.71}$$

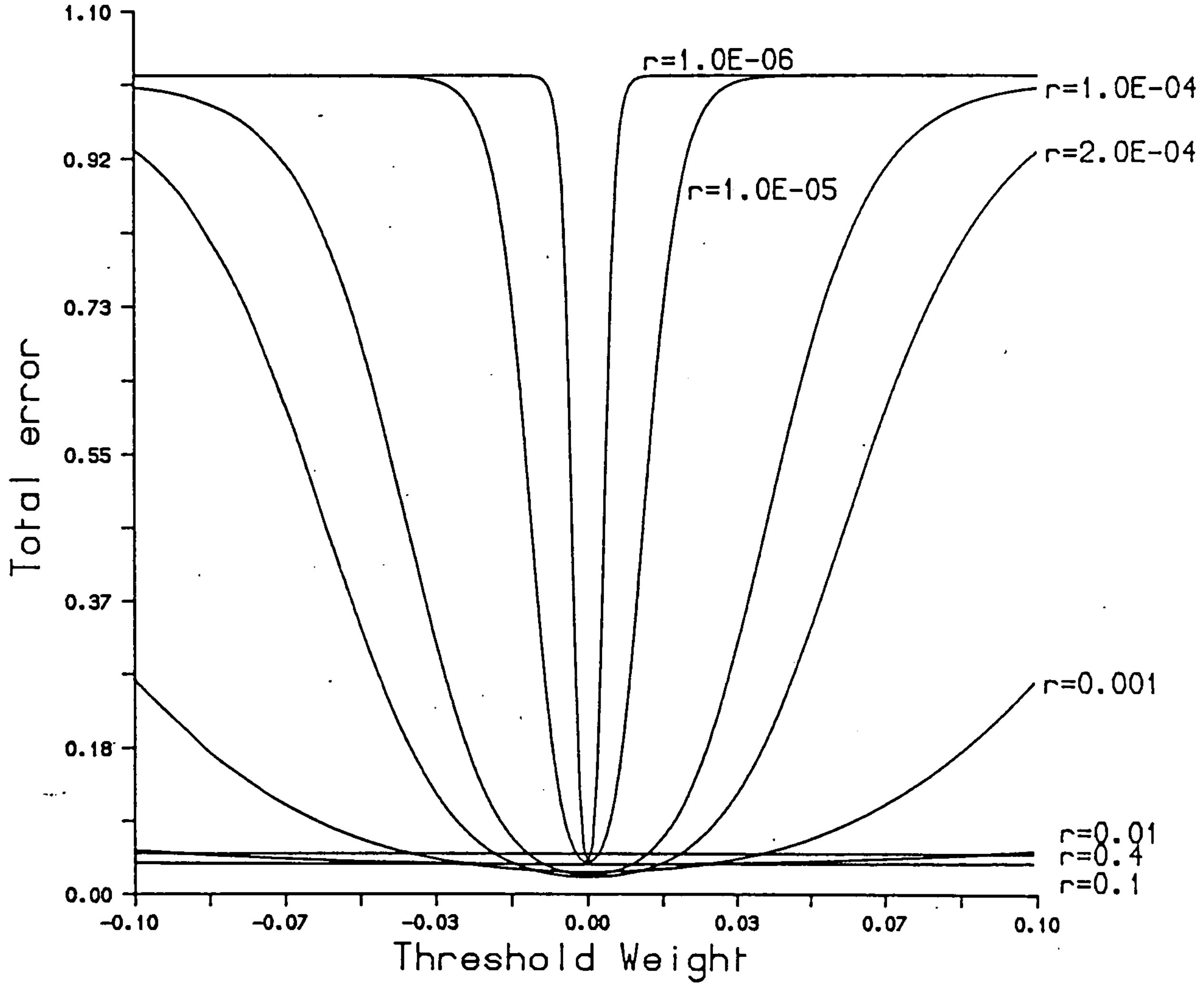


Figure 3.22: The terrain in the neighbourhood of a threshold weight for various levels of problem difficulty.

where the argument n indicates the update number as before. So the procedure is for a new step to be made from the backtracked previous point using this new value of η. Thus η now becomes dependent on the direction in weight space.

Use of momentum was found to be indispensable as a way of speeding up descent of slightly sloping regions (which characterize error surface in the directions of the heavy weights), and also for descent down valley walls when step size is small. However, it is important to ensure that momentum is 'switched off' whenever the descent reaches a stage at which it crosses the valley bottom (adding the previous weight change after this would result in ascent of the opposite wall). This is recognized by the gradient having the opposite sign on opposite valley walls.

By carefully controlling the speed of descent using this automatic parameter variation, and deformation, it was possible to solve tasks down to very small values of r (10^{-7}–10^{-8}), for all the systems studied. A comparison with the typical performance of the basic algorithm in figure 3.5 shows the effectiveness of the improvements suggested here.

3.7.4 Learning parameter variation (method B)

The method of changing η and α just described is good really only for a special case of descent such as the rounding problem provides. Thus a second more general way of altering these parameters was developed, and in fact used in the simulations in chapters 4 and 5, for a fast but safe descent.

The general task of gradient descent algorithms is to move the system down to the bottom of the nearest minimum, and this should be done as quickly as possible. However there are one or two points to consider first:

- **The size of the minimum**: it is not desired to descend into the nearest slight depression in the surface, or little dip. The question is however, when is a dip a valid (but not global) minimum, and when is it merely a "glitch"?
- **The speed of the descent**: exactly how fast is fast enough?

The first point is the hardest one in any descent procedure; it was decided here to descend into *whatever was nearby* but to allow leaps and bounds of a certain (specifiable) size, in order to free the system from any minor glitches in which it would otherwise perfectly trap itself, using the rest of the variation technique described below. Thus the methods for variation of the parameters η and α were based on control decisions over and above variations in network height caused by glitches in the surface. In practice this means that the parameters were not varied if there was a rise in error less than the amount taken to be a glitch, which we shall call ϵ. This is a way of allowing a small amount of "jumping" to enter into the descent, although it is still strictly a non-stochastic procedure.

The second point is similar, and follows on from what was said in the last paragraph. We allow the system to descend as fast as possible up to a rate Γ, and in so increasing the rate ensure that there is *no jump in error greater than* ϵ.[9] We shall denote the rate of descent by

$$R = \frac{1}{E}\frac{\Delta E}{\Delta t} = \frac{E(n-1) - E(n)}{E(n)} \tag{3.72}$$

for integer units of time t, (n)

The procedure for changing the parameters basically falls into the *control* and *speed* types:

- $R \geq \Gamma$ (satisfactory descent): No change to the parameters.
- $0 \leq R < \Gamma$ (requires speed-up): the surface is being descended, but not fast enough. The parameters η and α are both increased:

$$\eta' = \eta \times \eta_f \tag{3.73}$$

$$\alpha' = \alpha + \delta\alpha \tag{3.74}$$

where there is the constraint

$$\Delta\alpha = \alpha_f \tag{3.75}$$

$$\text{but if} \quad \alpha' \geq 1 \tag{3.76}$$

$$\Delta\alpha = \alpha_f / i, \tag{3.77}$$

[9]Note that this does not allow gradient *ascent*, but merely odd leaps of a limited size.

and so on until the constraint is satisfied, or $i > 100$ (further precision is not necessary for the momentum parameter). The initial value of i is 1.

- $-\epsilon \leq R < 0$ (small leaping): No change to the parameters.
- $R < -\epsilon$ (requires control): The wild behaviour is being caused either by a too large α or a too large η, or both.
    1. First α is set to zero, the network backtracked to the last point and the move made again. If R is now ok, the acceleration was clearly to blame, and is reduced by the amount α_f.
    2. If R is still bad, the step size must be too large, so it is reduced by the factor η_f, α remains off (so as not to confuse the issue), the step is backtracked and a new step calculated. This continues until a value of η is found which brings R to a satisfactory value.

The values of η_f and α_f are not too critical, but they are required to be small enough to allow a reasonable range of values of α and η to be tried.

3.8 Summary

In this chapter we have been concerned with technical aspects of the backpropagation learning algorithm. The limitations of the basic algorithm were demonstrated with the performance of the feed-forward net on the rounding problem domain. Before suggesting ways of improving this performance, we introduced the ideas of **learning curve, error maps in weight space and local minima.** These were used for an appreciation of the processes going on in the learning. The effect of the number of hidden units on the network performance was studied, and scaling laws suggested. The otherwise sensible introduction of a momentum term to speed up training was shown to have drawbacks in that it could make the initial descent too uncontrolled, and jeopardize the rest of the learning. General "tweaking" of the parameters was found to be unsatisfactory as a method of speeding up learning, and so the deformation procedure was introduce. This procedure helped to stabilize the descent and enable much harder difficulties of mapping to be achieved, through holding the system always in a low value of error, while shaping the error surface *above*, and therefore away from, the optimization area of the system. Thus the difficulties of descending a treacherous error surface were never encountered by the system. The optimization had in fact been split into two distinct parts: deforming the error surface, and descending the error surface. It was demonstrated with the mapping of noisy digits how the procedure for effective deformation need not be particularly complicated for other types of problems.

The observation of valley ascent, combined with the critical slowing-down of the network learning, inspired the development of a method for automatically adjusting the learning parameters step-size and momentum, for a faster descent. The use of this method, and the deformation procedure, enabled a vast improvement on the basic algorithm performance, as well as a much better controlled descent. A more general procedure for adjusting the network parameters was also introduced, based on similar ideas, which will be used in the rest of the simulations in this thesis.

Chapter 4

The importance of underlying correlations

4.1 Categories of problem domains

If the brain can be thought of as consisting of a large number of simple nonlinear processing units, then surely an MLP, given enough layers and units, could perform any function, however intelligent, that we might wish it to? Indeed, it should be able to perform any function that the brain can. The exact nature of the processing may not be known, but one is faced with the observation that the brain *can* perform extremely complex functions, whilst comprising in the main just a large number of seemingly simple nerve cells, which have straightforward behaviour when observed individually, but whose collective behaviour can produce a myriad of high-level processing. However, just because it is possible for the brain, with its almost unlimited supply of neurons, to process information in a particular way, using a system of simple processing units, it may not follow that an MLP of a given size and connectivity is the ideal model for this processing.

With this proviso, it is evident that in order to talk sensibly about the kinds of training sets which are "learnable", we must first define the scope of the analysis. The scope shall be defined here using the three layer MLP, that is a multi-layer perceptron with input, hidden and output layers, and no others. We believe this is a natural unit for the discussion of any level of processing we might want a feed-forward neural network to perform, since it allows us to organize the processing into sets of `[input → representation space (extraction of relevant features) → linking of features to form new inputs → next level]` and so on. In an actual system the linking stage could probably be combined with the next input stage, but for analysis purposes, it is necessary to be able to specify outputs explicitly. The three-layer MLP is therefore considered here to be the basic processing network from which generalization might emerge.

Now it is possible to be more specific about types of problem domains. We shall consider three types:

High-level domains. These include all training sets which require more than one intermediate level of processing to map from input to output. Many of these would require perhaps just one extra layer before the input layer of the basic three-layer MLP, to provide the appropriate coding. As it is though, the MLP would be required, from trying to reproduce the target outputs, to find a single transformation which both recodes the inputs so that the salient information is being used in the processing, and combines this information into a form which allows the inferences at the output to be made. Examples of such domains include the many types of scene analysis, letter and (even worse) word recognition and (still worse) also understanding or pronouncing them, and most other visual and auditory cognitive processing which call for transformation-invariance of some description. Even if an MLP with more than three layers is used to solve the mapping, we believe that the necessity to organize more than one hidden layer will prevent the network from discovering solutions which first code the inputs sensibly, and then combine these features. In short, we believe that these problem domains are best implemented in stages, so that, if an MLP is used, it can be directed to solving a specific mapping (for example, first *identify* the letter `A`, then *identify* the word containing `A`, then syntactically process the meaning of the word, etc.).

Numerical domains. By this is meant those domains which are easily defined, but interesting only from the point of view of defining predicates of a certain order, and exposing the limitations of the processing which may be performed, but rarely from the point of view of generalization [LB87, PH87, ŚR88]. Such domains include many of the predicates used by Minsky and Papert [MP69] such as exclusive-OR and connectedness, and also random mappings and "counting" problems such as parity.

Low-level domains. As might be anticipated, these fall into a category roughly midway between the first two. They combine the interesting generalization possibilities of the first and the single level of processing of the second. Indeed, any high-level domains could be constructed in a hierarchical manner from combinations of low-level and/or academic domains (i.e. combining nets). This idea in general is an attractive one, and is broadly discussed in [Min79]. The low-level domains themselves are characterized by the input being in a form ready to be processed right away into the representational stage of the hidden layer, such that salient information can immediately be manipulated and combined in the hidden layer, ready for inference to be performed in the output stage. This allows any generalization to emerge in the form of key linking of inputs to their required representational form [BB87]. Many experiments with these types of domains confirm the suspicion that just the single intermediate layer is necessary for the MLP to find good solutions. Dodd remarks in [Dod87] how one intermediate layer was sufficient for learning texture information, and that "A second intermediate layer was used to try to avoid the problems of output coding but was found to be unnecessary when the MLP was otherwise suited to the problem." We believe that this is true for many problems and that if possible they should be re-coded so as to allow network learning to be a matter of forming associations between "minimal microfeatures" [KL89]. We demonstrate the emergent properties which are possible from such a class of problem domains in the following sections.

4.2 Overview of the chapter

It is often the case when analyzing neural networks from a mathematical or physics point of view to use patterns selected from a random probability distribution. Such mapping problems fall into the numerical domain category. Unfortunately, this eliminates one of the more interesting features of distributed representations: the ability to capture the similarity between concepts by the similarity of their hidden-unit representations, resulting in the ability to generalize in sensible ways.

In order to study such emergent properties, one must be able to study the type of domains with underlying regularity; these are the other two categories of problem domains defined above.

In this chapter we generate low-level problem domains, and investigate the suitability of such domains, as opposed to certain numerical domains, for learning by a three-layer MLP. We observe the emergent properties of these nets when they are used to learn such domains, including learning speed and generalization. We generate such domains by associating each input pattern with a target calculated from a function which depends on a set of (fixed) parameters and the pattern vector itself. The fixed set of parameters serves to link the set of input/target pairs in the training set, such that we can speak of an **underlying correlation**[1] between each of the pairs.

The success of the MLP in learning low-level domains, which can also be referred to as the **natural** domains, since they are often those lifted from natural processes and functions, is demonstrated in experiments such as "NetTalk" and "NetSpeak" [SR87, MBB87] (translating from text to phonemic codes), a net which learns the past tense of English verbs [RM86], a backgammon-playing net [TS88], nets which perform medical diagnosis [1C85, YPB88], texture classification [Dod87] and predicting protein secondary structure [QS88].

In this chapter we suggest a method of generating *artificial* problem domains displaying similar characteristics to the natural domains. Once this is established, it is possible to study generalization and learning properties of problem domains clearly more suited to exploiting the emergent properties of MLPs, but using definable training sets.

The chapter has the following layout. Section 4.3 demonstrates the success of MLPs in learning and generalizing from the natural problem domains, with a preliminary evaluation of the performance of the net in learning to predict the middle amino acid in a family of proteins, in a window of 5 consecutive amino

[1]In the sense that fixed correlations between the elements of the input vectors give rise to the particular set of target values, and so the targets are correlated because of their common set of underlying generators.

acids. The results suggest that this, and other natural problem domains, are special in that the network can deduce some **underlying regularity** in the set of examples with which it is presented, which thenceforth allows it to *predict* the middle letter in many unseen windows (or some other task, in other natural domains), basing its choice on the assumption of a similar structure for the whole family of exemplars it has seen.

Following the preliminary observations of the feed-forward network results on the protein problem, we suggest a **reduction** of such natural problem domains into a form which reproduces only their postulated basic underlying properties. In order to verify that this minimal representation of the problem is sufficient to qualify for the categorization of "natural domain", various aspects of the learning performance and the generalization behaviour of the feed-forward net are studied with training sets taken from the domain. If the reduced domain retains the properties of learning and generalization possessed by the natural domains, then it can be assumed that it also embodies the essential generating characteristics of such domains.

In order to provide controlled experiments, two further problem domains are monitored alongside the reduced natural domain. The first is defined by purely random target values (i.e. a numerical domain, with no underlying regularities intentionally built in), and the second has permuted target values taken from the reduced natural domain. These are both categorized under a "no correlations" and thus "no generalization" group of domains. All three training sets are described in section 4.5.

As well as supporting the suggested extent of the reduction, the learning properties also indicate that fewer hidden units are required in order to learn natural domains, as opposed to the other two domains. This, and also the emergent properties of the MLP on this domain, are discussed in the context of the representations of the set of patterns which the MLP forms in hidden-unit space.

The method used for the learning is the backpropagation algorithm, as in the last chapter, but the cost function is modified to permit more efficiently the learning of real-valued (rather than binary) targets.

4.3 Predicting protein structure

Proteins are constructed from sequences of amino acids, which are linked into a polymer with a specific sequence determined by the translation of the messenger RNA (ribonucleic acid) three bases at a time. The ultimate goal in protein research is to be able to design them. For example an amino-acid sequence would be specified such that, when synthesized, it would assume a desired three-dimensional structure, bind any desired substrate, and then carry out any reasonable enzymatic reaction. The current state of research has not reached this stage, primarily because of the difficulty of understanding *why* certain proteins exist, rather than any of the millions of other possible combinations of amino-acids. The structure of a protein can be divided into various levels. For example, the *primary structure* of a protein is its linear sequence of amino-acids, the *secondary structure* is the local spatial structure of small numbers of amino acids, independent of the orientations of their side groups.

It is currently possible to determine long stretches of "cloned" genetic material. This sequence information needs to be interpreted by its relation to known genetic sequences (over 10 million bases are currently known), by inferring what regions are copied for translation into proteins and by the assessment of possible biological function of the putative protein. The comparison of protein sequences with each other has been extensively developed (see, for example, [LHCC86] for a DAP implementation) and can be applied to the entire set of known proteins in the databases.

With the rapid increase in the numbers of known proteins, it is becoming ever more desirable to have some form of intelligent database.

The Hopfield [Hop82] neural network model has been used as a content-addressable memory to store sequences, working on the assumption that the contextual information of incomplete sequences will restore varieties of possible complete sequences, depending on the (bit) noise in the input pattern [Wal87b]. Other work has been performed using feed forward networks [QS88, NRR$^+$89] attempting to predict the secondary structure (alpha helix, beta sheet or beta turn) of proteins from windows of amino acids. The results using these networks look promising, giving good generalization performance, and they may possibly form a major part of a future hybrid database, to provide the search techniques with a modicum of expert knowledge as guidance.

In this section we use a feed-forward net to predict the middle amino acid in a window of 5 (primary structure information) from a sample of proteins all of which are in the family **trypsin**. All these proteins have a common function, and so we might expect the groups of amino acids found together in the proteins to be similar. The trend for particular groupings in the protein chain is what we expect to give rise to any generalization ability.

4.3.1 The format of the data

The proteins are represented in their raw form as a set of chains of letters (see appendix A.1). Each letter codes one of 20 amino acid groups. Our data set comprises 14 members of the family **trypsin**, each of which has about 220 amino acids. Although in reality the proteins are coiled up in some way, in three dimensional space (the secondary structure), so that the neighbourhood of a particular amino acid may consist of amino acids from a long way down the chain, for this experiment the presence of each amino acid is assumed to depend on amino acids only in the neighbourhood of the window size (2 amino acids either way). It can be seen from the results of the network learning, that this is a reasonable assumption.

It was decided to split the 14 proteins into two halves. One half, consisting of the first 7 proteins, formed the training set, and the second half the test set. In the experiment we used both sets to gauge the performance of the network.

4.3.2 Net architecture

The network architecture is shown schematically in figure 4.1. Four letters form the input to the net: the four letters which surround the middle one the network is supposed to learn. The middle letter is not input to the net as then all the net would need to learn would be a straight one-to-one mapping of the middle

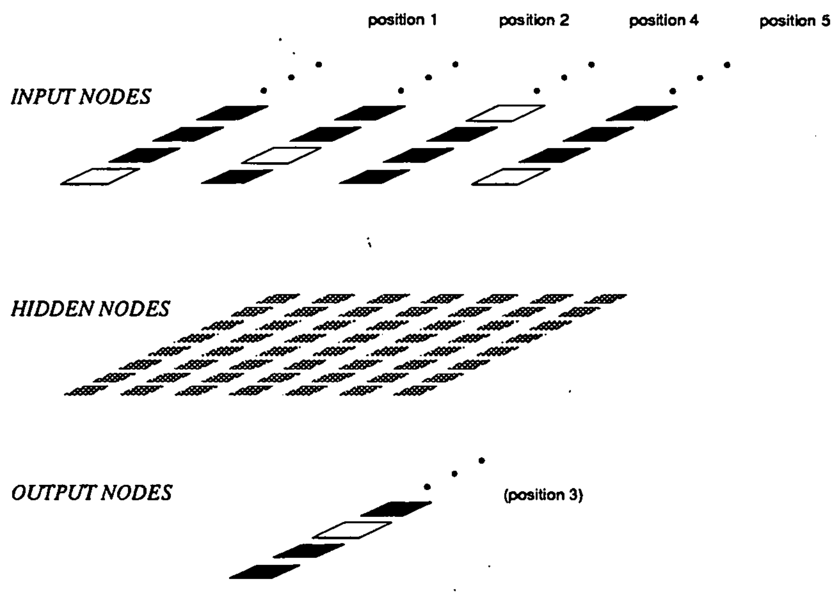


Figure 4.1: The net architecture for the protein experiment (weights not shown).

letter to itself, ignoring all the surrounding letters. The task of the network is to use the *contextual* information of the window amino acids to suggest possible middle amino acids. Each position in the window is represented by 20 letter-nodes. Thus if the window "ABCDA" were being read by the network, the four nodes representing A, B, D and A respectively would be activated at the input, with the target node being the node C at the set of nodes representing the middle letter at the output. This example situation is shown in figure 4.1, the state of +1 (on) being indicated by a white node, and 0 (off) by a black node. 50 hidden units are used, and the network is fully connected from input through hidden to output layer. The amino acids are therefore represented orthogonally by the network, which assumes no correlations exist between the amino acids, other than the ones which are discovered by the net in the course of the processing, and these are stored in the connections.

