Small Nets and Short Paths: Optimising Neural Computation

Marcus Roland Frean

Ph.D.
University of Edinburgh
1990

Declaration

I declare that this thesis has been composed by myself and that the research reported therein has been conducted by myself unless otherwise indicated.

Edinburgh, 23 October 1990.

Marcus Frean

For my parents,
Margaret and Roly

Acknowledgements

When I began this PhD I was devoid of all confidence, background and good sense, and accordingly gave myself even odds on completing it; that it ever arrived at the binders is very much a reflection on the people I have had around me.

I have an enormous amount to thank David Willshaw for. The environment fostered by his day to day presence and supervision has without doubt been the major reason I've got anything done over the last three years. It still amazes me that I had such good luck in stumbling upon someone whose style and approach suited me so perfectly. His guidance and understanding made this possible.

For supervision, practical support and encouragement I thank David Wallace.

I owe a great deal to Peter Dayan, for many conversations that shaped my thinking, and Jay Buckingham, not least for the luxurious computing environment I have used and abused here. To both of you, sincere thanks for your support. David Willshaw and Peter Dayan have read and re-read innumerable drafts; I'm at once staggered they could justify this to themselves, and extraordinarily grateful. Kate Jeffery provided important criticisms of a late draft, and her friendship, enthusiasm and criticism have been invaluable.

A special thankyou to Megan McEwan for her constant friendship.

Finally, if Justine Young hadn't been near and dear and given support, strength, friendship and love for so much of my time here I would never have got this far, and to her my gratitude is immense. I can't thank her adequately.

This work was funded by a United Kingdom Commonwealth Scholarship through the British Council and the Association for Commonwealth Universities, and carried out largely at the Center for Cognitive Science, to each of whom I am grateful.

Abstract

This thesis explores two aspects of *optimisation* in neural network research.

1. The question of how to find the optimal feed-forward neural network architecture for learning a given binary classification is addressed. The so-called constructive approach is reviewed whereby intermediate, hidden, units are built as required for the particular problem. Current constructive algorithms are compared, and three new methods are introduced. One of these, the *Upstart* algorithm, is shown to outperform all other constructive algorithms of this type.
This work led on to the ancillary problem of finding a satisfactory procedure for changing the weight values of an individual unit in a network. The new *thermal perceptron* rule is described and is shown to compare favorably with its competitors. Finally the spectrum of possible learning rules is surveyed.
2. Neurobiologically inspired algorithms for mapping between spaces of different dimensions are applied to a classic optimisation problem, the Travelling Salesman Problem. Two new methods are described that can tackle the general symmetric form of the TSP, thus overcoming the restriction on other neural network algorithms to the geometric case.

Contents

Prologue

Introduction

This thesis describes two types of connectionist or neural network algorithm whose common thread is *optimisation*. The first part is concerned with the design of optimal neural networks for learning binary classifications; the second with algorithms for solving a classic combinatorial optimisation problem.

Learning by Construction

In the first and major part of the work, attention is focussed on the design of optimal feed-forward networks to learn binary classifications. One of the drawbacks of backpropagation, the best known algorithm for training neural nets, is that it does not supply any criteria to specify the architecture of the network to which it is applied. This brings both technical and fundamental problems. In particular, if the network is given too many intermediate units between input and output (hidden units), it will learn only slowly; if it is given too few, then it may not learn at all. Such problems are alleviated by the *constructive* approach: units are added during the learning process as and when required. One way in which this can be done is to restrict the representations that the network is allowed to use to solve the problem. Two algorithms based on this approach are presented, both of which build a purely disjunctive representation. The Whittling algorithm constructs a single layer of hidden units, and the Splitting algorithm builds a binary tree architecture. For all constructive algorithms the learning rule applied to each unit must result in convergence of the weights to stable values. The perceptron learning rule does not have this property; the Pocket algorithm does, and hence the Splitting and Whittling algorithms are compared with other constructive algorithms using the Pocket algorithm to find the weights.

An altogether different method is to construct new units explicitly to correct the errors made by existing units. This idea gives rise to a simple recursive method for constructing networks, called the Upstart algorithm. This method has been evaluated on a number of different classification problems and is found to outperform existing constructive algorithms.