During a learning cycle each protein was presented to the network by shifting its length across the window, thus all but the four end amino acids (two at each end) per protein were used in the target set. The gradients were summed for the entire training set before the weights were updated (batch learning). This was determined to be the best method since periodic updating resulted in undesirable "recency" effects (later patterns are much better learnt than earlier ones, see also [Wal87b]), and also it was found that learning was much faster and ultimately more successful when batch learning was used. If the target for a node was 1 then the tolerance (*tol*) was taken to be 0.1, otherwise it was taken to be 0.2. (This was done because of the relatively large number of times a node state would be required to be zero as compared to the times it would need to be one, which might run the risk of the states getting trapped on the response function extreme, if they were trained to be too close to zero).

The total number of patterns in the training set is 1599, and in the test set 1576.

4.3.3 Performance

The performance of the network was examined twice: first after 1100 cycles of learning, and then after 8800 cycles. In order to test the network a protein was processed using the same window method as in the learning, and the states at all the output nodes were compared. If the node with the maximum activation, which was also greater than the acceptance threshold (a number between 0 and 1, which we use to cut off lower values when examining the network *after* learning, and which is not related to the tolerance used during the learning), happened to be the correct one (i.e. the target value for that protein window), the pattern was considered to be learnt.

The performance was judged on the basis of percentage correct patterns,[2] out of the whole testing set. The training set and the test set were both used, in order to gauge both aspects of the acquisition performance. Figure 4.2 shows the performance on both sets for both the testing sessions, as a function of the acceptance threshold.

Several interesting observations can be made from the graph:

[2]Note that this is equal to the percentage correct *nodes*, and so does not give rise to the same ambiguity as was experienced in the last chapter.

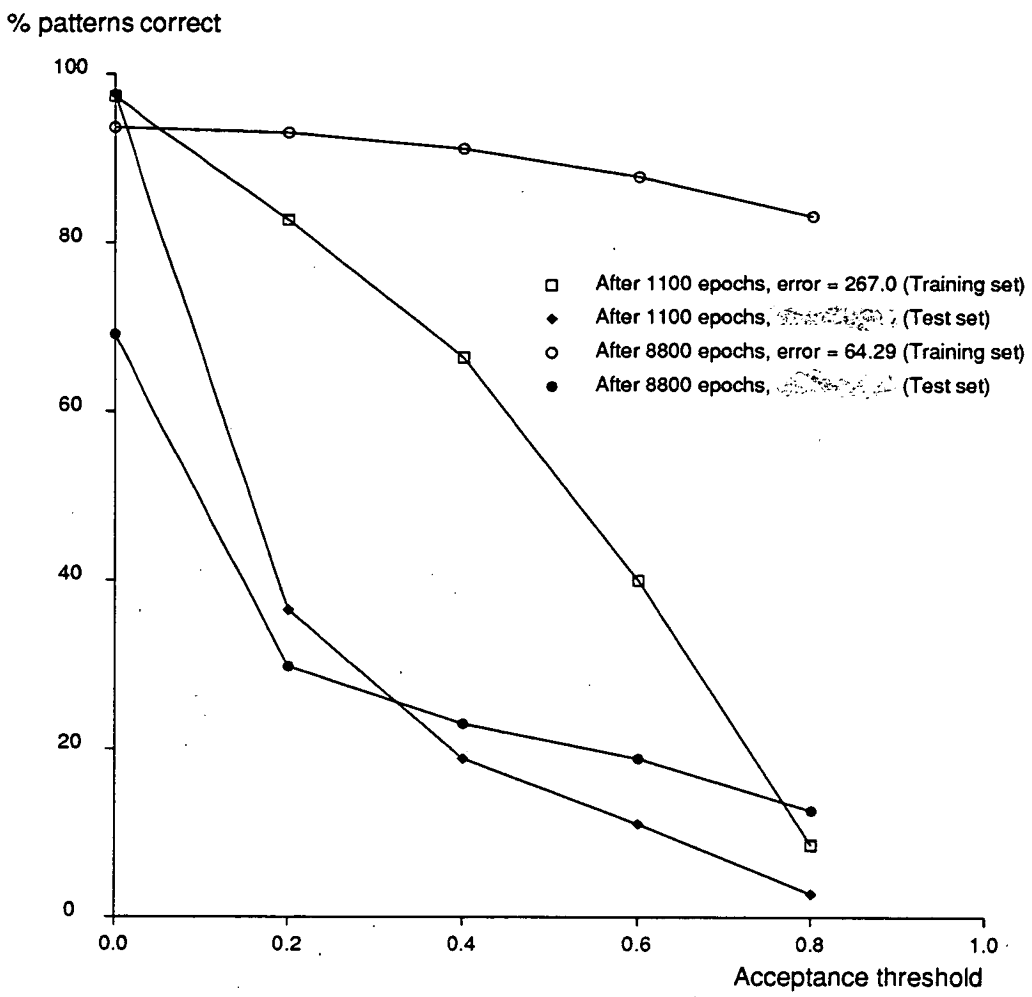


Figure 4.2: Performance of the network on the training and test set for the protein problem.

- The network can learn the training set to a high level.
- Increasing the acceptance threshold generally diminishes the performance.
- The diminution rate is much less for the well-trained network than for the less well-trained one.
- For the training set, the zero threshold performance for the well-trained network is not so good as that for the less well-trained one.
- The network generalizes on the test set.
- This generalization is better for low thresholds with the less well-trained net than the low threshold generalization for the well-trained net.
- The generalization performance diminishes slower with acceptance threshold for the well-trained net.

From the observation that the generalization performance is very good for low acceptance thresholds after only a short period of learning, it seems that the net learns the basic *structure* of the trypsin family fast. Since this low acceptance threshold generalization performance diminishes on training the net for a longer period, and a similar situation is seen in the training set performance, we infer that although the patterns are in general better learnt (we can be more confident about the output), the information about the general nature of the mappings is no longer so good. Also it appears that the fast learning of the basic training set structure is echoed in the generalization performance (i.e. the performance on the test set) which seems to have similar characteristics, with respect to zero threshold performance and diminution rate.

One of the more significant observations is that the better the net learns the training set, the worse the low acceptance threshold performance becomes, most markedly for the test set. This is, we believe, a common occurrence in such learning, where the general structure of a training set, or family, is gradually lost through *over-learning* of the training set.

There are a certain number of exemplars common to the training set and the test set. The total number of patterns which should not be put down to generalization of the correlations knowledge, but to replication in the test and training sets, is 109. Thus the net should always perform to about the level of 109/1576, or about 7%, of its training set performance, when processing the test data. If we assume no generalization, and that the net can merely guess the output letters on the basis of the output probabilities learnt from the training set, then we can work out the no generalization case of chance guesses. This behaviour (the probability of a correct guess on the test set is given by the sum of the products of the probability of each letter occurring in the training set and the probability of it occurring the test set) produces a 6.2% performance. The graph shows that the net always performs much better than if we were to attribute the stored knowledge solely to these factors, and so it is reasonable to assume that real generalization is taking place. The generalization normally takes the form of making some kind of decision as to what amino acid to use when the window may allow various possibilities. It is then that the knowledge gained implicitly, from the learning of the training set, of the correlations which characterize it, is brought into play. Thus, for example, if we have the test window `AA␣CD`, and there have been various occurrences of, for example, `AABDD`, `AEAED` and `CEECD`, etc., then the significance of each letter in the window for the presence or absence of any letter in the middle, is implicit in the network weights, since all these examples have been successfully mapped. This knowledge then is drawn upon in the form of the relative strengths of the letters suggested at the output, the largest output will be that which is most favourable for the window.

We saw how the zero threshold performance gave the best generalizing performance. This indicates that one should not be so much interested in actual size of output as in the relative sizes of output. This, then is the important information which the network extracts from the training set: the relative strengths of each amino acid for a particular window.

4.3.4 Discussion

The success of feed forward nets in learning about proteins indicates that there is something particularly consistent in the formations which are presented. In reality various forces are operating to determine the ultimate structure of a protein. These forces include electrostatic forces, such as direct ionic interactions between charged amino acids, dipole-dipole interactions, dispersion forces (very short range, with a strength depending on the shape of the molecules), hydrogen bonding, and the *chelate* effect (orientation dependent higher order interactions). All these forces act to determine the proteins which exist in nature. It is natural to consider these proteins to be low-energy states in the large group of possible configurations.

In this section we have seen how an environment of amino acids can provide sufficient information to allow the network to suggest a possible middle amino acid, using the knowledge it has gained from the learning period. In the next section we attempt to mimic the characteristics of the learning, by a simple model of the basic structure of a natural problem domain. This work does not propose to explain the way protein structure might be determined from the electrostatic potentials in which they exist, but merely to reduce this, and similar problems, to a tractable form, which still displays the vital features.

4.4 Reduction to a simple model

The important information which the net extracted from the training set in the last section was not the absolute *size* of the output activations, but the *relationships* between the outputs for each particular input. Thus we are not looking for the number of binary yes's in the output, but the relative strengths of each of the output node responses, to determine the information the network has deduced from the training set. In order to develop a training set which can be said to have similar properties to the protein, or "natural" training sets, we must define what it is that is special about the natural data. The postulate we explore is that there is **underlying regularity** in natural data, and that this can be reduced to a form of **correlation** between the individual elements of a pattern, giving rise to an activation of each pattern for a particular output state.

In the protein example, for instance, we can assume that the environment of letter C in the window ABCDE determines the *probability* that the letter C will be found within this grouping. The reasoning behind this is straightforward. If, as in the

Input	*target*	*number of occurrences*
1010	1000	10
1010	0100	7
1010	0001	3
1100	0100	7
1100	1000	7
1100	0010	7
0110	0010	10
0110	0001	3
0110	1000	3
1110	0001	1
1110	0100	7
1110	1000	3

Table 4.1: A training set with conflicting targets.

protein example, there is more than one possible target for a given input amongst all the training cases, that is input p has target t_{p1} in N_{p1} of the cases, t_{p2} in N_{p2} of the cases, etc., and if the training is done in batch mode (update only after seeing all the patterns in the training set), then the linearity in t of the error at the output layer implies that one may alternatively train that input with the *mean* target [Wal87a]:

$$p_p = \frac{\sum_k t_k N_k}{n}, \qquad n = \sum_k N_k. \tag{4.78}$$

If the training of all the patterns achieves this mean target output for that input, then the net will be producing a probabilistic output determined by the frequency of presentation. In the protein case, a particular letter i will be trained to have a target of one in n_i of the cases of the input window $\mathbf{I}_p$ being presented, and to be zero in the other $n_p - n_i$ cases.

As an example we shall consider the training set in table 4.1. The probabilities for each output node for each pattern p are shown in table 4.2, with the values settled on by a feed-forward network in actual simulation. The network settled to a constant error. It can be seen how the actual values obtained are very close to

pattern	P_p				*actual outputs*			
1010	0.5	0.35	0.0	0.15	0.5	0.35	0.0	0.15
1100	0.33	0.33	0.33	0.0	0.32	0.34	0.31	0.02
0110	0.19	0.0	0.63	0.19	0.19	0.01	0.61	0.18
1110	0.27	0.64	0.0	0.09	0.27	0.63	0.02	0.05

Table 4.2: Actual probabilities and outputs obtained, at a steady error. The tolerance was set to 0.01.

the theoretical probabilities.

Absolute strengths of the possible letters which may be found in groupings are of secondary importance to *relative probabilities* between the letters. Thus, if the network has learnt to map the most favourable letter to an activation a_f, for example, the important knowledge would be contained in the relative set of activations

$$\left\{\frac{a_1}{a_f}, \frac{a_2}{a_f}, \ldots, \frac{a_f}{a_f}, \ldots\right\},$$

which can be normalized to form a set of probabilities. The tests made on the network in section 4.3 actually indicate that the relative probabilities of letters in groupings were learnt very early on in the learning.

The model training set introduced below tries to embody the main characteristics of the natural training sets, in a minimal form. Actually the number of possible input letters is effectively reduced to two, and the effective number of output letters to one. We demonstrate the origin of this reduction below.

First assume that for each letter at each position in the window there is an independent set of interactions (correlations) with every other letter at every other position in the window. So, the input layer in figure 4.1 can be pictured as a fully connected Hopfield net [Hop82], where each connection T_{ij}^{ab} between letters i and j in positions a and b respectively represents an interaction strength. Note here that the situation has already been assumed to be representable by pair interactions alone. We now imagine a pattern of activations on the Hopfield net to comprise zeroes at all nodes except for those representing the letters present in a window, where the nodes are set to ones. There is an associated energy function for the Hopfield net, which we can write as:

$$E := \sum_{a<b}^{N_W} \sum_{i<j}^{20} S_i^a T_{ij}^{ab} S_j^b, \tag{4.79}$$

where S_i^a represents the state of node i in position a, and the summation is over all the node pairs, with $N_W = 5$ positions in the window and 20 possible letters. Now, for some letter groupings the energy E will be lower than for others, and we can take these to be more favourable states. More specifically, for a particular letter *environment* `AB␣DE` the energies of the system when letter `X` is at the centre position is given by the energy of *pattern* `ABXDE`.

For the next stage in reducing this problem, we reduce the number of letters in the input window to 2. Adding more letters will not affect the basic nature of the problem, since we are assuming the letters to be orthogonal (and therefore each letter is characterized by its own set of weights to the hidden layer, containing the information about its relationship with the rest of the letters in the window and the output letter.). Also, if we reduce the number of output letters to one, a further simplification comes about in that the presence of this letter and its associated interaction strengths is unnecessary. (That is, the letter itself does not need its own interaction strengths with the letters in the window to distinguish its energies from those of any others.) Thus, to recapitulate, there are now 2 types of letter which can be present in a window, in $N_W - 1$ possible positions, and each configuration of letters has a characteristic energy given by equation (4.79). This energy tells us about the probability of the output letter being found in the particular environment of letters.

The final stage comes in reducing the number of letter nodes in the input window to one. This is possible if we say that each position, now represented by a single node, can take on the two values $+1$ and -1, so that each window still represents an environment of $N_W - 1$ objects. A better way of viewing this is to consider that each position in the window is now characterized by the presence $(+1)$ or absence (-1) of an object. Now it can be seen that the situation we have is analogous to an interacting spin system. The probability of an output letter being in the environment of input letters has been reduced to the energy of a spin system consisting of pair interactions only. We can now simplify the notation, and so the probability of the output node being on when the set of inputs $\{S_i\}$ is presented to the net, is some function of the energy of the *configuration* of spins $\{S_i\}$, with pair interactions J_{ij}:

$$p(o) = f(E) = f(\sum_{i<j}^{N_I} S_i J_{ij} S_j), \tag{4.80}$$

where there are $N_I = N_W - 1$ input nodes (or, in the analogous model, spins). Finally, it is sensible also to include self-interactions I_i, so:[3]

$$p(o) = f(E) = f(\sum_{i}^{N_I} S_i I_i + \sum_{i<j}^{N_I} S_i J_{ij} S_j). \tag{4.81}$$

The major assumption made above is that the probability a letter will be present in an environment of other letters is a function of the letters present and where they occur in the window, and that this function can be expressed as one of a summation of terms involving increasing orders of interaction strengths between the letters, the only significant terms being the first and second order ones.

4.5 The training sets

Now we are ready to specify the three types of training set which will be used in this chapter. The networks used will be of size N_I–N_H–1, as in figure 4.3, and will be required to map a set of input vectors $\{\mathbf{v}^p\}$, where $p = 1, 2, \ldots, N_p$ labels the pattern, to a set of target outputs $\{t_p\}$, which are real numbers in the range $[0.1, 0.9]$. The training sets differ in the set of targets associated with the inputs.

[3]The reason for this is that otherwise the energies for mirror configurations will be identical, thus rendering the window equivalent to a size $N_I - 1$.

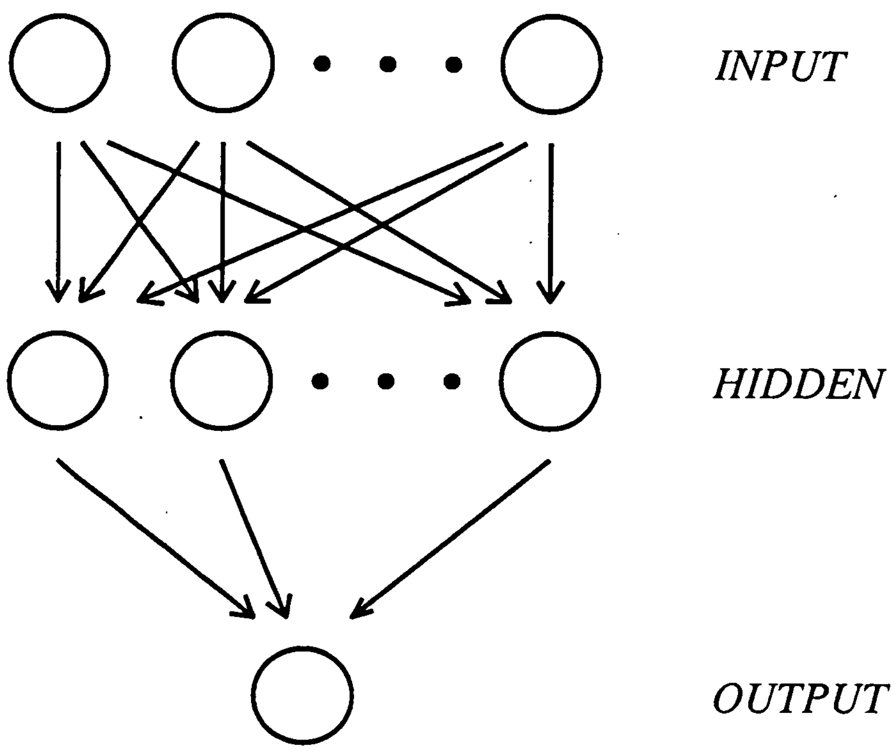


Figure 4.3: Network architecture.

The first training set is referred to as **correlated** and is derived from equation (4.81):

$$\begin{aligned} \text{input:} \quad \mathbf{v}^p &= \{S_i\}_p \quad \text{with} \quad S_i \in \{1,-1\} \\ \text{target:} \quad t(\mathbf{v}^p) &= f(\mathbf{v}^p, \{I_i; J_{ij}\}) \end{aligned} \tag{4.82}$$

which becomes

$$t(\mathbf{v}^p) = f(\sum_i^{N_I} v_i^p I_i + \sum_{i<j}^{N_I} v_i^p J_{ij} v_j^p). \tag{4.83}$$

The form of the function $f(x_p)$ is

$$f(x_p) = \frac{1-2t}{A}(x_p - B) + t \tag{4.84}$$

where

$$B = \min_p\{x_p\} \tag{4.85}$$

$$\text{and} \quad A = -B + \max_p\{x_p\}. \tag{4.86}$$

This simply rescales the energies in equation (4.83) so that the targets lie in the range $[t, 1-t]$. The value of t was taken to be 0.1, avoiding the very low gradient regions of the output response function. Thus the targets lie in the range $[0.1, 0.9]$.

Since the parameters $\{I_i; J_{ij}\}$ are fixed for a particular training set in this group, all the members can be said to belong to the same "family", in the same way as we understood the proteins to belong to the family trypsin. Any number of families can be generated, just by changing the parameters $\{I_i; J_{ij}\}$. In these simulations the parameters were chosen randomly from the range $[-0.05, 0.1]$ and $[0.05, 0.1]$, which was determined, in preliminary experiments, to be a suitable range for a good distribution of target values (i.e., if we allow widely differing interactions, by having a large range in which to choose them, we might find a particular letter with a very large influence on the total energy, thus giving rise to an *over*-correlated training set, which would not be useful for the properties we wish to investigate.)

This method of generating targets for each of the patterns can be considered a general way of synthetically writing in low-level similarities between patterns in a training set.

The second type of training set is referred to as the **random** set, and is generated by taking all the input configurations and associating each with a number, b_l, which is chosen randomly from a uniform distribution in the interval $[t, 1-t]$. The function f from equation (4.82) for the random group is therefore given by

$$\begin{aligned} f &= f(\mathrm{v}, \{b\}) = f(\{b\}) = b_l \\ \text{where} \quad b_l &\in [t, 1-t], \end{aligned} \tag{4.87}$$

Thus both the correlated and random training sets have normalized targets, in the range $[t, 1-t]$, and the same set of inputs.

The third type of training set is a combination of the correlated and the random. The random training set is really the worst-case type of control domain, and in order to make better comparisons of the more subtle properties of the correlated set we need to compare with a training set which is almost the same except that it does not possess consistent correlations between specific patterns. The training set is formed by taking a correlated training set and re-arranging the linking of inputs to target values. So this type of training set has *exactly* the same set of targets as a correlated training set, but they are associated with *different* input patterns. Thus the distribution of output values is of identical form to that of the correlated set, but the underlying regularity is missing. We shall call this the **permuted-correlated** training set.

In figure 4.4 are shown the numbers in the training sets ordered in terms of distribution in the output range, for examples of the random and correlated/permuted-correlated types of training set, for 5 and 7 input nodes.

4.6 The cost function and learning algorithm

If we write the actual output obtained when the pattern $\mathbf{v}^p$ is processed through the net as o_p, then the cost function to be minimized is defined as

$$L := \sum_p^{N_p} \left\{ t_p \log \frac{t_p}{o_p} + (1 - t_p) \log \frac{(1 - t_p)}{(1 - o_p)} \right\}. \tag{4.88}$$

The origin of this cost function is from a measurement of the "distance" between probability distributions: for a given input pattern $\mathbf{I}_p$, the output o_p tells us the conditional probability

$$P^+ = P\{x = \mathtt{ON}|p\} = o_p$$

that the attribute x represented by the output is on. Clearly we also have the associated probability

$$P^- = P\{x = \mathtt{OFF}|p\} = 1 - o_p.$$

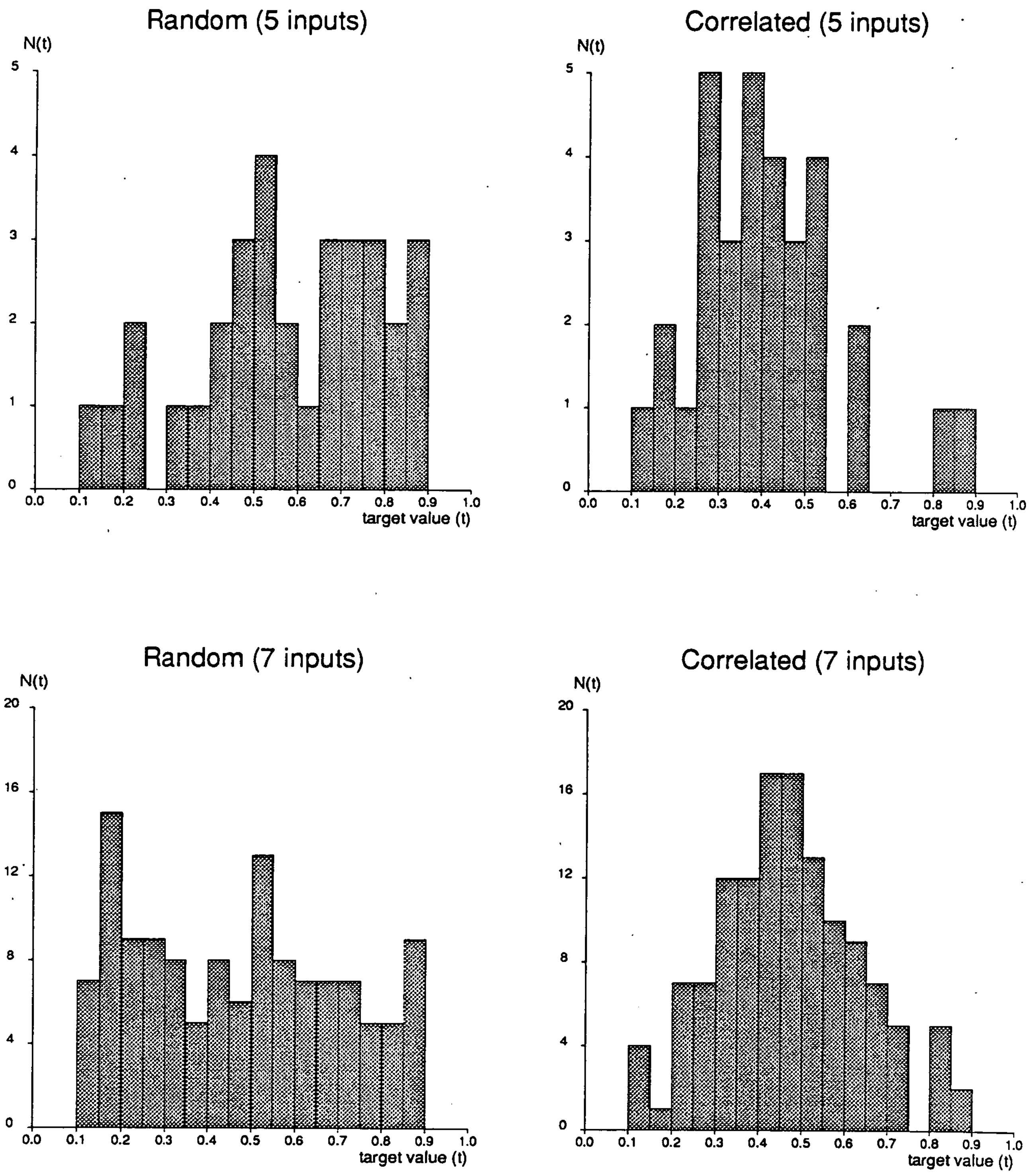


Figure 4.4: Example distributions of target values for the random and correlated/permuted-correlated training sets.

The Kulback [Kul59] distance between the target probability distribution (represented by a prime) and the network's distribution is

$$G = \sum_p \left\{ P_p'^{+} \log \frac{P_p'^{+}}{P_p^{+}} + P_p'^{-} \log \frac{P_p'^{-}}{P_p^{-}} \right\}, \tag{4.89}$$

from which we derive (4.88).

The measure (4.89) is related to the likelihood function used in information theory [BJM83]. Further instances of the use of distance measures similar to G can be found in [AHS85, Hin87, SLF88, PH86, Hop87, Wri88].

Although L has been derived from a distance measure designed for probabilities it can be used generally (and in fact is used here for this sole purpose) to measure the degree to which real numbers in the range [0,1] have been learned. The global minimum of the function is zero, when all the numbers are equal to the targets. Solla [SLF88] has tested such a cost function in learning the contiguity problem, to provide evidence that this "entropy" measure produces a generally more favourable (i.e. steeper with less local minima) error surface, as compared with the least squares error function.

This cost function is minimized by gradient descent of the surface defined by $L(\mathbf{w})$. Thus a weight w is changed on each update by the amount

$$\Delta \mathbf{w} = -\eta \frac{\partial L}{\partial \mathbf{w}} \tag{4.90}$$

where η is the step-size. The weight changes are calculated in the same way as in chapter 3. Inserting the cost function L for the E (indicating the sum of squared errors) in the derivations in section 3.2 the weight-change equations (3.25) and (3.31) become

$$\Delta w_{0j}^{H} = \eta \sum_{p}^{N_p} \left\{ (t_p - o_p) H_j^p \right\} \tag{4.91}$$

and

$$\Delta w^I_{jk} = \eta \sum_p^{N_p} \left\{ (t_p - o_p) w^H_{0j} H^p_j (1 - H^p_j) I^p_k \right\}, \tag{4.92}$$

where $o_p \equiv O^p_0$ and $t_p \equiv t^p_0$. The response functions are the same for hidden and output nodes, and are sigmoids in the range [0,1]. Note that factors from the differentiation of the logarithm and the differential of the response function $(o_p(1-o_p))$ cancel out to produce a simplified expression for the output δ's.

4.7 Implementation on transputers

The simulations in this thesis were all performed on the Meiko Computing Surface, a modular, reconfigurable array of transputers. This is an MIMD (Multiple Instruction, Multiple Data) machine, manufactured by Meiko Ltd., and run in Edinburgh by the Edinburgh Concurrent Supercomputer Project (ECS).

4.7.1 The transputer

The T800 transputer is a single VLSI chip that combines processing power, memory and communication links for direct connection to other transputers. It contains a fast integer and floating-point processor and can be used as a building block for even faster parallel processing systems, ranging from embedded systems to supercomputers. Each transputer has four bi-directional links through which it can communicate with the other transputers in order to transfer data and receive instructions for new operating modes (if this is necessary, since each transputer is able to work in "stand-alone" mode, processing its own data, running its own program). Each transputer in a system (transputer array) uses its own memory and can address up to 4 GBytes of off-chip memory with a 25 MByte/s band-width.

The transputer implements the process model of concurrency embodied in the high level language occam. In occam, communication between parallel processes is effected by uni-directional channels, which may connect processes on the same or different processors. Each transputer link implements two such channels, one in each direction.

4.7.2 Parallelizing the neural network model

There are various ways of implementing neural network models on parallel computers. The Ising-like Hopfield model is ideally parallelized by simultaneous bit manipulation, since update consists of all the neurons adopting a new (boolean) state, summing all the contributions from the other neurons in the network at that time. Hence this is best implemented on the computer which is almost designed for Ising model simulations, the ICL Distributed Array Processor, an SIMD (Single Instruction, Multiple data) machine (see, for example, [Wal87a]). If more compute-intensive calculations are required at each neuron however, a larger grain parallel computer may be more useful, such as the Computing Surface. The only drawback in using such a large grain computer as a transputer array to simulate neural networks, however, is the problems which arise with the high connectivity which may be necessary in such models. An image restoration program [For88], which was based on a Hopfield-type network, did not have such problems when implemented on a transputer array, because the connectivity of the network was sparse and well structured, so that the transputer array could be mapped onto the network almost directly, with each link handling the boundary information required for each transputer. The problem becomes less than trivial when near or total connectivity is required, combined with a learning algorithm requiring near global information for each neuron on each update.

Such are the characteristic problems associated with the feed-forward network and its backpropagation algorithm. Implementations include splitting up the neurons on different transputers and using sophisticated communications schemes to transfer data quickly when it is needed, [Wol88, BD87, Smi87], splitting the matrix multiplications over a number of transputers [Ric88], splitting the patterns in the training set over a number of transputers, with each transputer retaining a complete copy of the network in its memory, and combinations of the latter two.

Because of the way in which the number of patterns used in most simulations in this thesis scales with the size of the network (exponentially or faster), and the associated limitations on the actual useful sizes of network which could be implemented, the training set division method of implementation was used. In the protein simulations, however, and also the noisy digit simulations (section 3.6.3), the networks were large and the training sets comparatively small, and so a simulator embodying both the matrix multiplication splitting and the training set splitting was used (see [Ric88]).

4.7.3 Transputer configuration

Having decided on the method in which the model is to be partitioned on the transputers, the *configuration* of transputers which is to be used must now be determined. As in all such implementations, the important quantity to be minimized is the overhead

$$O := \frac{\text{transputer idle time}}{\text{transputer busy time}}. \tag{4.93}$$

A transputer is *idle* when it is waiting for data in order to perform its next stage in the calculation. This data comes through the links connecting the transputers, and if the configuration is such that messages take a long time to reach some, or one particular transputer, the entire calculation is slowed down, with valuable compute time being wasted. Ideally, if communications cost nothing, the training set parallelization should allow a linear speed up of the execution time with the number of worker transputers being used. Practically, communication does cost, firstly through the actual time spent sending the packets, and more importantly, through the increasing effective distance of transputers as the array size increases (due to the transputer having only 4 links).

A suitable configuration used in these simulations was the *binary tree*, shown in figure 4.5, where each transputer has two "children". The information about the network weights is passed to each transputer in the tree from the top after each update cycle, and each transputer has stores a (different) subset of the total training set. The gradient changes for each subset of patterns is accumulated

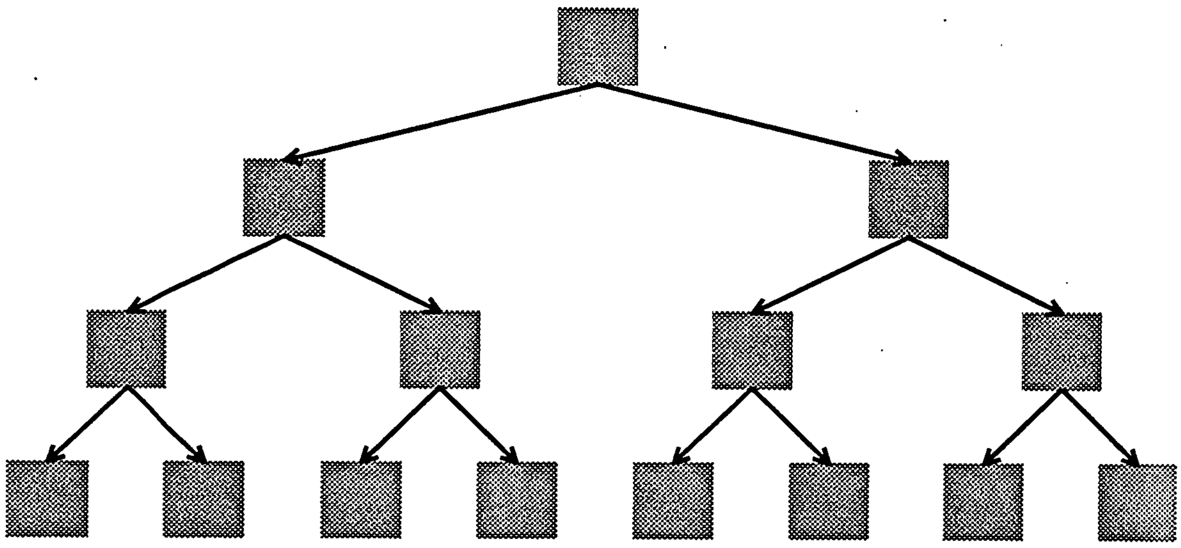


Figure 4.5: Binary tree configuration for transputers: each block represents a transputer, with the actual transputer links used indicated by arrows (these are 2-way links).

on its route to the top of the tree, where the weight changes are made. This was compared with a chain configuration, with the same calculations performed by the transputers. The number of links through which a message has to pass in order to reach the master transputer (where the weight updates and parameter variation are performed) scales linearly with the number of (worker) transputers N_t in the chain, but proportionately to $\log_2 N_t$ for the tree, for full trees. Thus, if the transputers are to spend a relatively short time computing before communicating data, the chain configuration will suffer much faster than the tree configuration. Clearly, there are several factors influencing overall speed, and we demonstrate the scaling of speed with array size and network size (with the number of training patterns scaling exponentially with the input layer size) for the two configurations, in figure 4.6. We show graphs of the time (in units of transputer ticks) taken to perform one learning epoch ($t(N)$) against the reciprocal of the number (N_t) of worker transputers being used, multiplied by the time taken using one transputer ($t(1)$). A straight line of unit gradient indicates linear scaling of the program's execution time with the number of transputers available, which is the upper bound on parallelization gain. The tree configuration manages to achieve linear scaling for more than about 2 workers, but the gradient is below the optimum. This improves with system size, however, and is altogether much better than the timings for the chain configuration.

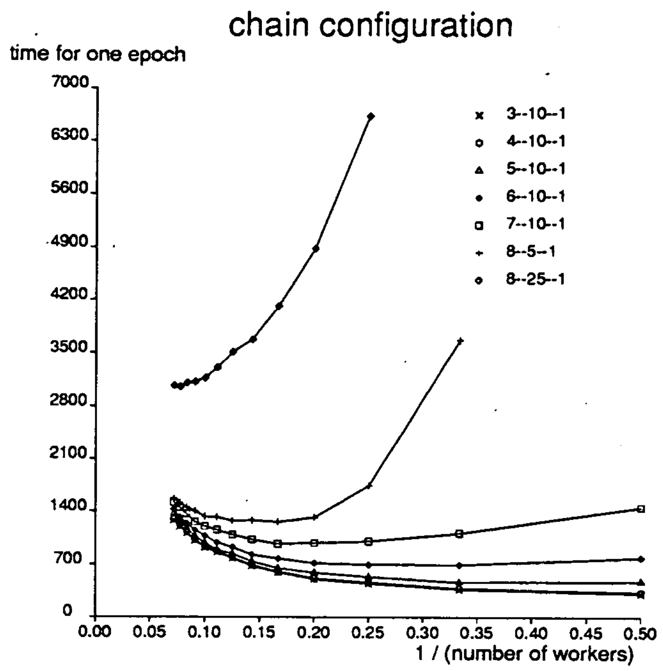
Another important consideration is the number of slaves which should be used for any particular network simulation. This is important because of the way the statistics were gathered. If a number of different size simulations are going to be performed on the same series of runs, then the number of worker transputers used should be optimized such that the correct number of workers are used at any one time. Such considerations are especially important for the smaller net sizes. In order to get this worker optimization the first few epochs of update were each performed on different numbers of slaves, and the actual time recorded. After this exhaustive search, the number of workers producing the fastest time was used. Such optimization was done each time the system size changed. The ease with which this could be done was a direct result of the parallelization of the model that was used.

In order to make full use of the number of transputers available, even when adding more transputers for the net size in use would actually slow the performance down, the network of transputers was replicated a number of times (this number depending on the particular number of transputers available at the time). Thus several trees, each with each own sub-master was allocated network simulations by the master, and on completion was immediately given another, and so on until the whole series of simulations was complete. Clearly in this higher level of parallelization the gain factor scales linearly with the number of replicas, and thus is optimal.

4.8 Observations of the learning performance

4.8.1 Details of the learning schedules

Due to the existence of real target outputs, it was not a simple matter to determine when the training was completed. Furthermore, it is not unreasonable to assume



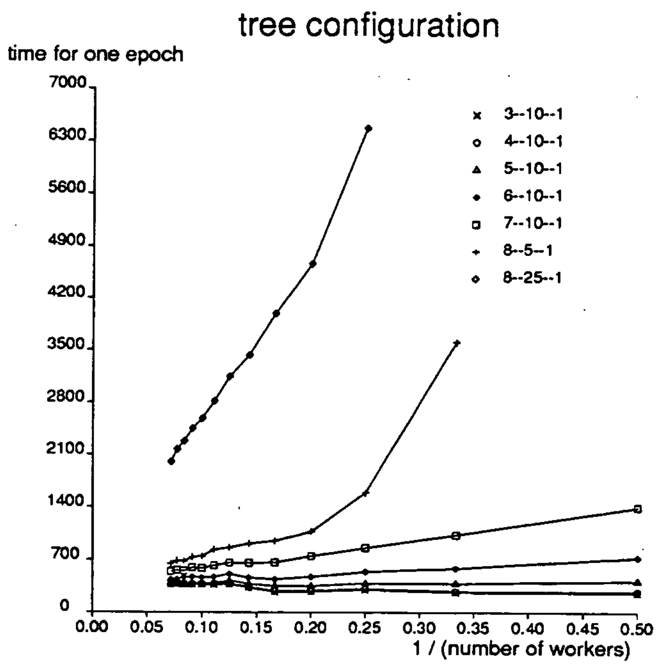


Figure 4.6: Time taken to perform one cycle vs. the number of worker transputers available, for the chain and tree configurations.

that, given enough hidden units, the network would asymptotically reach any desired accuracy for the target mappings. Therefore, it was necessary to curtail the learning at certain points, which were the same for all the training sets.

All the runs were terminated after 5000 cycles, whether finished or not. A run was considered finished when the value of L reached below $10^{-4} \times N_p$, which represents an average of better than two percent accuracy per pattern, for the range of values used (i.e. $[0.1, 0.9]$), which would in practice be near enough to distinguish the outputs. It turned out that greater accuracy was indeed superfluous, giving no extra information about the relative performance of the three types of training set or the generalization behaviour.

The learning parameter variation procedure outlined in section 3.7.4 (method B) was used in the simulations in this chapter, and so the learning basically consisted of finding the nearest minimum as quickly as possible. In such a procedure it is often going to be the case that the system becomes stuck in local minima, rather than finding a solution. Such occurrences cannot be foreseen, and so the best way to proceed is to take a sample of the best runs. First, between 5 and sometimes 20 runs were made on the same data, starting at different points on the error surface. The weights were initialized randomly (from a uniform distribution) in the range $[-0.5, 0.5]$ each time. This increased the chances of the network finding a good solution. The best out of these runs was used for the next averaging procedure. The second sampling was of the different examples of the *same* training set *type*. Any number of training sets can be generated for both the random, correlated and permuted-correlated type, simply by initializing new values of $\{I_i\}$ and $\{J_{ij}\}$ for the correlated and permuted-correlated sets, and new values of $\{b_p\}$ for the random set (these too being chosen from a uniform distribution). The actual values of the parameters $\{I_i; J_{ij}\}$ are not important, what is important is that they form a set of common parameters linking all the members of the training set, whereas the random set members are as independent as the random number generator allows.

Thus averages of the best solutions (the first sample) for *each* of the example training sets, which numbered 5 (although for the smaller system sizes it was possible to take larger samples), were taken to give an average performance for

parameter	*value*
initial η	0.6
initial α	0.0
η_f	1.1
α_f	0.1
small jump	0.001
Γ	0.005

Table 4.3: The parameters used in method B parameter variation procedure.

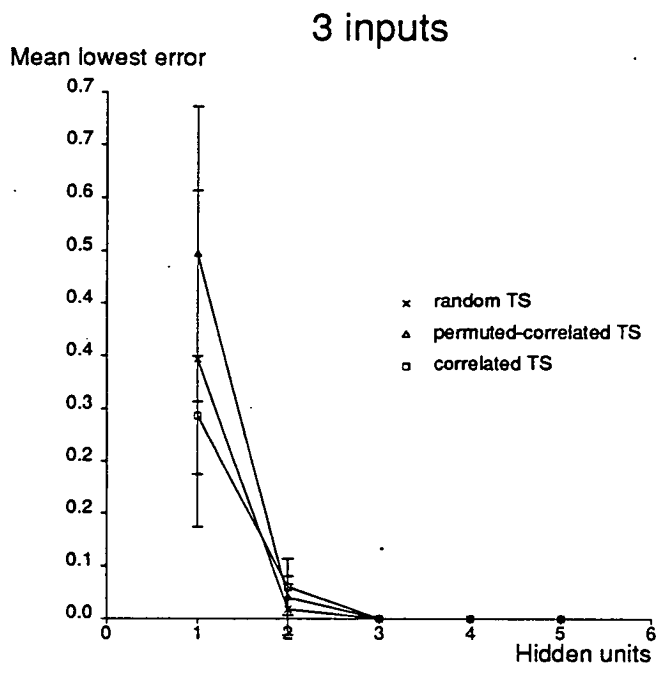
each of the random and the correlated training sets. Such averaging was done for most of the points plotted in this section.

The values of the parameters used in the learning parameter variation are listed in table 4.3, and were chosen after some preliminary runs, although they are in general (apart from Γ) not too crucial for the learning speed.

4.8.2 Scaling of the learning ability with hidden layer size

Figures 4.7, 4.8 and 4.9 show the average value of L ("mean lowest error") after learning (i.e. after at most 5000 epochs) for system sizes 3–N–1, 4–N–1, 5–N–1, 6–N–1, 7–N–1 and 8–N–1, for the two types of training sets, as the number of hidden units is increased. For the small system sizes (3–N–1 and 4–N–1), generally the performance of the random permuted-correlated and correlated training sets is comparable, the large error bars making it impossible to make any definite distinction between them. However, for larger net sizes the differences begin to become clear. The correlated set performance is significantly better than the permuted-correlated and random set performance, and this difference in performance increases with the net size. We notice that:

1. The net finds it easiest to learn the mappings of the correlated set, with the permuted-correlated and random sets significantly worse.
2. The number of hidden units the net requires to learn the mappings well increases with the size of the network, but much faster for the random and



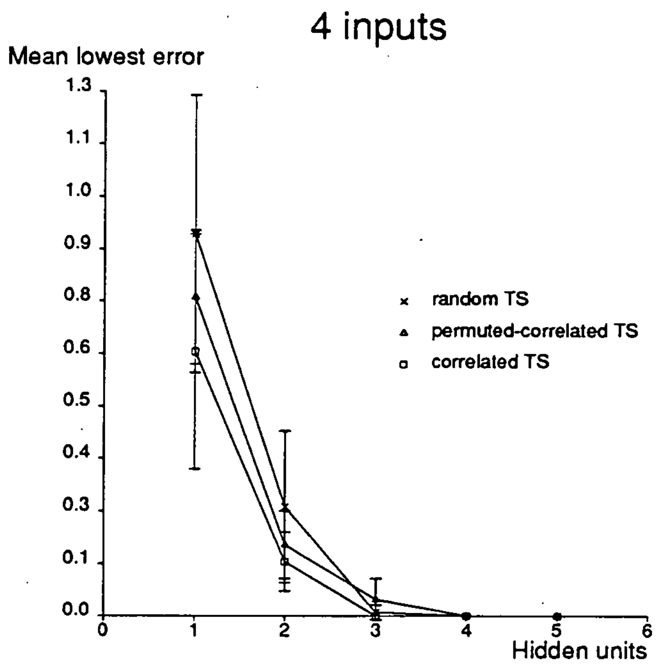
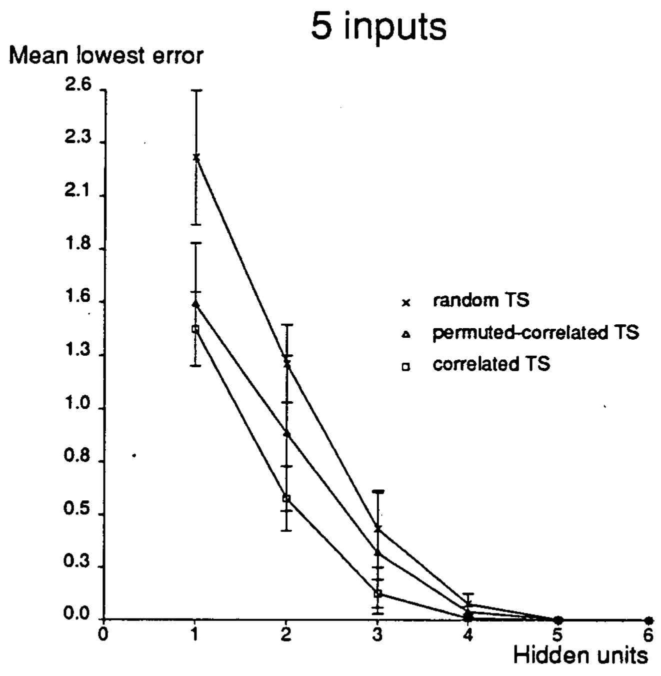


Figure 4.7: Scaling of the performance (final error) with hidden layer size, for the three types of training set, for input layer sizes 3 and 4.