An unexpected result was the development of a new rule for training individual units, called the "thermal perceptron". This rule is simple and local in operation, yet is found to be more efficient than the Pocket algorithm for problems of reasonable size. This in turn led to the examination of the general form of perceptron-like rules. A broad class of possible rules is parameterised and simulation studies are used to explore the parameter space.

The Travelling Salesman Problem

The second topic is the adaptation of neurobiologically inspired parallel algorithms to provide near-optimal solutions to a class of intractable computational problems as exemplified by the Travelling Salesman Problem. This problem can be viewed as a mapping problem, that of associating each city with a position in a tour. The success of the elastic net approach (Durbin & Willshaw 1987) shows that for problems in which the cities lie in a Euclidean space, mappings which preserve neighbourhood relationships are appropriate. The major motivation for the work presented here was to relax the Euclidean restriction. This can be done by considering the mapping as between two layers of units in a neural network. Two algorithms are developed for this purpose. One is an extension of the elastic net algorithm; the second is based on the neural activity model (Willshaw & von der Malsburg 1976), an early proposal for the way in which neighbourhood-preserving mappings form in the brain. Both methods use very simple learning rules, are intrinsically parallel and find short tours for problems in which the inter-city distances are reasonably close to those that would arise from a Euclidean metric. In both cases, the performance degrades substantially and consistently for highly non-Euclidean problems, suggesting a limitation in the general approach.

Plan of the Thesis

The first three chapters are concerned with constructive algorithms for learning binary classifications. Chapter 1 reviews other constructive algorithms, Chapter 2 discusses the Upstart algorithm and Chapter 3 investigates learning rules for training the weights of an individual unit in the network.

The next two chapters are concerned with the Travelling Salesman Problem. Chapter 4 reviews the problem itself, and the original neural network approach to its solution. Chapter 5 is about algorithms based on the topographic mapping approach.

In the epilogue, general characteristics linking neural networks and optimisation are discussed.

Part I

Pattern Classification by Perceptrons

Chapter 1

Constructive neural network algorithms

1.1 Overview

Distinguishing one pattern from another is among the most fundamental of operations for any system which responds to an environment. *Learning* to do so is among the most important of abilities. Connectionism has developed novel ways of thinking about both these problems, and has enjoyed a certain amount of success at solving them. However there are problems with the way these connectionist networks learn: the following chapters are about ways these limitations might be overcome by novel learning strategies.

1.2 Introduction

Suppose that a given set of patterns consists of two disjoint subsets or classes of patterns, A and B, and that some way of discriminating between patterns in A and those in B is required. In general this could be done by any system capable of assuming at least two states, with sufficient discriminatory power to respond to a pattern from A by going into one state, and to a pattern from B by going into another state. There are many such systems. In the simplest of all, each pattern is stored explicitly, together with a tag denoting the set it came from. This *look-up table* forms a trivial representation of the classification: no use is made of commonalities or other relationships between patterns in the same class, and the information required to distinguish one class from another is no less than that required to store all the patterns of one class explicitly.

Instead, here the interest is in the discriminatory capabilities of very simple elements known as Linear Threshold Units or perceptrons (Rosenblatt 1958), the terms being used synonymously throughout this thesis. These units are connected by *weights* to the inputs across which patterns occur and, in response to a given pattern, go into one of two output states given by

$$\begin{aligned} \text{output} \quad &= \quad \begin{cases} 1 \quad (\text{ON}) & \text{if } \phi > 0 \\ 0 \quad (\text{OFF}) & \text{otherwise} \end{cases} \\ \text{where} \qquad \phi \quad &= \quad \sum_{i=1}^{N} W_i \xi_i \; + \; \theta \end{aligned} \tag{1.1}$$

The W's are the weights, and ξ_i is the value of the i^{th} input in the given pattern. θ is called the *bias*, and is commonly treated as just another weight, typically the zeroth, from an input ξ_0 which is 1 in every pattern. In other words a perceptron is ON if the sum

$$\phi = \sum_{i=0}^{N} W_i \xi_i$$

is positive and OFF otherwise. Note the use of a 'hard' thresholding operation; although there are good reasons for eventually generalising to real-valued smooth transfer functions as opposed to step functions, the natural place to start understanding discrimination is with the binary decisions, and this restriction applies to all that follows.