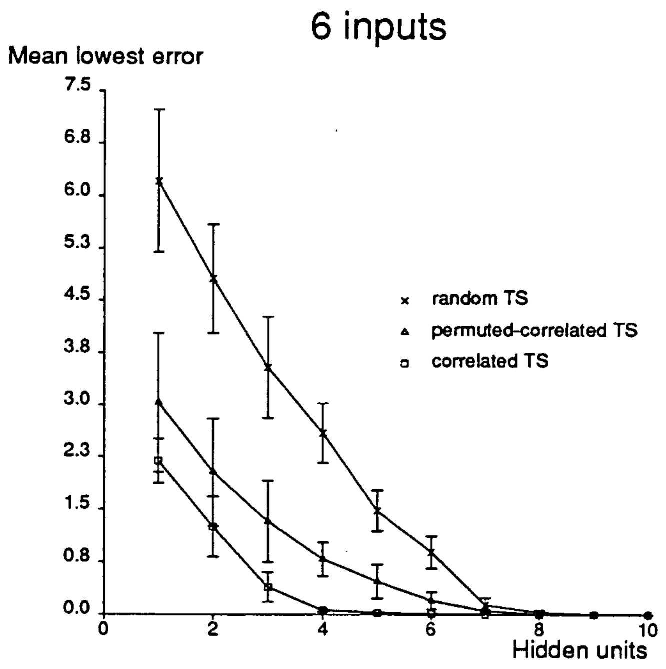
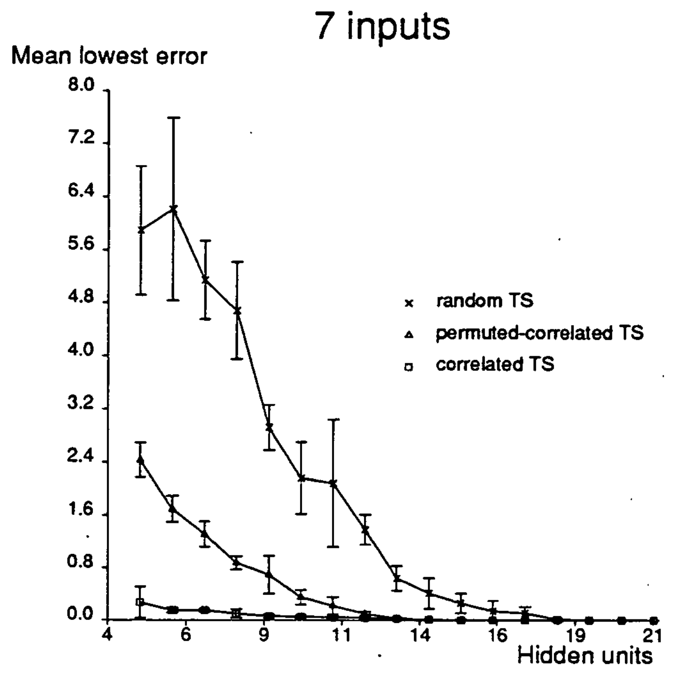


Figure 4.8: Scaling of the performance (final error) with hidden layer size, for the three types of training set, for input layer sizes 5 and 6.



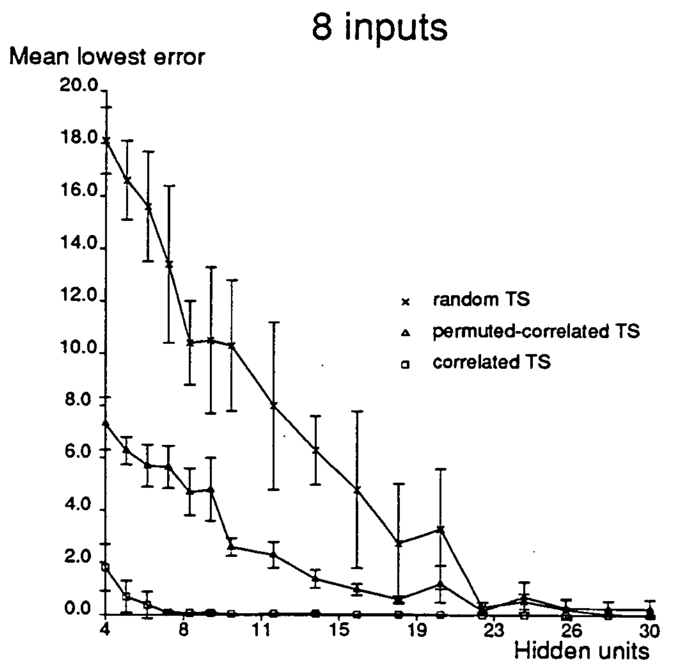


Figure 4.9: Scaling of the performance (final error) with hidden layer size, for the three types of training set, for input layer sizes 7 and 8.

permuted-correlated training sets than for the correlated.

We suggest reasons for this behaviour in section 4.10.

It is instructive to compare the performance scaling with the hidden layer size, with the scaling of the number of patterns to be mapped vs. the number of independent variables which are available for their representation. We assume that the important quantity in the scaling relation is the number of weights from the input to the hidden layer, since the weights from hidden to output perform in effect just an N_H–dimensional to one-dimensional projection of the hidden unit representations to the output node, thus adding nothing to the actual representational power of a network. Then the number of active parameters for the representation of the patterns is $N_I \times N_H + N_H = N_f$, including the threshold weights. If we now assume that the mapping problem is solvable only if the ratio of the number of free parameters to the number of independent output values which need to be reproduced (N_p) is greater than some critical value f_c, then for a solution to be possible:

$$\frac{N_f}{N_p} = \frac{N_H(N_I + 1)}{2^{N_I}} \geq f_c. \tag{4.94}$$

Whatever the value of f_c is (probably of order one), it will be the same for any value of N_I and N_H, and so we have an estimate of the scaling behaviour for a machine which is simply storing the values of N_p independent binary variables in a complex system of $N_I \times N_H$ free parameters. This provides us with a yardstick with which to gauge the network's learning of the random, permuted-correlated and correlated training sets. If the random training set were truly random then, for large enough numbers of required associations, we would expect its scaling to be the same as that suggested above. Figure 4.10 shows the critical number of hidden units required to solve the mapping problems, taken from the scaling graphs referred to above, compared with a line representing the worst case of completely random associations, with a value of unity assumed for f_c, at the critical hidden unit number.

It can be seen how both the random and the permuted-correlated training set scaling with problem size approximately follow the worst case theoretical line,

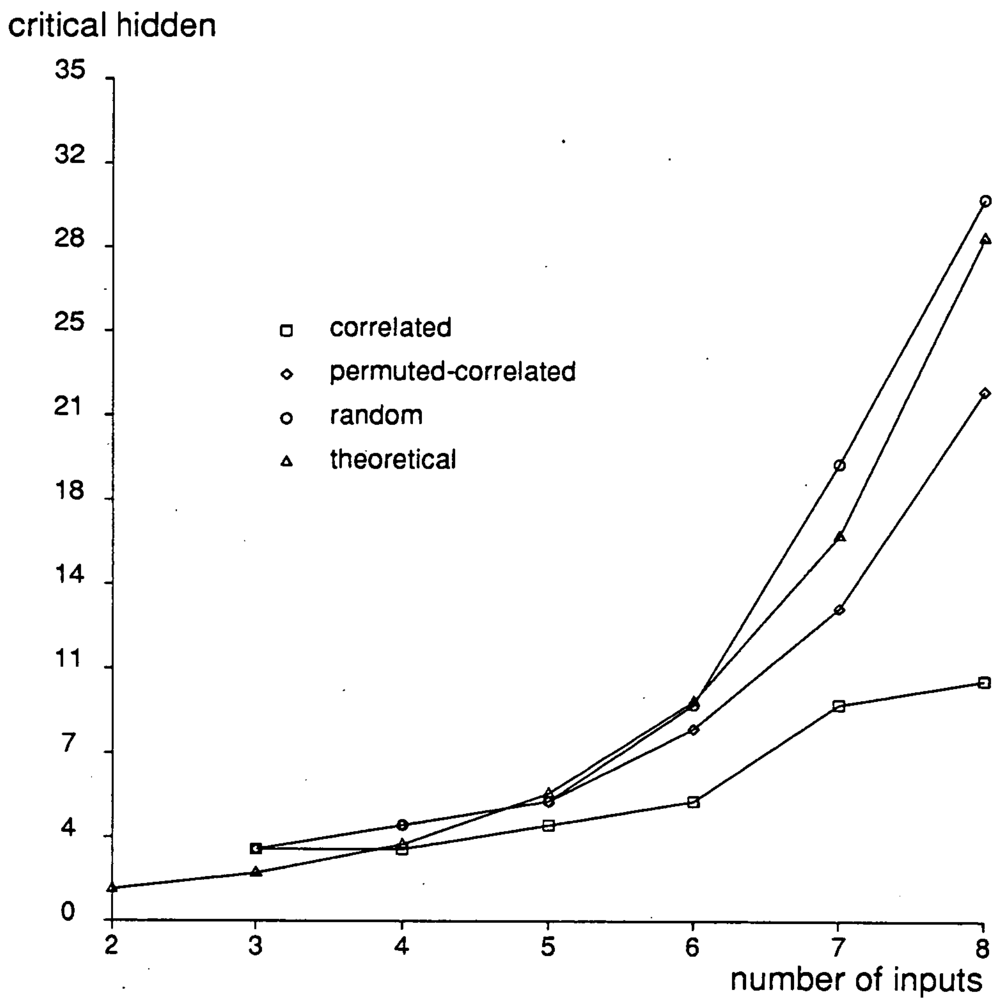


Figure 4.10: Scaling of critical number of hidden units with input layer size, for the three types of training set, and a theoretical scaling line (see text).

while the correlated set, despite starting near to this line, as would be expected for small problem sizes (since the number of free parameters is comparable to training set size), soon pulls away as the size is increased, showing a much slower rate of increase in critical hidden unit number.

4.8.3 Form of the learning curves

The learning curves are also helpful in evaluating the learning performance, and examples are shown in figure 4.11 for each of the training sets, for the network size 6–12–1 (at which all training sets can achieve good mappings), for five different training sets examples each.

It can be seen that the correlated training set performs once again significantly better than the other two, at a hidden layer size which enables all the training sets to find solutions easily. The descent is much quicker and steeper; the error decreases very rapidly at the start and reaches a very low value at about 600 epochs for all examples, as compared to the value of 900 epochs for the random, and even longer (actually about 1200 epochs) for the permuted-correlated.

4.9 Observations of generalization behaviour

Experiments were performed to determine the degree to which the network had "understood" the information about underlying trends provided by learning only some of the full training set, by examining the generalization on the patterns forming part of the complete training set family but which were excluded from the training set. The experiments done in this section concentrate on the comparison between the generalization performed on the correlated training set and the control system of the random set, in order to establish that the generalization is real. Chapter 5 is concerned with the scaling of generalization behaviour on similar training sets.

Three areas of generalization were studied: over-learning, learning times, and

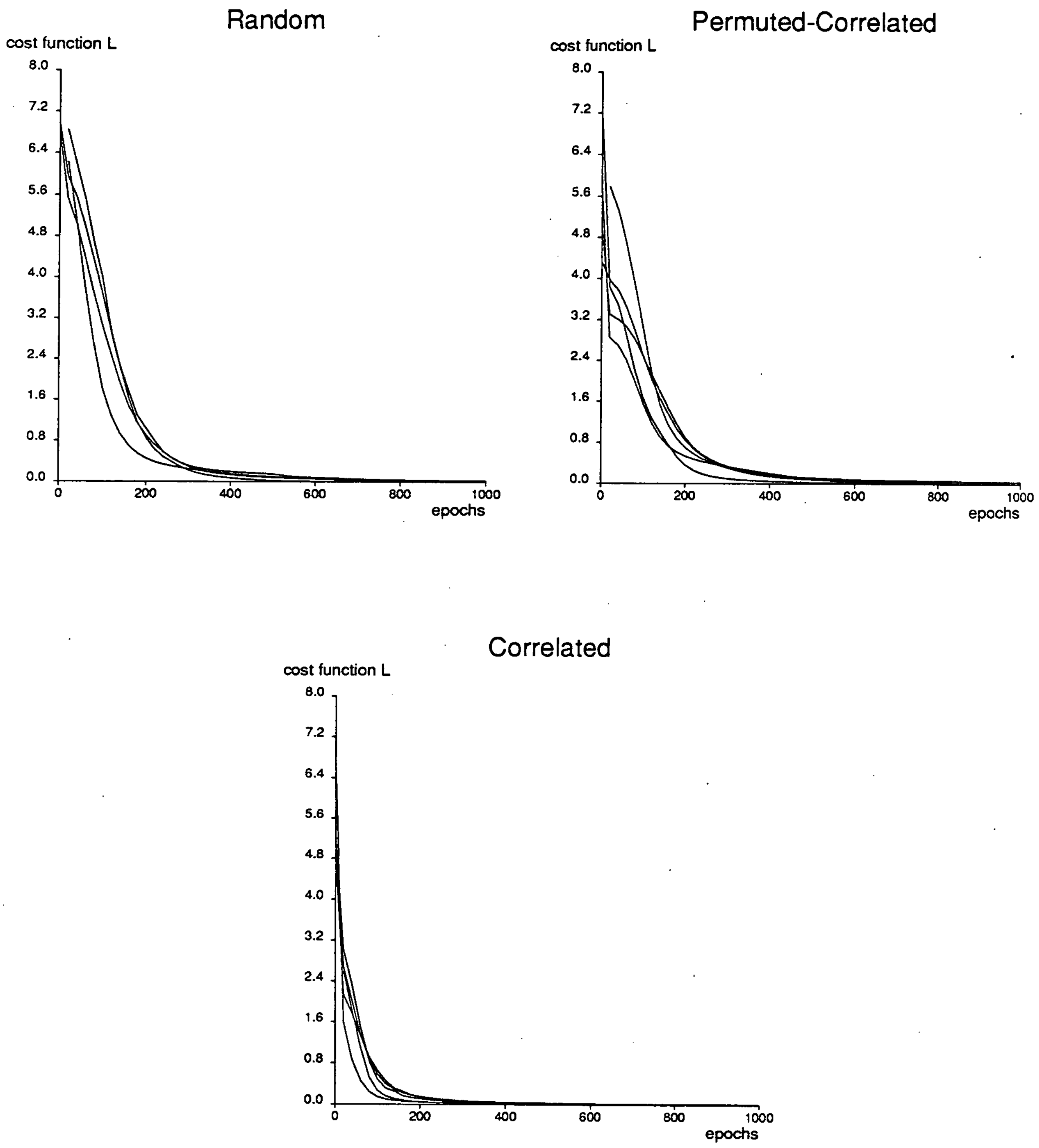


Figure 4.11: Five examples each of learning curves for the three types of training set at problem size 6–12–1. Note the scale is the same for each graph.

bootstrapping.

4.9.1 Over-learning

If the random training set is truly random it should show no generalization behaviour at all (on average). A characteristic of the natural training sets, such as the proteins in section 4.3, is that often better generalization is found when the learning of the training set is not done to too high a level. That is, if the patterns in the part of the family used as the training set are learnt too well there is the danger that the rest of the family will not be guessed as well as if the network had a more general feeling of the family.

In figure 4.12 are shown 3 examples of over-learning occurring with the correlated set. One line represents the value of L' for the training set being used, which was always the complete family (2^{N_I} members) less one (randomly chosen) pattern, for which the value of L' was plotted alongside. The prime on L indicates the value of L *per pattern* in the training set, or L/N_p. This allows more useful comparison. The first part of the learning displayed shows the generalization proceeding well for all examples — the error of the absent pattern decreases as the error for the other patterns is reduced. However at certain points in the learning, which depends on the absent pattern, the error starts increasing to a greater or smaller extent again. Occasionally it is found that the error for the missing pattern remains stable to the end (bottom example in figure 4.12).

In figure 4.13 we show similar curves for the random and permuted-correlated sets. We do not expect any over-learning to be found, and what is actually seen is a fairly controlled movement to a final error of a large value. (Note for these graphs the large difference in the y-axis scales compared with the previous figures). As the rest of the training set is learned there is no reason for the missing pattern to cause an output anywhere near its target output, and this is reflected in the graphs. The end points of the lines reach up to about $L' \sim 1$. The maximum a pattern might be incorrectly mapped (0.1 for 0.9 for example) gives an L of about 1.75. The same scenario was observed for the permuted-correlated net, although very occasionally, as can be seen from the bootstrap curves in section 4.9.3, a pattern

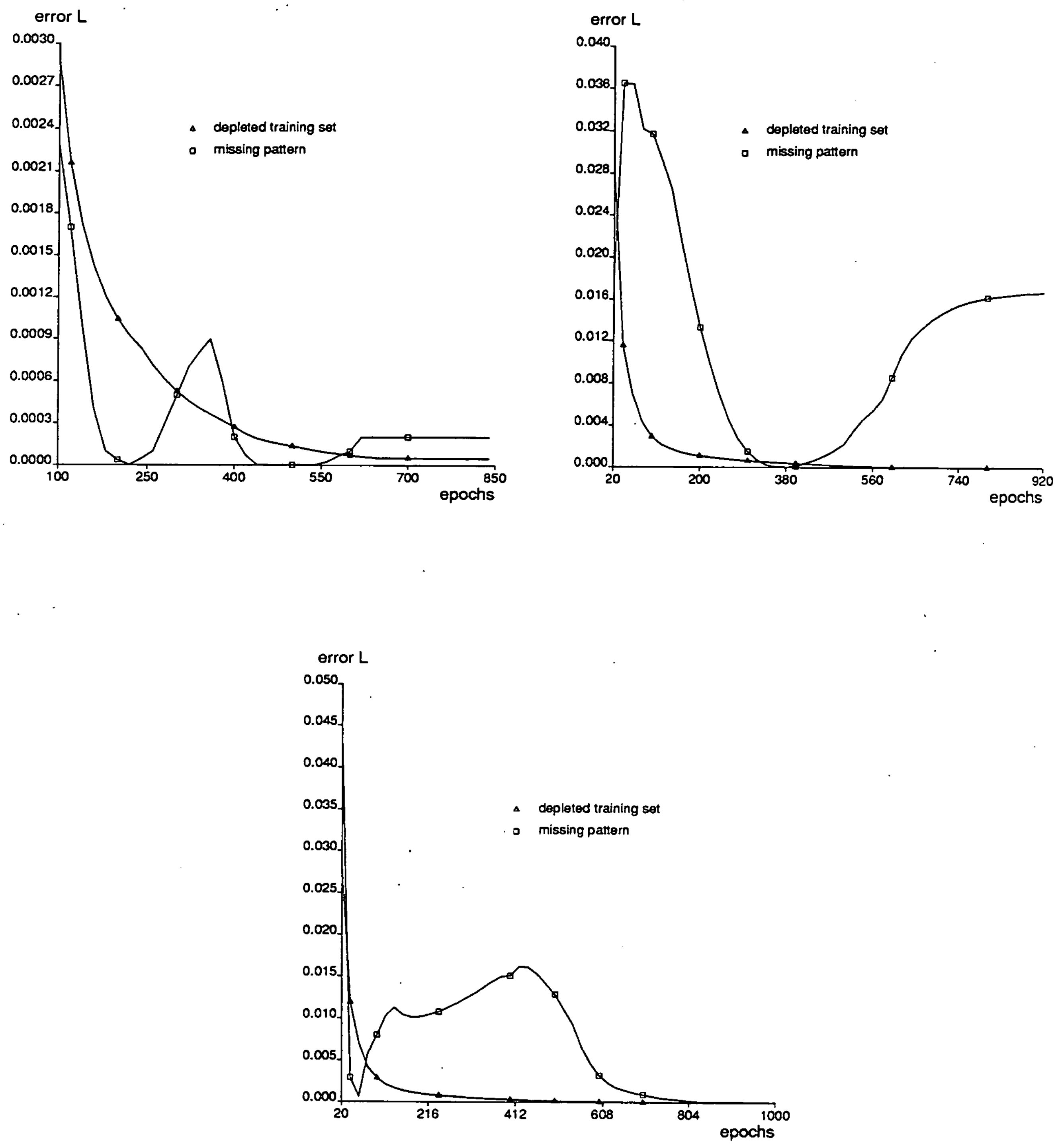
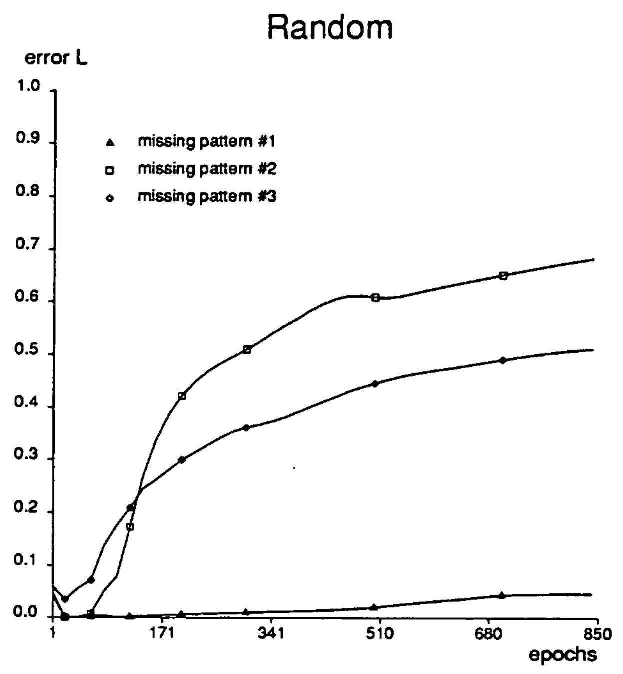


Figure 4.12: Three examples of the variation of the error of missing pattern (monitored alongside the training set error per pattern) for the correlated training set The network size is 6–10–1.



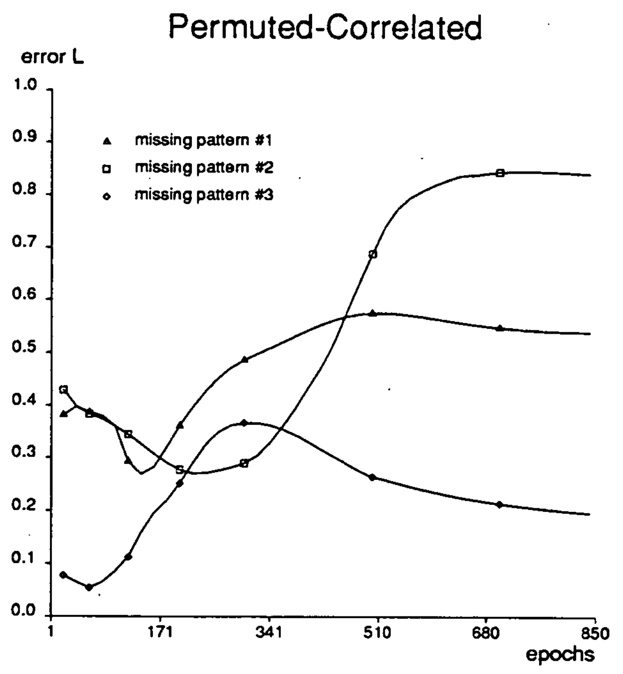


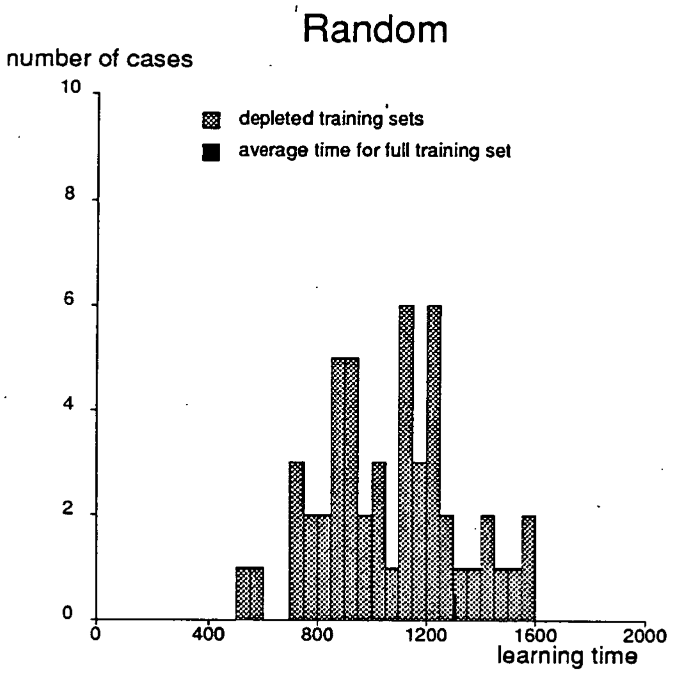
Figure 4.13: Three examples of the progress of error for the random and permuted-correlated sets for the missing pattern, for the system 6–10–1.

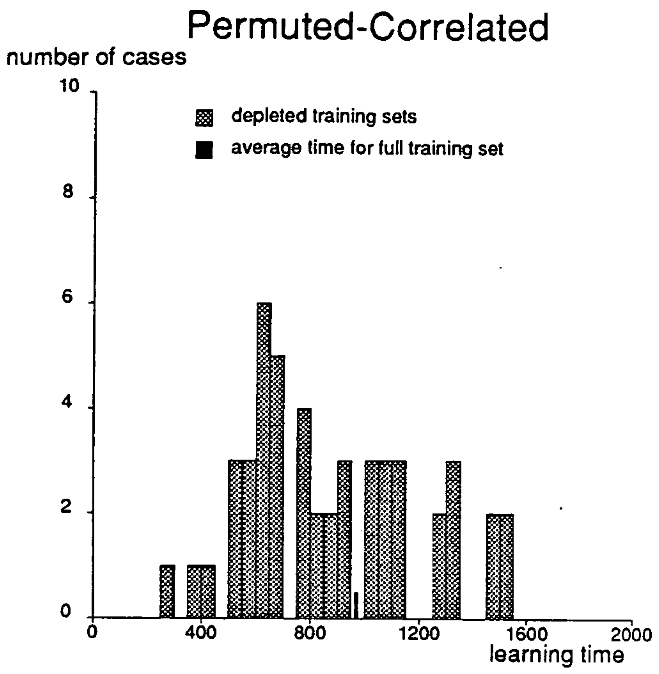
can be generalized; however, this is more to do with a consistent distribution of target values rather than underlying trends. The extent of the overlearning in the correlated net is much more widespread than in either of the other types of net.

4.9.2 Learning speeds

It might be imagined that removal of a pattern from the training set would "lighten the load" of the mapping, thus making it easier, and therefore allow the mapping to be performed (assuming it was possible with all the patterns) in less epochs than before. This should certainly be the situation for the random training set, since the more the mappings the harder it is to find the correct representation for the independent patterns in the weights. Figure 4.14 shows the distribution of learning times for the 6–12–1 system, for several weight initializations, using ten randomly chosen examples of the 64 possible one-pattern depleted training sets. The small black bar indicates the average learning time using the complete set of patterns.

It can be seen how the distribution is peaked to the left of the black bar for the random and permuted-correlated training sets, showing that, as expected, pattern removal improves the learning time in general. In fact, there is on average an 18% decrease in learning time for the random set. With the permuted-correlated training set removal of a pattern on average improves learning time, but less markedly than with the random set. The average time is reduced by 3.4%. This reduction in improvement can be put down to the adverse effect removal of a pattern has on the fast initial learning of the general training set distribution (as in figure 4.4). Conversely, for the same experiment with the correlated training set, shown also in figure 4.14, removing a pattern has little or no effect in the learning time, on average. The average percentage change in learning time in removing a pattern from the training set is actually a 1% *increase*. Thus the difference between this value and the 3.4% decrease of the permuted-correlated set, shows that cooperative weight-updating is an important feature in the learning of the correlated training set, all other things being equal.





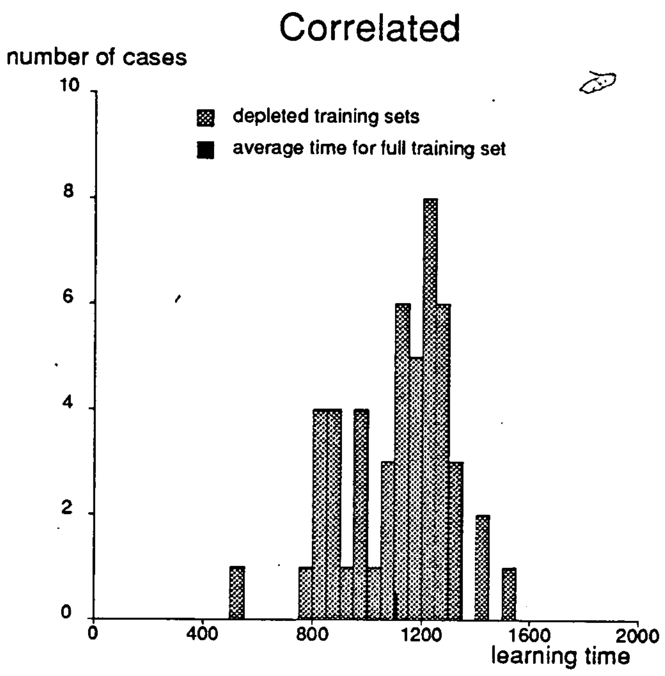


Figure 4.14: Distribution of learning times for the three training sets, with one pattern removed. The black bar indicates the average learning time for the full training set.

4.9.3 Bootstrapping

Bootstrapping the training set means taking each of the patterns out in turn (one at a time), learning the rest of the training set, and after learning asking the net to provide a guess as to the value of the missing pattern (by processing this pattern). If any correlations exist among the patterns in a complete training set, the learning of a majority of these patterns should provide enough constraints to suggest reasonable values for the others.

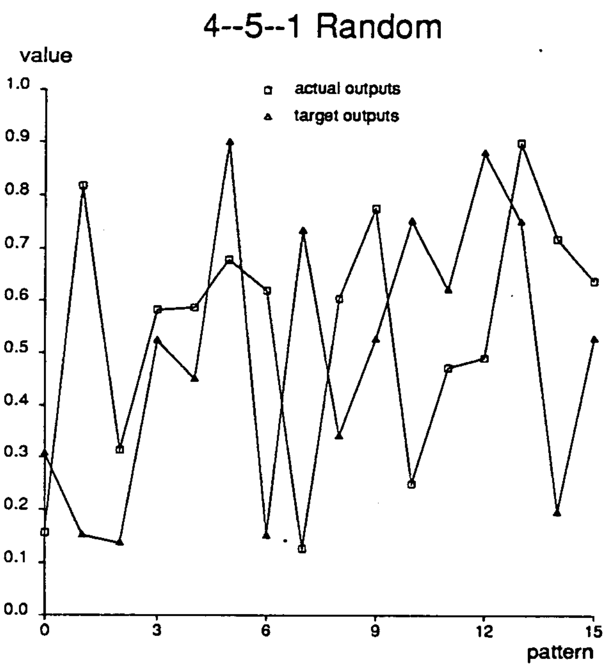
Figure 4.15 shows the bootstrap curves for the three training sets, for a net size of 4–5–1, for which there are 16 patterns in all. If the points are joined up then we notice that the target curve and actual curve compare quite well for the correlated set, actually far better than chance guesses. This can be seen by comparison with the random bootstrap graphs for the same system size. We can put a number to the bootstrap performance by defining the error in fit F of the two curves in the following way:

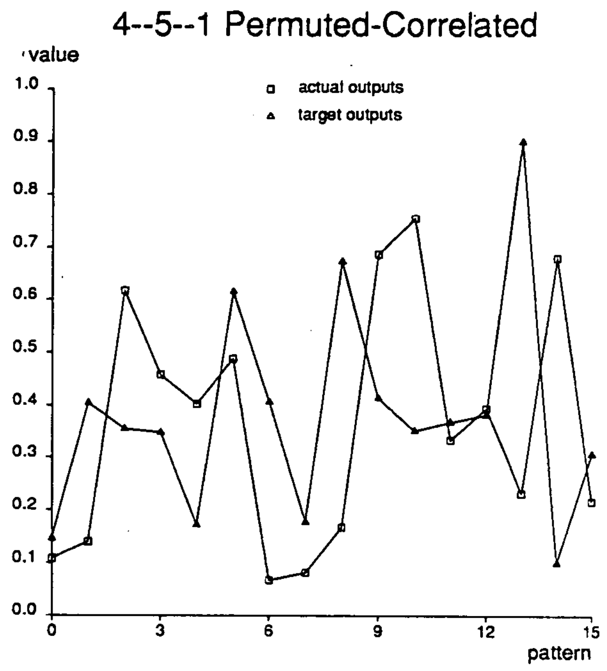
$$F := \frac{\sum_i (x_i - y_i)^2}{N_p} = \left\langle (x-y)^2 \right\rangle . \tag{4.95}$$

The worst value F can have, if the two curves are independent, and the points are chosen from a uniform distribution in the range $[0,1]$, is given by

$$\begin{aligned} F_0 &= \left\langle (x-y)^2 \right\rangle = \left\langle x^2 - 2xy + y^2 \right\rangle && (4.96)\\ &= 2\int_0^1 y^2 dy - 2\int_0^1 \int_0^1 x\ y\ dxdy && (4.97)\\ &= \frac{2}{3} - \frac{1}{2} = \frac{1}{6} && (4.98)\\ &= 0.167. && (4.99) \end{aligned}$$

Actually, the targets are always in the range $[0.1, 0.9]$, while the outputs are free to take on values in the range $[0,1]$, so the worst estimate is modified to 0.176. Any departure from this value indicates a greater or lesser degree of non-random behaviour. The values for the curves in figure 4.15 are 0.044 and 0.126 respectively for correlated and random bootstrapping. The permuted-correlated set scored 0.102 – almost as bad as the random net! The random and permuted-correlated nets effectively make guesses based on the distribution of target values (as was





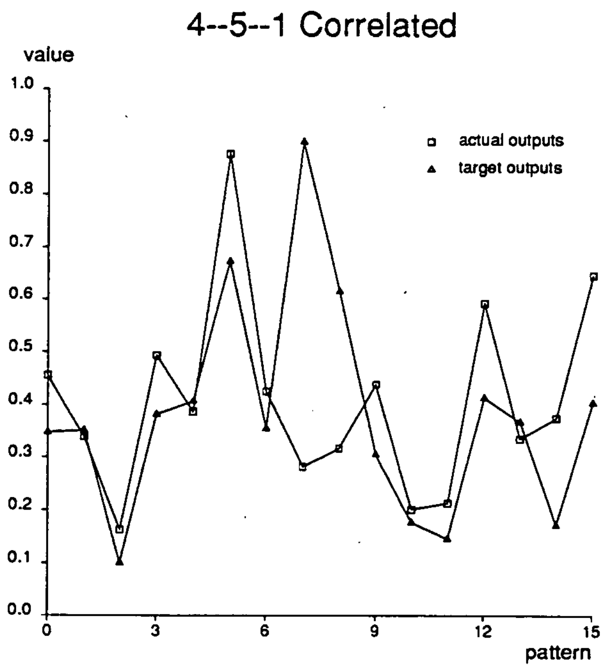


Figure 4.15: Bootstrap curves for the three training sets with 4 inputs (see text for explanation).

assumed in the estimate (4.99) for a uniform distribution). Since the permuted-correlated training set is peaked (see figure 4.4), the average error will be less than for the approximate uniform distribution of the random. Note also, that as the number of exemplars increases, we would expect the permuted-correlated performance to improve (since the target distribution becomes more peaked) while the random should reach nearer the worst case of 0.176 above.

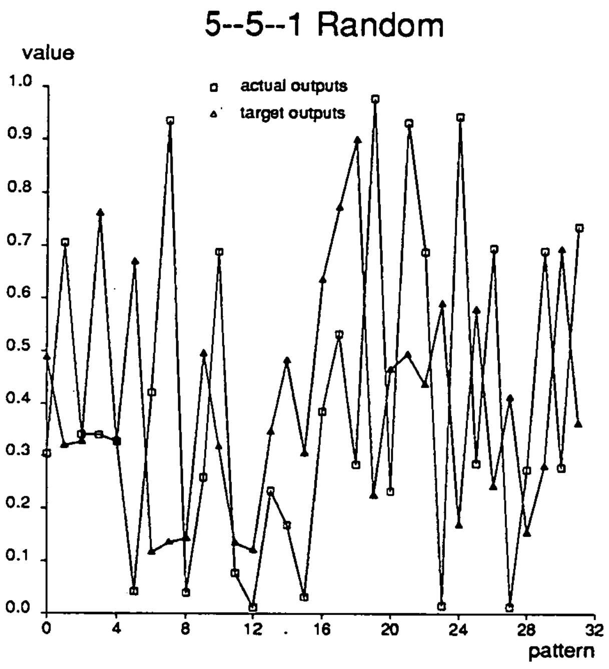
Figure 4.16 shows the situation for the 5–5–1 network. These are more impressive still, as would be expected from previous results for larger networks. The values of F for these curves are 0.161 for the random training set, 0.015 for the correlated, and 0.081 for the permuted-correlated.

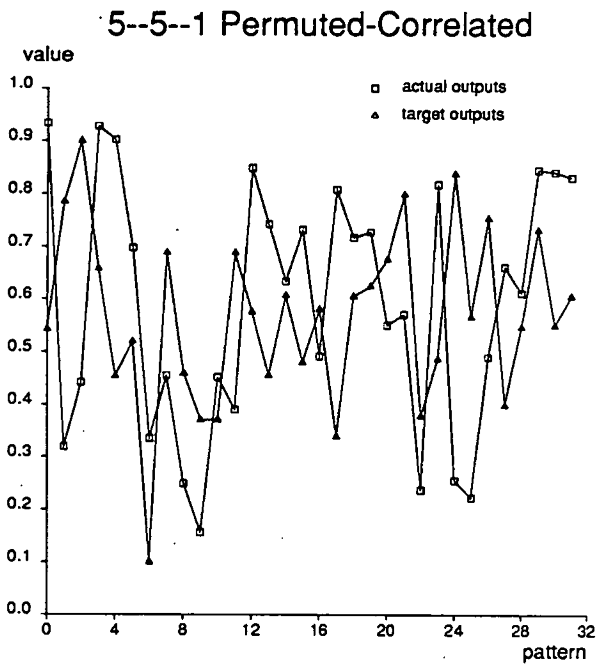
Figure 4.16 for the correlated set is particularly impressive when one considers that the network has not in fact received any knowledge about what is plotted *directly*, but merely through *picking up the underlying trends* in the training set and generalizing to a missing input. In this way, although none of the mappings on the graph was actually seen, the network can produce a remarkable guess at the complete set of target outputs.

For the 6—12—1 net, the correlated set produced an F of 0.006, the random 0.078 and the permuted-correlated 0.065. The improvement in the random performance is surprising, considering what was said above, but we would expect values of F averaged over many training set examples to tend to the predicted value. The significant result is the clear improvement in the correlated set behaviour for larger training sets. The consistent factor of more than ten better than the permuted-correlated provides clear evidence for the underlying correlations in the training set guiding the generalization.

4.10 Discussion and conclusions

The observations above indicate that there is a significant difference between the correlated training set and the other two. The main cause of this difference is the fact that there exists underlying regularity in the input/output pairs in the





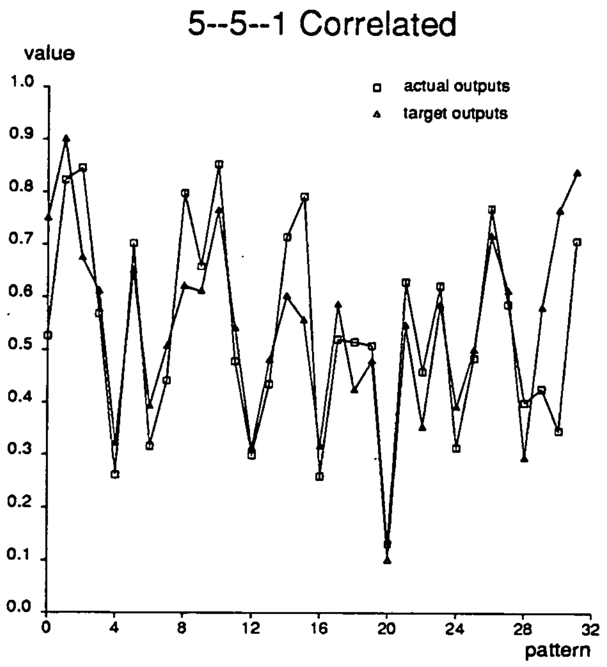


Figure 4.16: Bootstrap curves for the three training sets with 5 inputs (see text for explanation).

correlated training set, while such regularities are absent from the other two.

The regularities give rise to the better learning behaviour and the generalization behaviour.

The form of the learning curves. As was pointed out, the descent for the correlated training set was much steeper and reached low values much quicker, than either of the other two training sets. We believe this to be due to the *cooperative* effect of the δ terms in the backpropagation. If the mapping of pattern i is consistent with the mapping of pattern j, then not only will the error terms serve to reduce the error of the respective patterns, but the error term for pattern i will to a certain extent benefit the pattern j, and *vice versa*. Hence we see the cooperative effect in the learning curves, which is apparent even when all the networks have enough hidden units to be able to perform their respective mappings easily. By the same token, we find that the learning curves, when there are not enough hidden units to perform the mappings, differ in the opposite direction: the correlated set stops at a plateau very early on (after a fast initial descent), while the random sets decrease in stages of short plateaus, always reducing the error a little bit with each step, although tending to an error well above that for a solution, but also much lower than the correlated plateau. The correlated error terms, which call for similar conflicting (because of the number of hidden units available) weight changes, cancel each other out.

The critical number of hidden units. This property tells us quite emphatically of the presence of any underlying correlations. The fewer the number of independent parameters used in the definition of the patterns (which defines the extent of the correlations in the input/output pairs), the fewer independent parameters there are to be stored in the network (from a purely information theory point of view), and so the fewer the number of free parameters that are needed in order to achieve the mappings. We note how the scaling of good performance at a certain problem size, with the number of hidden units (which specifies the number of free parameters in the system), differs for the three types of training sets. The random training set in general has the worst performance, with the scaling proceeding almost parallel to the theoretical worst case. We would not in general expect the permuted-correlated set to have as bad an *absolute* performance as the random set, but nevertheless note how the *scaling* of the performance too follows the theoretical worst case for no correlations. The absolute performance is expected to be better because of the distribution of the target values. These provide the set with a pseudo-correlation through giving a higher occurrence for certain ranges of target values than others, and thus a free parameter may be used to store information for more than one pattern. It must be realized, however, that this is not real correlation between patterns and their targets, but solely between the targets. This is actually a relatively small effect insofar as the learning curves are concerned, with the permuted-correlated curves having similar general forms to the random ones. The form of the scaling for the correlated set is considerably better than the theoretical worst case. This type of non-exponential scaling is what should be expected from real problems of this nature. Academic problem domains are often characterized by such exponential scaling of hidden layer size. It should also be pointed out here that although the number of hidden units is a crucial factor up to the critical number, after this point further addition makes little difference in learning *speed* and the form of the learning curves. One may ask if any difference at all is made in real terms through further hidden unit addition. We explore this question in the next chapter.

We have discussed the learning above in terms of weight changes and cooperative behaviour, but it is also useful to move from the reference frame of the individual weights to that of the hidden units. This is also used in the next chapter to define a solution found by an MLP. The hidden layer can be thought of as defining a representational space for the input patterns. As the mappings are being learnt, the representations will change relative to one another. If it is possible, the representational stage of the mapping organizes the patterns such that it is possible to produce the correct output values. If the patterns are independent, the representational space will need to be larger, because, in the worst case, each input pattern needs to be represented as far apart from every other pattern as is possible. However, with correlated patterns, the space can be smaller because patterns can be much closer to one another, and indeed it is possible for them all to fit into a very small space and still achieve all the required relative representations for the very reason that they are self-consistent.