There are three main reasons for the recent excitement in networks of such elements.

Firstly, they superficially resemble neurons. Real (biological) neurons consist of a cell body, with input and output processes called dendrites and axon respectively, both of which may be highly branched tree-like structures extending well away from the cell body and making contact with many other neurons. These contacts, called synapses, enable one neuron to influence another. Individual neurons are rarely seen to control others directly; generally a neuron's activity is a result of the cooperative action of many contacts from many other neurons. The 'resting' or inactive state of a neuron is one in which its cell membrane is electrically polarised, this being characterised by a negative cellular potential. If however the potential at the cell body exceeds a critical value, a wave of depolarisation sweeps rapidly from the cell body along the axon. This wave, called an *action potential*, is an 'all-or-nothing' response, continuing undiminished throughout the entire axonal tree. The cell body may reach the necessary trigger potential as a result of activity by other cells synapsing with its dendritic tree. Action potentials arriving at individual synapses have the effect of causing momentary depolarisations or hyperpolarisations (excitation and inhibition respectively) at the cell body, and the net result of many synapses is roughly cumulative. The degree to which a given synapse is able to affect the potential at the cell body is called the synaptic *efficacy* or *strength*. There is strong evidence that it is this variable which is altered during at least some forms of learning (Bliss and Lomo 1973). The accumulation of the effects of many synapses and subsequent triggering of an all-or-nothing response is closely reflected in the operation of a perceptron. The hope is that seeking parallels with the brain will have benefits for 'artificial' (*ie.* non-biological) information processing, and conversely that studying the computational properties of such simplified yet somewhat 'brain-like' networks can enhance understanding of the brain itself.

Secondly, perceptrons represent an archetypal form of *parallelism*, because they take account of (albeit in a very simple way) many factors and then make a decision, based on the total effect of all the factors. This simple summation of factors is an intrinsically parallel operation.

A third and major reason for interest in perceptrons is their capacity to learn from examples. The existence of learning algorithms both for individual perceptrons and for networks of many perceptrons or similar elements is one of their most attractive aspects. Such learning algorithms are the subject of this and the following two chapters.

In all that follows, a 'pattern' is taken to mean a set of values defined over a finite number of nodes, *these nodes possibly being perceptrons themselves.* This is because whether simulating brain-like processing or implementing a classifier, the real interest is in *networks* of perceptrons, because individual units have very limited computational power. It is therefore important to be able to consider the inputs to a unit as the outputs of other units. Unless explicitly stated otherwise, Part I of this thesis is concerned exclusively with binary inputs and outputs.

It is often instructive to treat the patterns to be learned across N inputs as points in an N dimensional hyperspace (often called 'pattern space'), with some of these points being from class A and others from class B. The classification problem is then to construct a *decision surface* such that all the class A patterns are separated from all the class B patterns.

1.2.1 Capabilities of single perceptrons

It is now fairly well understood what perceptron-like elements can discriminate in principle. If patterns are pictured as points in a space, the decision boundary formed by a perceptron as defined in equation 1.1 is simply a hyperplane in this space[1]. Thus clearly a set of patterns consisting of two classes may or may not be correctly classifiable by a single perceptron depending on whether there exists a hyperplane which separates all of one class from all of the other. Pattern sets where this separation is possible are therefore referred to as being *linearly separable*. Single perceptrons can separate (*ie.* represent the classification of) any two classes which are linearly separable. If there are N inputs, then there are 2^N possible patterns, and 2^{2^N} possible divisions (or *dichotomies* as they are called) of the space into two classes. However of these only a small subset, less than $2^{N^2}/N!$, are linearly separable (Lewis & Coates 1967). Moreover [Cover, 1965] proved that the expected number of random patterns in general position (that is, linearly independent patterns) classifiable by a perceptron is $2N$. Clearly it is not the case that all dichotomies of interest are expected to be linearly separable.