Over-learning. These observations indicate that at a certain point (the minimum) the net pays less attention to the general structure of the training set than it does to the specific requirements of the chosen patterns. This behaviour is a warning against the over-learning of exemplars, if a certain degree of general behaviour is desired at the outcome of training. We believe this is a result of the learning proceeding in two steps:

(i) the optimization heads towards the area in weight space where the global minimum for the *complete* training set exists, but,

(ii) after this level, the optimization heads towards the nearest minimum which is relevant for the *current* training set. This sub-minimum is (a) not necessarily the same as the one for the *complete* training set (i.e. that containing the whole family of mappings), (b) possibly different depending on direction of entry to this region of weight space, and (c) not necessarily in a sub-region of weight space which is closer to the full training set minimum than the starting point.

The existence of many solutions for an incomplete training set, which do not necessarily include a solution for the full training set, is thought to be a major problem in the search for "clever" artificial neural nets (i.e. ones that generalize on previous relevant but scattered information). We suggest in this thesis however, that maybe the search for "a good generalization" is a red herring, and that, given the right type of problem (usually possible through appropriate input codings) correct generalization is no problem. The over-learning observed here is minor, as we note from the bootstrap curves (plotted using the "over-learned" values of the outputs). The results show, however, that it is always a possibility, and care should be taken in not learning the training set so well that generalization performance is noticeably curbed (as in the protein example).

Learning times. With single pattern removal the average learning time of the training set is affected differently for each of the training set types. This has been discussed in the last section. We emphasize that for the correlated case, due to cooperative behaviour, removal of a pattern in general *slows down* the learning.

Bootstrap behaviour. This is one of the impressive displays of the generalizing power the MLP has. The curve built up through the network's "educated guesses" for single patterns, after learning all the others, so closely models the target curve for the correlated set, that we can see clearly how the network has correctly "stored", in its weights, the salient information about the training set.

With these positive results, the reduction made in section 4.4 can be fully justified, and suggests also that, having made the reduction and reproduced the main characteristics of the natural training sets, it might also be possible to say something about the *types* of correlations (we have used just pair correlations here) dominant in the particular natural training set the network is learning.

Chapter 5

Solutions and scaling of generalization ability

5.1 Introduction

In this chapter we investigate the effect on the generalization ability of an MLP of increasing the size of the hidden layer, using an artificial diagnosis problem domain. We use the relative hidden-unit representations of the training set patterns to provide a definition of an MLP **solution** to a mapping. This is then used to establish that the possible number of different solutions which a network may find, given a certain range of starting positions in weight space, increases very quickly with increase of number of hidden nodes above the critical number required to solve the problem. This being so, we ask whether the prospect for satisfactory generalization ability grows correspondingly poorer, as many more solutions are made available, each perhaps with its own set of generalizations. Results suggest that, for a wide range of hidden layer sizes, the generalization performance remains high.

Such findings on this artificial problem domain confirm what has been observed on many occasions with the low-level problem domains cited in section 4.1, and is encouraging, in that getting the "correct" number of hidden units may not necessarily be too critical a factor in the training of MLPs on real problems.

We illustrate the factors which may be more important in successful training, with a study of the way the generalization scales with the *number* of patterns included in the training set, and the variation of this with the actual patterns *included* in the training set. We note that the inclusion of some patterns rather than others may have a significant effect on the generalization performance.

5.2 Unique solutions to a mapping problem

The backpropagation learning algorithm is mathematically gradient descent of a surface in N_W–dimensional space, where N_W is the number of parameters specifying the MLP (the weights). Clearly for $N_W > 2$ it is no longer a helpful concept to think of the algorithm as some kind of flow down a hill. It would be more useful to know something about the distribution of different solutions populating the search-space. One way of understanding such a property of the search-space is to run the algorithm several times starting at different points on the $(N_W + 1)$–dimensional surface, and then to observe the occupation numbers of the different final network configurations obtained. However, one must first allow for various symmetries in the network states, since the mapping solution discovered by the algorithm may have many equivalent weight realizations. Before the class of symmetries in the N_W–dimensional parameter-space can be identified however, it is necessary to define what is meant by a mapping "solution".

In this section we offer a method of specifying uniquely the solutions found, and a way of counting the different solutions available to an MLP, for problem domains in which the solutions do not overlap.

In order to learn the correct mappings for several patterns at the same time, the network must choose a set of weights which allows the representations of all the patterns in the hidden layer to have the correct *relative relationships*. Depending on the type of problem domain, the patterns will require a greater or lesser degree of independence from one another. Given a certain number of hidden units, there will be a number of different ways of representing the patterns in order to achieve this. Each of these ways characterizes an *interpretation* of the problem domain (that is to say, how each pattern is related to every other pattern), and can be justifiably used to specify a *solution* to the mapping task in hand. There may be many different weight realizations of one particular solution, and so the way to proceed is to define a solution not by the weights found by an MLP, but by the set of relative representations of the patterns these give rise to.

Each dimension in hidden-unit (**hu**–) space is specified by one of the hidden units in the system. The space spans the range of values which may be taken on by the hidden units, in each direction. Thus each hidden unit defines an axis in an N_H–dimensional space, where N_H is the number of units in the hidden layer. Then each pattern is represented by a point in this space, or equivalently, by a vector $\mathbf{h}^p = (h_1^p, h_2^p, \ldots, h_{N_H}^p)$.

The vectors $\mathbf{h}^p$ need to satisfy the following requirements:

$$\mathbf{h}^p.\mathbf{o}_i = \phi_{pi} + \theta_i \qquad \forall \;\; p,\; i \tag{5.100}$$

where i labels output units, ϕ_{pi} is the target activation for output unit i on presentation of pattern p, and θ_i is the threshold value of output unit i. $\mathbf{o}_i$ is the *output vector* from the hidden layer to the output unit i. This vector is in fact the representation of the weights from the hidden layer to the output unit, as drawn in the **hu**–space, using the *axes* defined by the hidden unit states. Thus one can imagine various configurations of representation vectors $\mathbf{h}^p$ producing the correct set of scalar products as defined by (5.100), but which have different *relative* orientations and lengths. Hence a solution can be represented by $\{\{\mathbf{h}^p\}, O\}$, where the $\{\mathbf{h}^p\}$ specify the relative pattern representations, and O is the *output matrix* (matrix of all the vectors $\mathbf{o}_i$).

If the hidden units have states ranging from 0 to 1, the representation vectors are defined in the region of the N_H–dimensional hypercube of unit side with one corner at the origin and a diametrically opposite corner at the point (1,1,...,1), and if they range from -1 to 1, in a hypercube of side 2 with centre at the origin and a corner at (1,1,...,1). We need to determine the class of symmetry operations under which a solution will be invariant. If the vectors lie within the hypersphere of unit radius (we will assume the hidden unit states lie in the range $[-1,1]$ centred at the origin), then the representations will be invariant to the class $O(N_H)$ of rotations and reflections (orthogonal transformations) in

N_H–dimensional space. Global scaling operators also leave solutions invariant, if these operators keep the representations within the hypersphere. The same symmetry operation must always be performed on the matrix O in the case of the $O(N_W)$ class, and the inverse operation on O for the scaling class[1], in order to get correct values at the output of the net (however, this need not concern us here since we know that all solutions must satisfy this requirement by definition). Invariance to this set of symmetry operations allows sets of weight configurations for identical solutions to range quite considerably. In general, however, we cannot assume the vectors lie within the hypersphere mentioned above. Indeed, it is often the case that hidden units have values close to the upper and lower ends of the response function. If this is so we have to decide at what point two representations containing the *same* set of relative *angles* but different set of magnitudes are different. The case of the hypersphere was valid for linear response units, but not for the nonlinear case. Indeed, it is precisely the nonlinearities which afford the MLP the capability of performing arbitrarily complex mappings [Lip87]. Before we answer the above question, we define more specifically what will be taken as the parameters specifying the representations.

A representation set defining a solution, $\mathcal{R}$, is written as

$$\mathcal{R} = \{l_{ij} : i,j = 1,2,\ldots N_{TS};\ i<j\} \tag{5.101}$$

$$\text{where} \quad l_{ij} = |\,\mathbf{h}^i - \mathbf{h}^j\,|, \tag{5.102}$$

and N_{TS} is the number of patterns in the training (representation) set. Thus the representation $\mathcal{R}$ is defined by the set of distances $\{l_{ij}\}$ between all the points representing patterns in the hu–space, which contains information about the relative pattern representations. To allow for scaling symmetry we use the set $\{\mathbf{h}'^p\}$, which is the set $\{\mathbf{h}^p\}$ rescaled such that the condition

$$\begin{aligned} |\mathbf{h}^m| &= \max_p\{|\mathbf{h}^p|\} \\ |\mathbf{h}'^p| &= |\mathbf{h}^p|/|\mathbf{h}^m| \qquad \forall\ p \end{aligned}$$

is satisfied.

[1]The set of output states is given by $O(\mathbf{h}^p)^T$, and remains invariant when the pattern representations $\{\mathbf{h}^p\}$ are scaled globally, provided O is scaled inversely by the same factor.

Provided the combination of relative angles and relative magnitudes are not permitted to vary more than the differences in these quantities between different solutions, it is possible to compare solutions unambiguously. Actually, the more output units there are to constrain the tolerable variation in the representations, the better will be the comparisons. Similarly, the lower the dimension of the hu-space available in which to represent the patterns, the less freedom and therefore tolerable variation there is. As a measure of the similarity of two representation sets $\mathcal{R}^a$ and $\mathcal{R}^b$, we define the quantity C_{ab}:

$$C_{ab} := \frac{1}{1 + \sum_{i<j}^{N_{TS}} |l_{ij}^a - l_{ij}^b|}. \qquad (5.103)$$

Thus C_{ab} ranges from 0 to 1, and can be interpreted as the similarity between two solutions a and b. In order to allow for a certain leeway in the l values, we use a tolerance to determine whether to include a value $|l_{ij}^a - l_{ij}^b|$ in the summation. The presence of this leeway is needed because of the tolerance allowed in the training of the values of the output units, deriving from the nonlinear response functions, and is equivalent to assuming the bottom of a minimum in solution space is a ring a certain height above the true minimum. We need to take account of such a leeway since we do not wish to accumulate little differences in l, which become significant in (5.103) as N_{TS} becomes large causing quite similar representations to have low values of C. The tolerance was generally taken to be of the order $\Delta_{ij} = 0.1$, although allowance was made for generally larger l–values as the dimension of the hypercube (number of hidden units) increased. The suitability of the C measure depends entirely on the type of training set used, and the distribution of solutions available, and we use it here mainly to illustrate scaling of solutions for the parity problem, which are clearly defined.

First we give an example of identical solutions having different weight configurations, and show how this can be understood with reference to more obvious net symmetry.

Consider a network with $N_H = 2$, the hidden units taking on values in the range $[-1, 1]$. Figure 5.1a shows a set of representations in this space for 4 patterns. Since these representations lie within the circle of unit radius centred at the origin,

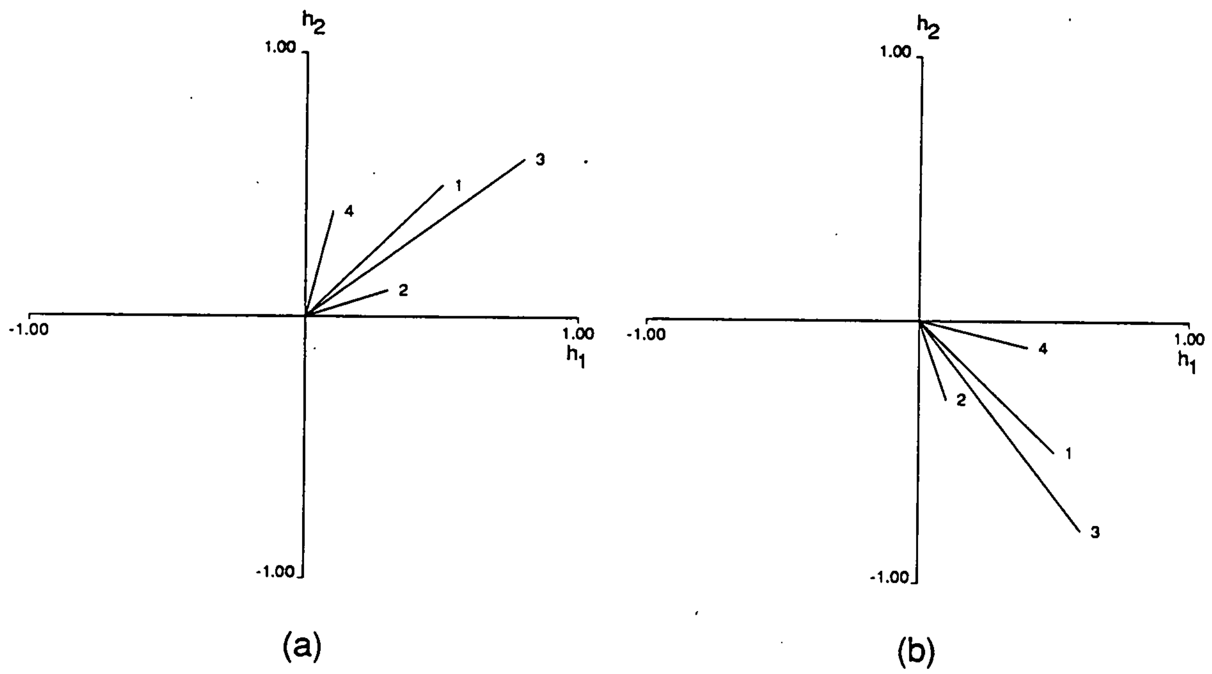


Figure 5.1: (a): a set of hidden-unit representations in 2–d space; the hidden units 1 and 2 label the axes h_1 and h_2 respectively. (b): the same set rotated through 90 degrees in a clockwise direction. (a) and (b) are equivalent solutions.

any of the group of $O(2)$ operations will leave the solution invariant in so far as the net is concerned. But even if some of the representations lie outside the circle we can still perform global rotations of multiples of 90 degrees about the origin, and the solution will be the same (figure 5.1b). This set of operations is the symmetry group of a square, or more generally an N_H–dimensional hypercube. Furthermore, a rotation of 180 degrees is just equivalent to changing the sign of all the hidden units for each pattern, and that is the same as changing the sign of all the weights going into each hidden unit. Clearly, we must also do the same to the output vector $\mathbf{o}$, and it is also clear that we can perform the 180 degree rotation by using two reflections, one in the h_1–axis and another in the h_2–axis. In general, we can reflect in any number of axes and the solution will remain the same, provided the same type of operation is performed on O. But this is just the same as saying that any network is invariant when we change the sign of all the weights going into and out of a hidden unit, a symmetry pointed out in [DSB$^+$87]. It is also clear that the hidden units can have their labels permuted without changing anything. This trivial symmetry is equivalent to relabelling the axes in figure 5.1. In general, when there may be instances of the representations lying in the hypersphere, $O(N_H)$ is a continuous group of transformations, and symmetrical solutions will not be so easy to spot.

Given a set of T representations, $\{\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_T\}$, we should be able to isolate how many unique solutions there actually are. It is necessary for the set of values $\{C_{ab}\}$ between all the solutions to satisfy an equivalence relation, if the set of unique solutions is discrete (i.e. there are no solutions which can be continuously transformed into others while remaining solutions to the mapping problem). Thus we expect to have a distribution of C–values consisting of two overlapping distributions, one centred towards zero, and the other towards one. The higher distribution is the scatter of values of C between solutions which are similar, and the lower one those values between dissimilar solutions. For some problems it may not be obvious where to draw the line between the two distributions, and for this reason an equivalence relation constraint may be used. The equivalence relation requires that, if we establish a value of C_{ab} above which two solutions a and b are the same (i.e. $a \rightsquigarrow b$), and below which they are different, then:

$$\begin{array}{ll} a \leadsto a & \text{(reflexivity)} \\ a \leadsto b \Rightarrow b \leadsto a & \text{(symmetry)} \\ a \leadsto b,\ b \leadsto c \Rightarrow a \leadsto c & \text{(transitivity)} \end{array}$$

To illustrate the use of this coefficient we will take three solutions to the parity problem for a 2–2–1 MLP (this is similar to the XOR problem), and note the values of C obtained. Figure 5.2 shows the three network configurations. It is not clear which solutions are the same. We obtain the following values of C:

C_{12}	0.271
C_{13}	0.274
C_{23}	1.000

It is clear that we have two different solutions, configurations 2 and 3 being equivalent. One solution consists of the patterns (00) and (11) being clustered together and the other two at opposite corners, while the other consists of the patterns (01) and (10) being clustered, with the other two at opposite corners. The MLP must do this kind of mapping in order to separate the patterns (01) and (10) from (00) and (11).

5.3 Scaling of available solutions for the parity problem

In order that the solution sets might satisfy an equivalence relation with regard to their relative C–values, it was required above that solutions be discrete. If the dimensionality of the hu–space is increased beyond the minimum size *required* for a solution to be possible, we expect the following scenario to become more likely: a particular hidden unit may be used for the sole representation of a single pattern, and thus we may perform symmetry operations on the other dimensions of hu–space without changing the basic representation of the above pattern. Similarly we can scale the weights going into and coming out of the hidden unit in question without disturbing the other pattern representations. Although such a

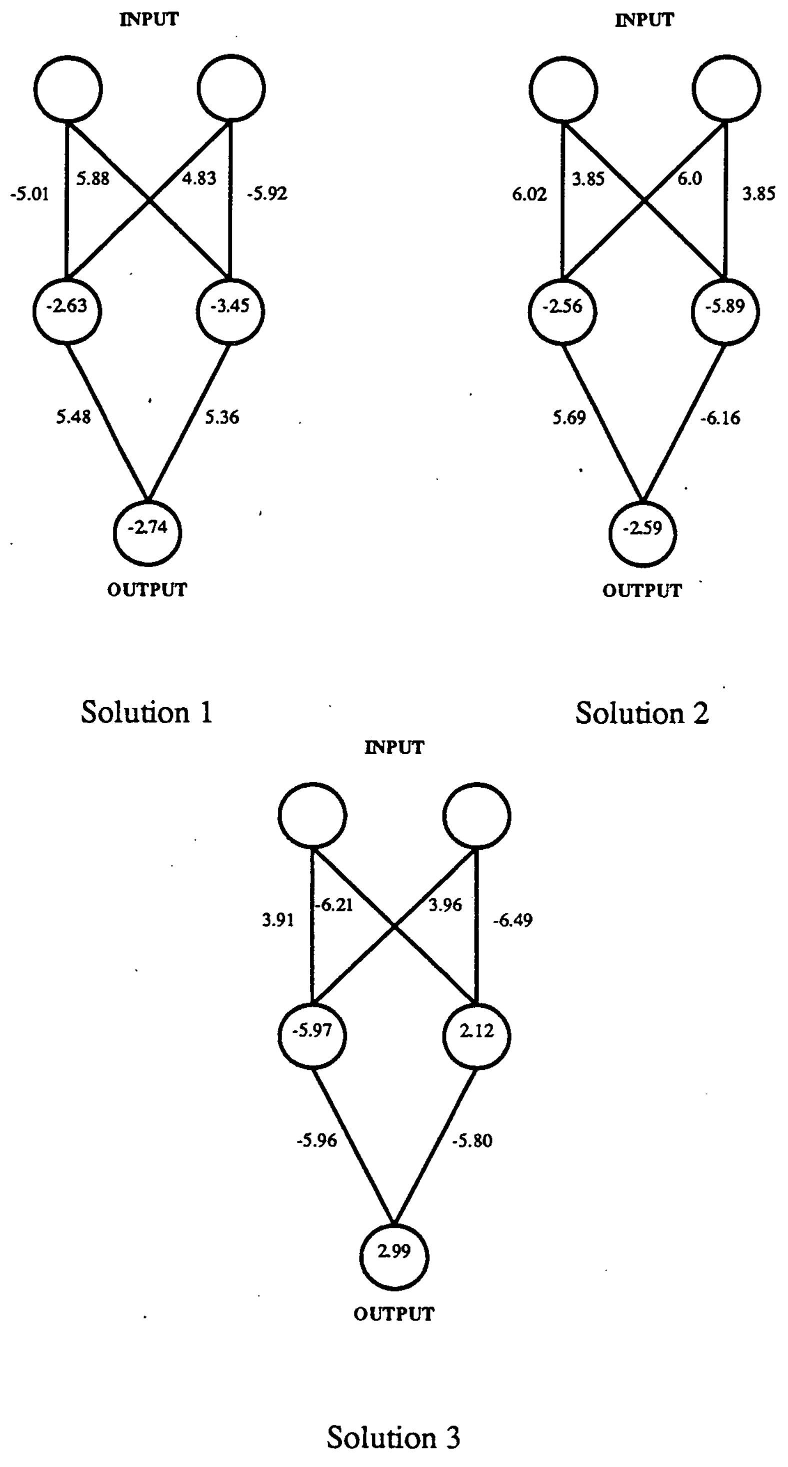


Figure 5.2: Three examples of net configurations after solving the parity problem for a 2–2–1 system. Numbers inside nodes indicate threshold values.

representation will be a rare case, the probability of such *independence* between patterns becomes more likely with the number of hidden units used above the essential number. The upshot of this is that we do not expect to satisfy the constraint of the equivalence relation between C–values for large numbers of hidden units. This progressive change is characterized by the values of C for a set of solutions becoming less easy to separate into those which indicate a definite equivalent solution, and those which indicate no equivalence. Initially, for the smallest number of hidden units required to solve a problem, it is relatively easy to separate the equivalent solutions from different ones. However, as hidden unit number is increased the two types merge, and at large numbers of hidden units, all representations appear to be different. Although it is possible to have complete independence of solutions at 4 hidden units, for the 2–2–1 system, since there are 4 patterns, this is not necessarily the easiest solution, but such solutions become more probable later on.

When such independent or semi-independent solutions are possible then we might not expect good generalization behaviour, since the independent pattern representations are created by weight values which are not constrained to be consistent with the other mappings in the training set.

It is the transition between rigidly interdependent pattern representations and the type of independence mentioned above, which should have important consequences for the generalization ability of a net.