The Perceptron Learning Rule (PLR)

On presentation of pattern ξ^μ, the Perceptron Learning Rule (Rosenblatt 1962), henceforth abbreviated to 'PLR', alters the weights in the following way[2]:

$$\Delta W_i^\mu = \alpha\,(t^\mu - o^\mu)\xi_i^\mu \tag{1.2}$$

where o^μ is the actual output given by equation 1.1, t^μ is the desired or target output, and α is a positive constant. The Perceptron Convergence Theorem (Block 1962) states that *if* a set of weights exists for which the perceptron makes no errors, the PLR will converge on such a set after a finite number of pattern presentations. Hence perceptrons can learn any dichotomy that they can represent. However if no such solution exists, the weights are never stable since they change every time an error is made.

[1] If the bias is considered as another input, the picture is of patterns as points on a hyperplane displaced from the origin in an $N+1$ dimensional space, with the perceptron decision boundary being another hyperplane, passing through the origin.

[2] Notation: although here and elsewhere learning rules are stated with an index indicating the particular (i^{th}) input, the rule is assumed to be implemented for all inputs (i=0..N) convergent on the unit, where the zeroth input is understood to provide the necessary bias.

The Pocket algorithm

A simple extension of perceptron learning for non-separable problems called the *Pocket algorithm* (Gallant 1986a) suffices to make the PLR well behaved, in the sense that weights which minimise the number of errors can be found. The Pocket algorithm consists of applying the PLR with a randomly ordered presentation of patterns, but also keeping a copy of a second set of weights in addition to those currently used in the perceptron. These weights are just the set of perceptron weights which have lasted for the largest number of consecutive presentations without being changed. Since the patterns are presented in random order, good sets of weights will have a lower probability of being altered at each presentation, and therefore tend to remain unchanged for a longer time than sets of weights which engender more errors. This 'pocketed' set of weights will give the minimum possible number of errors with a probability approaching unity as the training time increases. That is, if a solution giving say p or fewer errors exists then the Pocket algorithm can be used to find it. Another attractive aspect is that continuously generated data may be used, instead of a training set of fixed size, and the unit learns continuously on-line, instead of updating its weights after each sweep through an entire training set. The amount of computation involved is little more than that of the perceptron alone. Note that if the patterns are indeed separable, then the Pocket algorithm's behaviour reduces to the usual PLR. [Gallant 1986a] mentions that the Pocket algorithm out-performs a standard technique (Wilks method, in [SPSS-X 1984]), by about 20% for a set of 15 learning problems.

The original algorithm improves the pocketed weights in an entirely stochastic fashion - there is nothing to prevent a good set of weights being overwritten by an occasional long run of successes involving bad weights. The so-called 'ratchet' version of the Pocket algorithm (Gallant 1989) is a simple way to ensure that every time the pocket weights change, they actually improve the number of errors made. We keep track of another quantity, which is the actual number of errors made using these weights if every pattern in the training set is applied to the input. Suppose that a certain set of perceptron weights has remained unchanged (that is, made no errors) over the last L presentations, and that L is greater than the previous longest run. In the simple Pocket algorithm these weights replace the existing pocket weights. In the Pocket algorithm with ratchet, at this point we run through *all* the patterns in the training set, and count the number of errors made if these weights are used. If this number is less than that associated with the existing pocket weights, we replace the pocket weights, the longest run length, and the least number of errors, with their new values.

The algorithms discussed in this chapter rely upon the Pocket algorithm, both from the theoretical point of view to show how convergence can be guaranteed and in practice to generate the weights. The main strength of the Pocket algorithm is the fact that optimal weights are found with probability approaching unity, given sufficient training time.