The basic plan of the parity problem is to transform the pattern representations such that an $(N_H - 1)$–dimensional hyperplane can separate the two sets of patterns (those mapped to 1, and those mapped to 0). The solutions will vary in the different ways it is possible to represent the patterns in the **hu**–space. We will now use the C coefficient to count the number of such solutions, for various system sizes.

Figure 5.3 shows how the percentage of solutions found to the parity problem which are unique, scales with N_H. From this plot we see that the possible solutions diversify much quicker with hidden units when a higher number of input units is used. This may sound strange since as the number of inputs in the parity problem

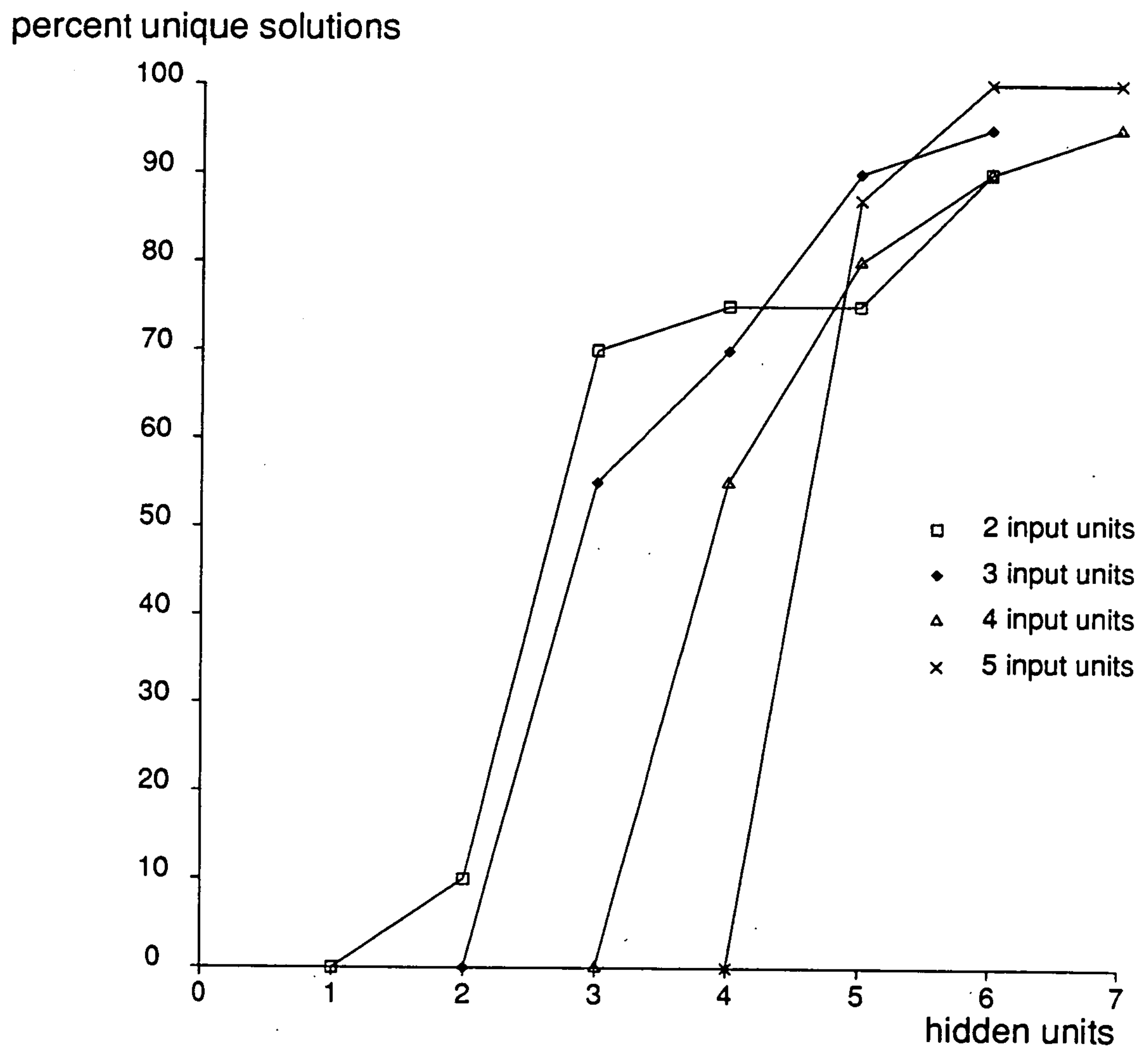


Figure 5.3: Scaling of the number of different solutions which can be distinguished as hidden layer size is increased. Shown here is the scaling for 2, 3, 4 and 5 inputs, for the parity problem.

increases, so the number of patterns requiring to be mapped increases, and one would expect that this is a harder task with fewer solutions. But one must also remember that no solution is possible until the number of hidden units reaches N_I (number of input units), and after that solution multiplicity is aided by the range of permutations of solutions possible, each themselves different in the sense of hu–representations. The parity problem is hard, and the learning algorithm looks for a solution whereby it can represent input patterns which are very close as far apart as possible in the hu–space. This is done by generally putting as many patterns as possible in a different corner of the N_H–dimensional hypercube. For $N_H < N_I$ there are many less corners than patterns, and a solution cannot be found (we have found none using the backpropagation algorithm). As we increase N_H after this bound, there are increasingly many possibilities, and much more chance of different solutions being found. This is especially so with larger values of N_I, due mainly to the solutions resulting from the greater number of permutation possibilities.

5.4 Resumé

We have established that the number of solutions found by an MLP using a *full* training set scales very quickly with the number of hidden units used in the network, and that this scaling is quicker the larger the number of mappings which are to be made. We wish to know now how this affects a guess at a missing pattern, since it might be expected that the more solutions that can be found, the more probable it is that patterns are represented semi-independently or independently, and the less the chance that the correct generalization will emerge from them. In the next section we observe the actual generalization scaling for an artificial diagnosis problem.

5.5 Generalization and a diagnosis problem

Another example of a natural problem domain, with good prospects for generalization, is fault diagnosis, given a collection of symptoms. Experiments with medical diagnosis of diseases has been successfully performed using feed-forward networks [lC85, YPB88]. It is reasonably straightforward to set up an artificial diagnosis problem domain, extending the derivation of the correlated domain of the last chapter to include multiple sets of correlations, and more than one output. We use the diagnosis type of problem in this section to illustrate the generality of the artificially generated problem domain in the last chapter, and to show how it might be used for training sets of more than one output.

This time we allow an input unit to take on the values $\{0, 1\}$, and this is interpreted as the absence or presence of a particular symptom. An output unit can take on values in the range $[0, 1]$, and is interpreted as the likelihood of a particular disease, each output unit representing a particular type of disease (or fault). We define the function mapping input states to the target t_k in the same way as in the last chapter for the reduced correlated domain:

$$f(\mathbf{I}^p, k) = t_k^p = \frac{1-2t}{A_k}\left[\sum_{l\leq m}^{N_I} I_l^p J_{lm}^k I_m^p - B_k\right] + t \tag{5.104}$$

where A_k and B_k serve to confine the set of numbers such that they all lie within the range $[t, 1-t]$. Thus

$$\begin{aligned} B_k &= \min_p\{\sum_{l\leq m} I_l^p J_{lm}^k I_m^p\} && (5.105)\\ A_k &= -B_k + \max_p\{\sum_{l\leq m} I_l^p J_{lm}^k I_m^p\}, && (5.106)\end{aligned}$$

as before, except that now there is more than one output node, each one representing the likelihood of a particular disease. Each disease k has its own set of interactions $\{J_{ij}^k\}$ which define the way in which the symptoms influence the likelihood of the particular disease. The diseases have been made to be independent (that is, there are no conditional probabilities between the diseases), and there are still self-interaction terms as in the correlated set of the last chapter, so that the presence of symptoms at all, and the co-occurrence of symptoms, both influence whether a disease or fault is present.

Since the values to be learnt are real numbers, we use the L–cost function from chapter 4, summing over all output units N_O:

$$L = \sum_{p}^{N_p} \sum_{k}^{N_O} \left\{ t_{kp} \log \frac{t_{kp}}{o_{kp}} + (1 - t_{kp}) \log \frac{1 - t_{kp}}{1 - o_{kp}} \right\} \tag{5.107}$$

The network used in all the simulations below had $N_I = 5$ inputs and $N_O = 3$ outputs. This was sufficient size for the exploration of the generalization scaling properties in this chapter. The training was considered to be finished when the value of L reached $10^{-4} \times N_p \times N_O$, giving about 2 percent accuracy per output node per pattern as before (or each pattern was learnt to within a tolerance of about 0.01).

5.5.1 Effect of training set size

In the first experiment we study the effect of training set *size* on the success in generalizing the remainder of the whole set of inputs. Each point in figure 5.4 represents the *best* generalization obtained, chosen from 5 different initial weight starting points, and 10 different, randomly selected, training sets, using the same complete training set.

The generalization G is defined using the mean output unit error e_r which a random classifier might be expected to obtain, and the mean output unit error e_n obtained by the network:

$$G := \frac{e_r - e_n}{e_r} \tag{5.108}$$

with e_n given by

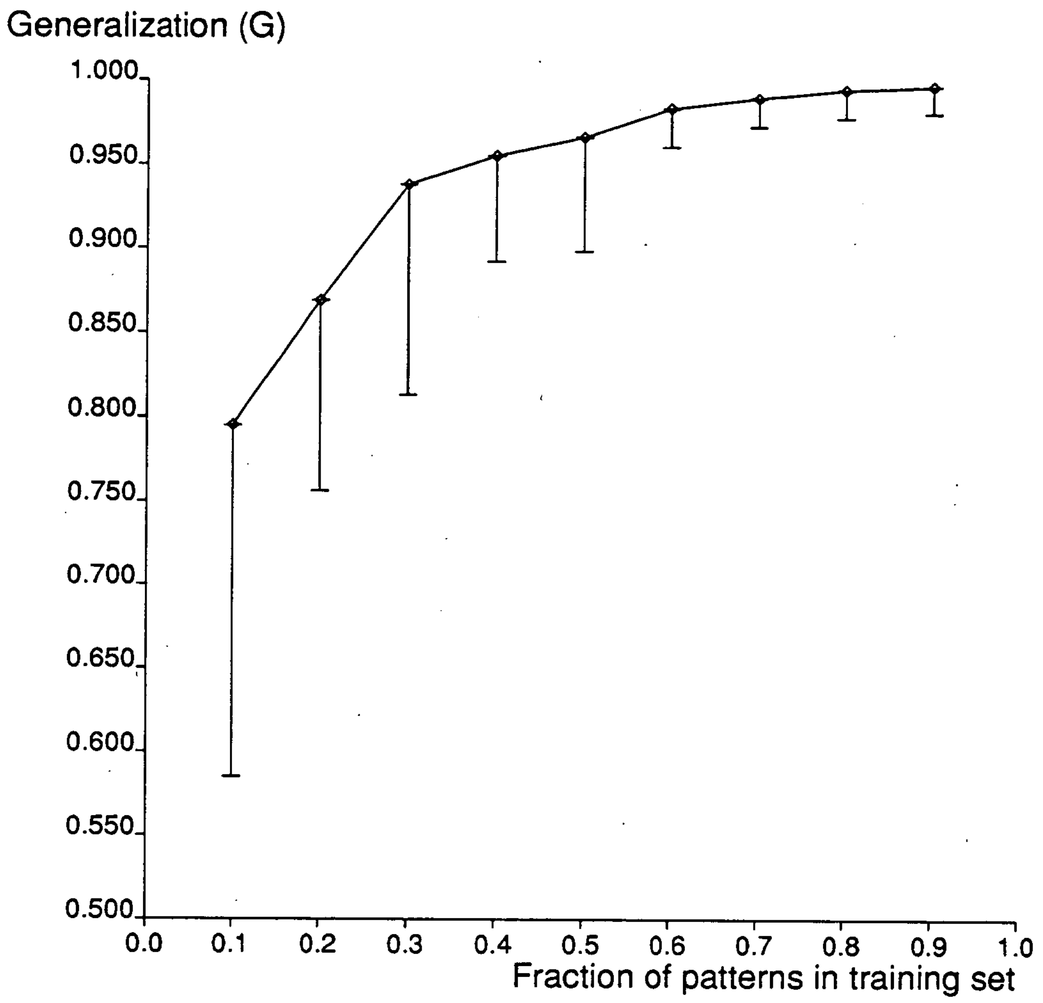


Figure 5.4: Dependence of the generalizing ability of a net on the number of patterns in the training set, and the composition of the training set. The points indicate the best generalizations obtained for a particular *size* of training set, and the error bars indicate the *range* of generalizations obtained for each particular training set size.

$$e_n = \frac{1}{M} \sum_{p \in \mathcal{F}_t}^{N_t} \sum_{i}^{3} (t_{ip} - o_{ip})^2 \qquad (5.109)$$

where N_t indicates the number of patterns in the test set $\mathcal{F}_t$, and $M = N_t \times 3$ is the total number of unit comparisons. From section 4.9.3, the value expected from a random classifier is about $e_r = 0.176$, and this is the value we shall use when calculating G. The best value G can have is 1, and $G = 0$ indicates no generalization, and less than zero some kind of anti-generalization.[2]

The actual points plotted in the figure indicate the best generalization obtained out of the range of training set *compositions* which were used. We observe that as we include more of the patterns in the training set, the performance in best generalization improves steadily. The quantity plotted reflects the mean closeness of the actual output with the target output, per output *node*. Thus as we use a larger training set, the generalization set is shifted closer to the target values as a whole. Thus the more patterns we use in the training set, the closer the typical generalizations will be to the ones we want.

Shown also in the figure are the range of generalizations obtained, for each training set size. These are the positive error bars, leading from the plotted points. From these we see how the best generalization obtainable with a given training set size depends on the *actual* set of patterns included in the training set. Some patterns are more important than others for the attainment of good generalization. This becomes less crucial as the size of the training set increases. Particularly bad generalization could be a result of an unproportionately large number of patterns in the training set being chosen from the high and low ends of the target distribution (see figure 4.4). The likelihood of this happening of course decreases as we allow more patterns in the training set.

Each pattern provides some information about the underlying trends behind the generation of the target values. It is interesting to note from the figure that the generalization remains relatively stable once more than about 60% of the patterns are included in the training set, irrespective of which patterns we include. Thus

[2]The meaning of this is not clear, but since such a phenomenon is not observed in this thesis, we shall not pursue it further.

the extra information provided by more patterns, about the trends in the training set, does not improve significantly on that information extracted from the given set of patterns.

Ahmad and Tesauro have found [AT88] that, in the majority mapping function,[3] the fraction of patterns in the test set which are correctly mapped, scales with the exponential of the number of patterns included in the training set. If we plot the logarithm of $1 - G$ (G is the *average* generalization performance) against the fraction of patterns in the training set (figure 5.5), we obtain an approximate linear relationship. The gradient of the least-squares fit to these points (the line shown in figure 5.5) is $-4.47 \pm 0.14 = -k$, with the y-intercept $a = -0.36 \pm 0.04$. Thus we have the approximate relation

$$G = 1 - e^{-kf+a} \tag{5.110}$$

where f is the fraction of patterns, from the total set of input patterns, that are in the test set.

5.5.2 Scaling with hidden layer size

We now examine how generalization performance is influenced by the number of hidden units between input and output, and relate this to the idea of relative hu–representation sets and solution number, introduced in the first part of the chapter.

Figure 5.6 shows the performance of the net on a particular training set of a fixed size (we chose $0.7 \times N_t$). The results are surprising. We would expect to observe that as we increase hidden layer size, the generalization performance would get worse, since the number of available solutions increases. This is not what happens. The generalization performance remains about the same for the range of sizes chosen. The smallest hidden layer chosen was the smallest one

[3]The single output node is to be 1 if the number of ON bits in the input is greater than the number of OFF bits.

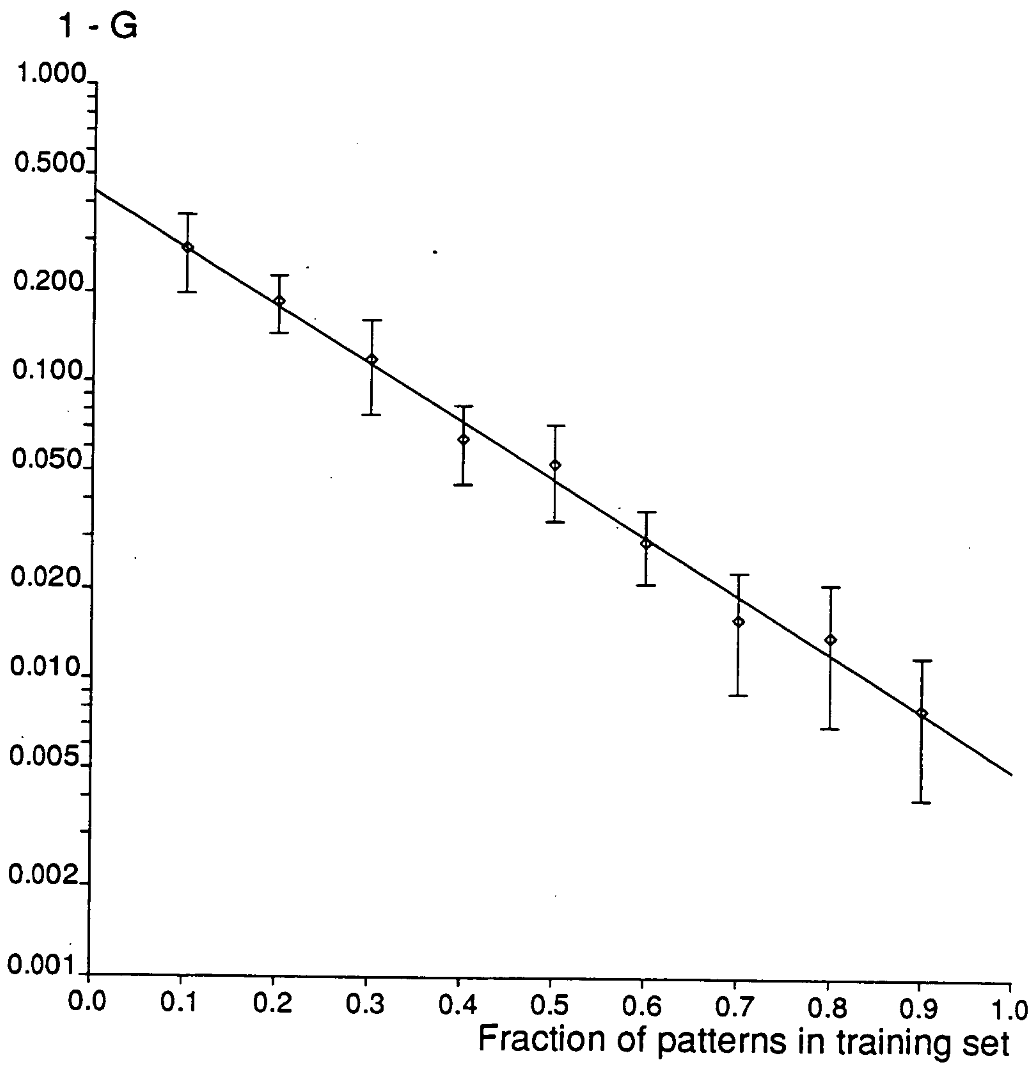


Figure 5.5: log(1 − G) plotted as a function of the fraction of the full set of patterns in the training set. The line is a least-squared fit to these points.

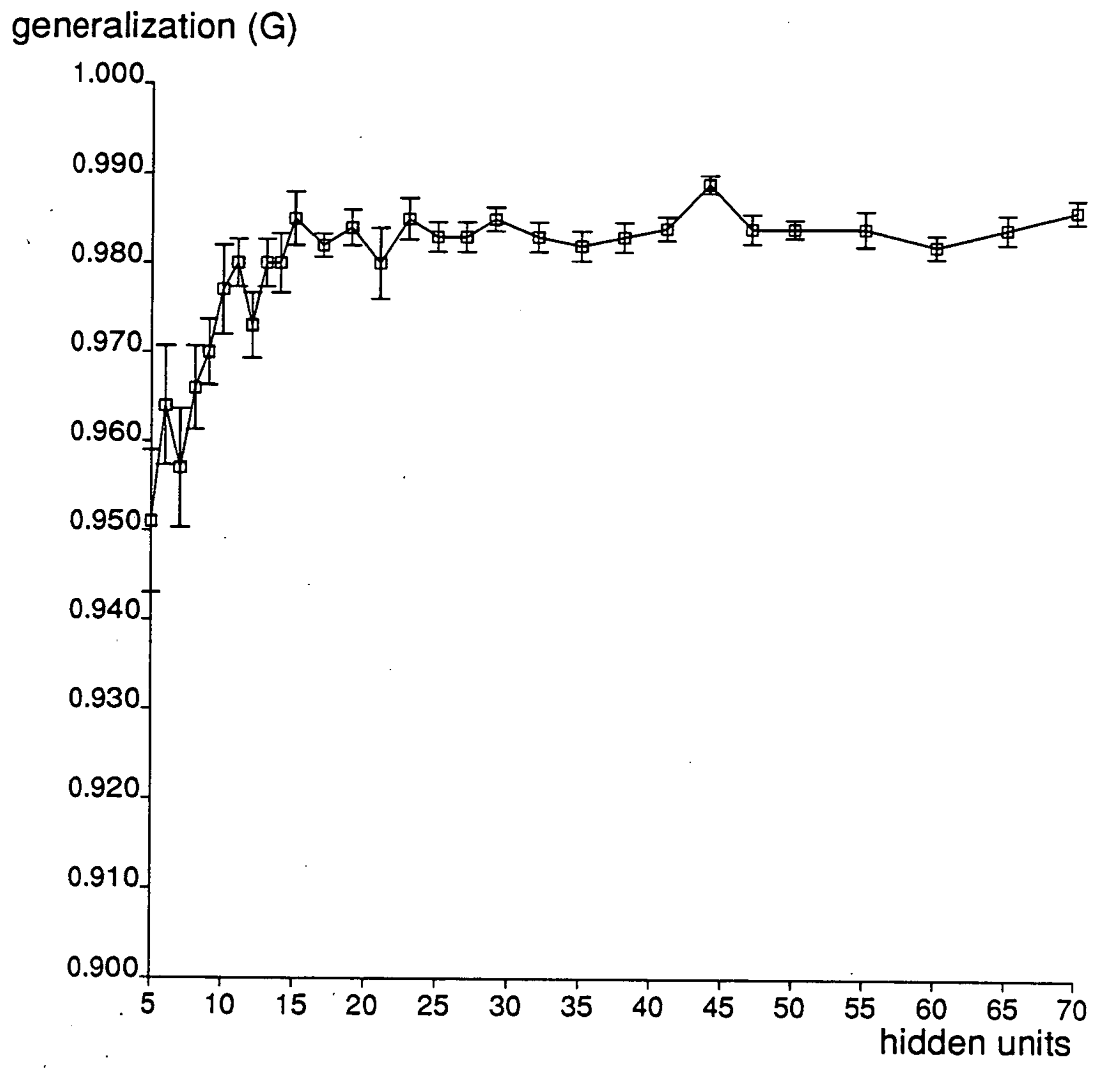


Figure 5.6: Generalization as a function of the number of hidden units. The error bars indicate the standard error of the sample of generalizations for each network size.

which could actually find a solution to the mapping. Even though the number of solutions found by the network on average increases with the hidden layer size, these solutions, although different with respect to relative pattern representations, produce *on average* similar generalizations.