Unfortunately this method has a number of unattractive aspects. There is no bound known for the training time actually required to achieve a given level of performance, and this considerably weakens its theoretical appeal. Moreover, in practice the weights do not improve much beyond the first few cycles through the training set. Although the weights so obtained are better than many other methods, they still tend to make many more errors than an 'optimal' set would make. To get better weights from the same procedure the 'ratchet' is required. However whereas the original doubling of the number of parameters (that is, the weights) does not greatly increase the computational cost, the search over all patterns required in the ratchet version does. In practice, learning over the same number of epochs takes orders of magnitude longer to implement using a ratchet than without when reasonably difficult problems are attempted. Finally, the Pocket algorithm doesn't indicate much of interest about pattern classification: the simple PLR algorithm operating on non-separable patterns apparently executes little more than a random walk around a bounded[3] region of weight-space, and the Pocket algorithm merely samples sequences of points in this weight space at random.

1.2.2 Networks of perceptrons

Networks of interconnected perceptrons are much more powerful than individual ones. If an extra layer of 'hidden' perceptrons is inserted between input and output, *any* consistent[4] dichotomy can be represented, provided there are enough units in this layer. The extra power arises because, unlike the outputs, the hidden units do not have their states prescribed externally, but are 'free' to learn to respond to the input patterns in a useful way. This type of network is termed *feed-forward* because the activity of each unit depends only on activities of units in preceding layers.

The problem of actually *learning* with success, generality and simplicity in such networks has been one of the major questions confronted by neural networks research. The central problem is that although the hidden units are known to be necessary in order to perform complex mappings from input to output, it is not

[3]This is a corollary to the Perceptron Cycling Theorem (Minsky & Papert 1969).

[4]Obviously if the same pattern appears in both classes, no classifier can succeed since the set is inconsistent.

obvious how best to train them. In 1969, at the end of their highly influential book *Perceptrons*, Minsky and Papert wrote pessimistically of the extension to hidden layers:

> 'The problem of extension is not merely technical. It is also strategic. The perceptron has shown itself worthy of study despite (and even because of!) its severe limitations. It has many features to attract attention: its linearity; its intriguing learning theorem; its clear paradigmatic simplicity as a kind of parallel computation. There is no reason to suppose that any of these virtues carry over to the many-layered version. Nevertheless, we consider it an important research problem to elucidate (or reject) our intuitive judgement that the extension is sterile. Perhaps some powerful convergence theorem will be discovered, or some profound reason for the failure to produce an interesting 'learning theorem' for the multilayered machine will be found.'

To a large degree, the recent resurgence of interest in connectionist systems has been due to the development of learning procedures which enable hidden units to be trained (see for example [Rumelhart *et al.* 1986, chapters 7 and 8], [Hinton 1987], [Lippmann 1987], [Cowan & Sharp 1987]).

1.3 Networks with fixed architectures.

The majority of research on neural network learning has concentrated on networks whose architecture is fixed before learning occurs. However learning with fixed architectures poses two major problems:

the representational problem. There is no *a priori* way of assessing how large a network has to be in order to solve a given task (in fact the problem of deciding whether a given structure can implement a given mapping has recently been shown to be *NP*-complete (Judd 1987; Blum & Rivest 1988)). Use of a fixed architecture therefore means adopting some network structure without knowing if it is adequate for the task.

the learning problem. Even where a network structure is known to be adequate for the given task, fixed networks cannot be guaranteed to achieve this level when using the usual learning methods.

Methods such as backpropagation (Rumelhart, Hinton & Williams 1985; Werbos 1974) work by attempting to minimise a single global quantity, namely some measure of total error made by the network. This measure can be interpreted as a surface (the *error surface*) in the space of free parameters (the weights), with the optimal configuration of weights giving the deepest minimum in this surface. Such a surface may have many other 'local' minima which are not as deep as the 'global' minimum sought. Methods such as backpropagation are described as *gradient descent* methods because they alter the weights so as to move down the surface by means of some estimate of the local gradient. The above problems may be seen as symptomatic of the pitfalls of this approach, which emphasises the view of learning as a search for the global minimum in a space of high but fixed dimensionality. The first problem is that even the global minimum may not be deep enough for the purposes of the network, and the second is that even if it is deep enough, any method which works by moving in small steps across this surface is prone to becoming 'stuck' in sub-optimal minima.[5]

1.4 Networks of minimal size

While both the above problems may be avoidable if the size of the network is very much in excess of that needed in principle, there are good reasons for wanting to produce trained networks that are close to the minimal size required for a given task. This involves the notion that a network which solves a learning problem with as few as possible hidden units is capturing something important about the nature of the task.