Hidden	Weights	
units	1st layer	2nd layer
10	0.0046 ± 2.237	-0.017 ± 0.742
30	-0.058 ± 1.553	-0.032 ± 0.378
50	-0.0038 ± 1.306	-0.0049 ± 0.331
70	-0.0047 ± 1.056	-0.0052 ± 0.310

Table 5.1: Distribution of weights in the two layers (layer 1 = input to hidden) of the MLP as a function of hidden layer size. The entries indicate the mean value of the weights and the standard deviation of the weights about this mean.

So it must be concluded that for this problem, and problems like this (and there are probably many which can be defined in a similar way to this), the number of hidden units used is not so crucial as originally suspected. This conclusion is borne out in the applications cited in section 4.1, where the number of hidden units used is typically an arbitrary, but reasonable choice (i.e. perhaps half as many again as are required to perform the training set mappings).

It might be argued that only a certain number of hidden units are actually being *used* to produce the mapping, and that this is giving rise to the almost constant value of the generalization in figure 5.6. Table 5.1 shows that the weights in general are reduced in magnitude, as more are used to do the mapping, with the weights in the layer from input to hidden occupying a wider range than the weights from hidden to output. Thus the increase in the number of hidden units available in general causes the weights to assume lower values, as might be expected if they were all being used in learning the mapping. It must be concluded that the information about the underlying trends in the training set is stored in a *distributed* manner, throughout all the weights in the network.

Therefore it seems that despite the large increase in availability of different solutions as the hidden layer grows, demonstrated in the last section, the algorithm favours those solutions producing the emergent effect of good generalization. Additional hidden units merely cause the network to distribute the information about the training set over a larger number of weights, rather than encouraging increasingly independent mappings.

This behaviour can be understood if we consider the results on learning presented in chapter 4. The emergent properties of learning and generalization derive from the same source: the underlying regularities in the training set. Thus acquisition of the regularities allows the mappings to be better and faster learnt. Therefore we can think of this type of solution as having a greater probability of being found, or to have either a higher frequency of occurrence in the search space, or a larger basin of attraction. The solutions which are most quickly and readily learnt are those which also display good generalization.

Thus although the actual number of solutions available to the network scales very quickly with the number of hidden units, the solutions actually found for such *low-level* domains are distributed closely about the optimal solution (that found for the full training set).

5.6 Conclusions

We conclude that the solutions available to an MLP as it proceeds through search-space do, in general, increase in number as the dimension of the search-space (hidden unit number) increases. However, this is in general not problematical for the subsequent generalizing ability of the network, in the sense that the *performance* is pretty much the same for each hidden layer size.

The above conclusion may only be valid for the particular low-level problem domain used in this chapter, but we believe it to be more general. Finding the solution characterized by good generalization performance is another *emergent* property of the network. That is, learning of the patterns proceeds such that the underlying correlations effectively *speed up* the learning (the *cooperative* effect discussed in section 4.8), and these solutions are characterized by steeper and broader minima in weight-space, *however many hidden units (or dimensions) are present.* These, therefore, are the solutions that will most likely be found.

We note that the choice of patterns to include in the training set, of any particular size, has a greater effect on the final generalization ability than the size of the hidden layer. We conclude that judicial choice of training exemplars is an important factor in the effective training of feed-forward networks, and suggest that this, together with the choice of an appropriate input coding strategy, are the chief considerations likely to entice the best emergent behaviour from these networks.

Appendix A

Appendix

A.1 The Trypsin proteins used in chapter 4

Below are listed the seven proteins used in the training set:

```
Protein 1:
ILGGHLDAKGSFPWQAKMVSHHNLTTGATLINEQWLLTT
AKNLFLNHSENATAKDIAPTLTLYVGKKQLVEIEKVVLHPNYSQVDIGLI
KLKQKVSVNERVMPICLPSKDYAEVGRVGYVSGWGRNANFKFTDHLKYVM
LPVADQDQCIRHYEGSTVPEKKTPKSPVGVQPILNEHTFCAGMSKYQEDT
CYGDAGSAFAVHDLEEDTWYATGILSFDKSCAVAEYGVYVKVTSIQDWVQ
KTIAEN.
```

```
Protein 2:
VVGGEDAKPGQFPWQVVLNGKVDA
FCGGSIVNEKWIVTAAHCVETGVKITVVAGEHNIEETEHTEQKRNVIRAI
IPHHNYNAAINKYNHDIALLELDEPLVLNSYVTPICIADKEYTNIFLKFG
SGYVSGWGRVFHKGRSALVLQYLRVPLVDRATCLRSTKFTIYNNMFCAGF
HEGGRDSCQGDSGGPHVTEVEGTSFLTGIISWGEECAMKGKYGIYTKVSR
YVNWIKEKTKLT.
```

Protein 3:

RPQGSQQN
LLPFPWQVKLTNSEGKDFCGGVLIQDNFVLTTATCSLLYANISVKTRSHF
RLHVRGVHVHTRFEADTGHNDVALLDLARPVRCPDAGRPVCTADADFADS
VLLPQPGVLGGWTLRGREMVPLRLRVTHVEPAECGRALNATVTTRTSCER
GAAAGAARWVAGGAVVREHRGAWFLTGLLGAAPPEGPGPLLLIKVPRYAL
WLRQVTQQPSRASPRGDRGQGRDGEPVPGDRGGRWAPTALPPGPLV.

Protein 4:

IVGGYTCGANTVPYQVSLNSGYHFCGGSLINSQWVVSAAHCYKS
GIQVRLGEDNINVVEGNEQFISASKSIVHPSYNSNTLNNDIMLIKLKSAA
SLNSRVASISLPTSCASAGTQCLISGWGNTKSSGTSYPDVLKCLKAPILS
DSSCKSAYPGQITSNMFCAGYLEGGKDSCQGDSGGPVVCSGKLQGIVSWG
SGCAQKNKPGVYTKVCNYVSWIKQTIASN.

Protein 5:

IVGGYTCAANSIPYQVSLNSGSHFCGGSLINSQWVVSAAHCY
KSRIQVRLGEHNIDVLEGNEQFINAAKIITHPNFNGNTLDNDIMLIKLSS
PATLNSRVATVSLPRSCAAAGTECLISGWGNTKSSGSSYPSLLQCLKAPV
LSDSSCKSSYPGQITGNMICVGFLEGGKDSCQGDSGGPVVCNGQLQGIVS
WGYGCAQKNKPGVYTKVCNYVNWIQQTIAAN.

Protein 6:

IVGGYTCPEHSVPYQVSLNSGYHFCGG
SLINDQWVVSAAHCYKSRIQVRLGEHNINVLEGDEQFINAAKIIKHPNYS
SWTLNNDIMLIKLSSPVKLNARVAPVALPSACAPAGTQCLISGWGNTLSN
GVNNPDLLQCVDAPVLSQADCEAAYPGEITSSMICVGFLEGGKDSCQGDS
GGPVVCNGQLQGIVSWGYGCALPDNPGVYTKVCNFVGWIQDTIAAN.

Protein 7:

IVGGYTCQENSVPYQVSLNSGYHFCGGSLINDQWV
VSAAHCYKSRIQVRLGEHNINVLEGNEQFVNAAKIIKHPNFDRKTLNNDI
MLIKLSSPVKLNARVATVALPSSCAPAGTQCLISGWGNTLSSGVNEPDLL

QCLDAPLLPQADCEASYPGKITDNMVCVGFLEGGKDSCQGDSGGPVVCNG
ELQGIVSWGYGCALPDNPGVYTKVCNYVDWIQDTIAAN.

The following seven proteins were used in the test set:

Protein 8:
IVGGYECPKHAAPWTVSLNVGYHFCGGSLIAPGWVVSAAHCYQ
RRIQVRLGEHDISANEGDETYIDSSMVIRHPNYSGYDLDNDIMLIKLSKP
AALNRNVDLISLPTGCAYAGEMCLISGWGNTMDGAVSGDQLQCLDAPVLS
DAECKGAYPGMITNNMMCVGYMEGGKDSCQGDSGGPVVCNGMLQGIVSWG
YGCAERDHPGVYTRVCHYVSWIHETIASV.

Protein 9:
IVGGTDAVLGEFPYQLSFQETFLGFSFHFCGASIYNENYAITAGHCVYGD
DYENPSGLQIVAGELDMSVNEGSEQTITVSKIILHENFDYDLLDNDISLL
KLSGSLTFNNNVAPIALPAQGHTATGNVIVTGWGTTSEGGNTPDVLQKVT
VPLVSDAECRDDYGADEIFDSMICAGVPEGGKDSCQGDSGGPLAASDTGS
TYLAGIVSWGYGCARPGYPGVYTEVSYHVDWIKANAV.

Protein 10:
CGVPAIQPVLSGLSRIVNGEEAVPGSWPWQVSLQDKTGFHFCGGSLINEN
WVVTAAHCGVTTSDVVVAGEFDQGSSSEKIQKLKIAKVFKNSKYNSLTIN
NDITLLKLSTAASFSQTVSAVCLPSASDDFAAGTTCVTTGWGLTRYTNAN
TPDRLQQASLPLLSNTNCKKYWGTKIKDAMICAGASGVSSCMGDSGGPLV
CKKNGAWTLVGIVSWGSSTCSTSTPGVYARVTALVNWVQQTLAAN.

Protein 11:
CGVPAIQPVLSGLARIVNGEDAVPGSWPWQVSLQDSTGFHFCGGSLISED
WVVTAAHCGVTTSDVVVAGEFDQGLETEDTQVLKIGKVFKNPKFSILTVR
NDITLLKLATPAQFSETVSAVCLPSADEDFPAGMLCATTGWGKTKYNALK
TPDKLQQATLPIVSNTDCRKYWGSRVTDVMICAGASGVSSCMGDSGGPLV
CQKNGAWTLAGIVSWGSSTCSTSTPAVYARVTALMPWVQETLAAN.

Protein 12:

IVGGTNAPRGKYPYQVSLRAPKHFCGGSISKRYVLTAAHCLVGKSEHQVT
VGSVLLNKEEAVYNAKELIVNKNYNSIRLINDIGLIRVSKDISFTQLVQP
VKLPVSNTIKAGDPVVLTGWGRIYVNGPIPNNLQQITLSIVNQQTCKSKH
WGLTDSQICTFTKRGEGACHGDSGGPLVANGVQIGIVSYGHPCAIGSPNV
FTRVYSFLDWIQKNQL.

Protein 13:

IVGGTDAPRGKYPYQVSLRAPKHFCGGSISKRYVLTAAHCLVGKSKHQVT
VHAGSVLLNKEEAVYNAEELIVNKNYNSIRLINDIGLIRVSKDISYTQLV
QPVKLPVSNTIKAGDPVVLTGWGRIYVNGPIPNNLQQITLSIVNQQTCKF
KHWGLTDSQICTFTKLGEGACDGDSGGPLVANGVQIGIVSYGHPCAVGSP
NVFTRVYSFLDWIQKNQL.

Protein 14:

VVGGTRAAQGEFPFMVRLSMGCGGALYAQDIVLTAAHCVSGSGNNTSITA
TGGVVDLQSAVKVRSTKVLQAPGYNGTGKDWALIKLAQPINQPTLKIATT
TAYNQGTFTVAGWGANREGGSQQRYLLKANVPFVSDAACRSAYGNELVAN
EEICAGYPDTGGVDTCQGDSGGPMFRKDNADEWIQVGIVSWGYGCARPGY
PGVYTEVSTFASAIASAARTL.

Bibliography

[AHS85] D. A. Ackley, G. E. Hinton, and T. J. Sejnowski. A learning algorithm for boltzmann machines. *Cognitive Sciences*, 9, 1985.

[AS87] J. Austin and T. J. Stonham. The ADAM associative memory. Technical Report YCS 94, Dept. of Computer Science, York University, 1987.

[AT88] S. Ahmad and G. Tesauro. A study of generalization in neural networks. In *Abstracts of the First Annual Meeting of the INNS*, September 1988.

[BB87] M. Bedworth and J. S. Bridle. Experiments with the backpropagation algorithm: A systematic look at a small problem. Technical Report RIPRREP/1000/9/87, Royal Signals and Radar Establishment, 1987.

[BD87] T. Beynon and N. Dodd. The implementation of multi-layer perceptrons on transputer networks. Technical Report RIPRREP/1000-/13/87, Royal Signals and Radar Establishment, 1987.

[BJM83] L R Bahl, F Jelineck, and R. L. Mercer. A maximum likelihood approach to continuous speech recognition. *IEEE transactions on pattern analysis and machine intelligence*, PAMI-5(2):179–190, March 1983.

[Bod87] M. A. Boden. *Artificial Intelligence and Natural Man.* MIT Press, 1987.

[Bou86] D. G. Bounds. A statistical mechanical study of boltzmann machines. Technical report, Royal Signals and Radar Establishment, 1986.

[Dod87] N. Dodd. Texture discrimination using multi-layer perceptrons. Technical Report RIPRREP/1000/15/87, Royal Signals and Radar Establishment, 1987.

[DSB+87] J. Denker, D. Schwartz, Wittner B, S. Solla, R. Howard, L. Jackel, and J. Hopfield. Large automatic learning, rule extraction and generalization. *Complex Systems*, 1(5), 1987.

[FN71] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[For88] B. M. Forrest. Restoration of binary images using networks of analogue neurons. In *Parallel Architectures and Computer Vision*, pages 19–31. Oxford University Press, 1988.

[Heb49] D. O. Hebb. *The Organization of Behaviour.* Wiley & Sons, 1949.

[Hin87] G. E. Hinton. Connectionist learning procedures. Technical Report CMU-CS-87-115, Carnegie-Mellon University, 1987.

[Hop82] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Science*, 79:2554–2558, 1982.

[Hop87] J. J. Hopfield. Learning algorithms and probability distributions in feed forward and feed back networks. In *Proceedings of the National Academy of Science, USA,* **84**, pages 8429–8433, 1987.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[KL89] J. Kindermann and A. Linden. Detection of minimal microfeatures by internal feedback. In J. Retti and K. Leidlmair, editors, *5. Österreichische Artificial-Intelligence Tagung*, Berlin Heidelberg, 1989. Springer.

[Kul59] S. Kulback. *Information Theory and Statistics.* Wiley, New York, 1959.

[LB87] P. Lloyd and D. Bounds. A numerical study of the back propagation algorithm for multi-layer perceptrons. Technical Report RIPRREP-/1000/11/87, Royal Signals and Radar Establishment, 1987.

[lC85] Y. le Cun. Medical diagnosis using neural networks. In *Proceedings of Cognitiva*, Paris, 1985.

[LHCC86] A. Lyall, C. Hill, J. F. Collins, and A. F. W. Coulson. Implementation of inexact string matching algorithms on the ICL DAP. In M. Feilmeier, G. Joubert, and U. Schendel, editors, *Parallel Computing '85*, pages 235–240. 1986.

[Lip87] R. P. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, April 1987.

[Lor76] G. G. Lorentz. The 13th problem of Hilbert. In F. E. Browder, editor, *Mathematical developments arising from Hilbert problems.* The American Mathematical Society, 1976.

[MBB87] N. A. McCulloch, M. D. Bedworth, and J. S. Bridle. Netspeak: A multi-layer perceptron that can read aloud. Technical Report RIPRREP/1000/4/87, Royal Signals and Radar Establishment, 1987.

[McC75] J. McCarthy. Programs with common sense. In R. C. Schank and B. N. Nash-Webber, editors, *Semantic Information Processing*, June 1975. Proceedings of the Workshop of the Association of Computational Linguistics.

[Min79] M. Minsky. The society theory of thinking. In P. H. Winston and R. H. Brown, editors, *Artificial Intelligence: An MIT Perspective*, pages 421–452. MIT Press, Cambridge, Mass., 1979.

[MP69] M. Minsky and S. Papert. *Perceptrons.* MIT Press, 1969.

[NRR+89] M. G. Norman, N. J. Radcliffe, G. D. Richards, F. J. Śmieja, D. J. Wallace, J. F. Collins, S. J. Hayward, and B. M. Forrest. Neural network applications in the Edinburgh Concurrent Supercomputer Project. In *Proceedings of the NATO Advanced Study Institute on Neural Computing*, Les Arcs, 1989.

[Ola89] M. Olazaran. The perceptron debate: An insight into the history of connectionism. Technical report, Dept. of Sociology, University of Edinburgh, April 1989.

[Par85] D. B. Parker. Learning logic. Technical Report TR-47, Sloan School of Management, April 1985.

[PH86] B. A. Pearlmutter and G. E. Hinton. G-maximization: an unsupervised learning procedure for discovering regularities. In *Neural Networks for Computing.* American Institute of Physics, 1986.

[PH87] D. A. Plaut and G. E. Hinton. Learning sets of filters using backpropagation. *Computer Speech and Language*, 2, 1987.

[PNH86] D. C. Plaut, S. J. Nowlan, and G. E. Hinton. Experiments on learning by backpropagation. Technical Report CMU-CS-86-126, Carnegie-Mellon University, June 1986.

[QS88] N. Qian and T. J. Sejnowski. Predicting the secondary structure of globular proteins using neural network models. *Journal of Molecular Biology*, 202:865–884, 1988.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. *Nature*, 323(533), 1986.

[Ric88] G. D. Richards. Implementation of backpropagation on a transputer array. In J. Kerridge, editor, *Proc. 8th Technical Meeting of the Occam User Group*, pages 173–179, Amsterdam, 1988. IOS.

[RM86] D. E. Rumelhart and J. L. McClelland. On learning the past tense of English verbs. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing, volume 2,* chapter 18. MIT Press, 1986.

[Ros59] F. Rosenblatt. *Principles of Neurodynamics.* Spartan Books, New York, 1959.

[Sam63] A. L. Samuel. Some studies in machine learning using the game of checkers. In A. Feigenbaum and J. Feldman, editors, *Computers and Thought.* McGraw-Hill, New York, 1963.

[Sel59] O. G. Selfridge. Pandemonium: A paradigm for learning. In D. V. Blake and A. M. Uttley, editors, *Proceedings of the symposium on mechanization of thought processes*, pages 511–529, London, 1959. National Physical Laboratory, HMSO.

[SLF88] S. Solla, E. Levin, and M. Fleischer. Accelerated learning in layered neural networks. *Complex Systems*, 2(6), 1988.

[Smi87] L. Smith. Simulation of connectionist machine on a transputer array. Technical report, University of Stirling, 1987.

[Śmi88] F. J. Śmieja. The significance of underlying correlations in the training of a layered net. In *Abstracts of the First Annual Meeting of the INNS*, September 1988. Available as Edinburgh preprint 88/447.

[Śmi89] F. J. Śmieja. Mlp solutions, generalization and hidden-unit representations. In *proceedings of the DANIP Workshop 1989.* Oldenbourg Verlag, 1989.

[SN63] O. G. Selfridge and U. Neisser. Pattern recognition by machine. In A. Feigenbaum and J. Feldman, editors, *Computers and Thought.* McGraw-Hill, New York, 1963.

[SR87] T. J. Sejnowski and C. R. Rosenberg. NETtalk: A parallel network that learns to read aloud. *Complex Systems*, 1(1), 1987.

[ŚR88] F. J. Śmieja and G. D. Richards. Hard learning the easy way: Backpropagation with deformation. *Complex Systems*, 2(4), 1988.

[Sus75] G. J. Sussmann. *A Computer model of skill acquisition.* New York, 1975.

[TGWW74] J. M. Tenenbaum, T. D. Garvey, S. Weyl, and H. C. Wolf. An interactive facility for scene analysis research. Technical Report 87, Stanford Research Institute, 1974. AI technical Note.

[TS88] G. Tessauro and T. Sejnowski. A parallel network that learns to play backgammon. *Artificial Intelligence*, 1988. in press.

[TW75] J. M. Tenenbaum and S. Weyl. A region-analysis subsystem for interactive scene analysis. In *4th International Joint Conference in Artificial Intelligence*, pages 682–687, Tbilisi, USSR, 1975.

[Ull73] J. R. Ullmann. *Pattern Recognition techniques.* Butterworth & Co. (publishers) Ltd., London, 1973.

[Wal87a] D. J. Wallace. Neural network models: a physicist's primer. In R. D. Kenway and G. S. Pawley, editors, *Computational Physics*, pages 167–210. SUSSP Publications, 1987. Proceedings of the 32nd Scottish Universities Summer School in Physics.

[Wal87b] D. J. Wallace. Using neural networks to analyse protein sequences, 1987. Confidential Report to RSRE Malvern.

[WBLH69] D. J. Willshaw, O. P. Buneman, and H. C. Longuet-Higgins. Non-holographic associative memory. *Nature*, 222(5197), June 1969.

[Wer74] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioural Sciences.* PhD thesis, Harvard University, 1974. unpublished.

[WH60] B. Widrow and M. E. Hoff. Adaptive switching circuits. *1960 IRE WESCON Conv. Record, Part 4*, pages 96–104, August 1960.

[Win75] P. H. Winston. Learning structural descriptions from examples. In P. H. Winston, editor, *The Psychology of Computer Vision.* McGraw-Hill, New York, 1975.

[WL88] A. Wieland and R. Leighton. Shaping schedules as a method for accelerated learning. In *Abstracts of the First Annual Meeting of the INNS*, September 1988.

[Wol88] K. Wolf. Werkzeuge zur simulation neuronale netze auf parallelrechnern. Master's thesis, Universität Bonn, 1988. Diplomarbeit.

[Wri88] W. A. Wright. Toward a generic learning algorithm. preprint AIP/AW/88/4, British Aerospace, Sowerby Research Center, Bristol, UK, 1988.

[YPB88] Y. Yoon, L. L. Peterson, and P. R. Bergstresser. Desknet: The dermatology expert system with knowledge-based network. In *Abstracts of the First Annual Meeting of the INNS*, September 1988.