Firstly, a network which has in some sense extracted the essence of the task it has learned should be better at generalising to novel input patterns drawn from the same distribution. This is closely related to the issue of under- *vs* over-fitting of data sets, in that if a data set can be fitted perfectly well by a curve defined using a small number of parameters then this is likely to be a better estimator of the underlying distribution than a complicated curve. Therefore networks of near-minimal size might be expected to exhibit better generalisation to novel inputs than much larger networks, where both succeed to the same degree in classifying the training set. This conjecture is supported by theoretical work such as [Denker, Schwartz *et al.* 1987] and [Baum & Haussler 1989], as well as simulation results as in [Le Cun 1989] and [Morgan & Bourlard 1990].

[5]This prompted Minsky and Papert to conclude in the epilogue of the expanded edition of 'Perceptrons' in 1988 that the quotation I gave above was still applicable.

Secondly, a succinct representation of a problem is a desirable goal in itself. Part of what makes learning in networks so interesting is that they develop their own representations instead of merely implementing solutions imposed from outside. However if the nets are large the representations are often inscrutable and at worst trivial. An overly large network can simply form a look-up table representation, whereas a small network *must* take account of the relationships between patterns in order to succeed.

One way to approach the problem of building optimal sized networks is to train very large networks and then remove weights or units wherever possible. Examples of this approach can be found in [Chauvin 1989], [Hanson & Pratt 1989], [Mozer & Smolensky 1989] and [Le Cun, Denker & Solla 1990]. The opposite approach is adopted here, namely the building up of a representation from a single unit until the task can be solved. This gives rise to so-called *constructive* learning algorithms, which include the capacity to add units to the network as well as to train the weights.

1.5 Building networks incrementally.

The major appeal of constructive methods is the possibility of generating near-minimal sized networks. In addition to this there may be other benefits.

Firstly, learning may be a lot easier if not all the units are learning at the same time. This is related to the 'credit assignment problem' encountered in training networks with hidden layers. The problem of assigning credit arises because when an output is in error it is not clear which of the hidden units should alter its weights (and by how much), since potentially every one of them contributed to the error. Algorithms for learning in fixed architectures can always be recast in terms of heuristics regarding which units were 'to blame'. By learning in only a small number of the units at any one time this problem is avoided. The only substantial credit assignment is the local one involved in building weights for a given individual unit, which is not nearly as hard as deciding which weights in a whole network to change, in which direction, and by how much.

Secondly, it is possible that learning in this way has significantly different properties from that in fixed networks[6]. This may be especially true where the way in

[6] For example a network constructed from a single unit is bound to have some sort of hierarchical structure, and this will be reflected in the internal representation formed of the training data.

which weights are altered is intrinsically tied to the way in which units are added (rather than say, gradient descent of the usual kind but which merely adds another unit). Just as the view of learning as gradient descent of a surface has proved a valuable tool for thinking about learning in general, approaches which actually combine weight modifications with incorporation of new units into networks may bring out similar insights.

It is well to keep in mind a notable disadvantage of building networks incrementally: the learning process cannot be as parallel or distributed as it is in a fixed architecture. At the very worst the learning can only proceed on one unit at a time, instead of on all units at once.

1.6 Constructive algorithms for perceptrons

There are two powerful design constraints on constructive algorithms for neural networks.

1. They should provide a guarantee for convergence on the training set. This is important because the most immediately interesting quality of incremental methods is that they get better; the *proof* that this occurs is important whatever the procedure in question. The hope is that in this way one may avoid the kinds of 'fuzzy' convergence properties of algorithms based on gradient descent in fixed architectures.
2. The algorithm should involve only local computations amongst connected units, both for altering weights and classifying input patterns. No other external 'control' should be required, apart from the provision of the target signal.

Gradient descent methods can readily be adapted to be constructive, but this generally means there is no real guarantee of convergence or worst-case network size, and in many cases the second constraint is also stretched. Several workers have studied the effect of simply adding units to networks which learn by back-propagation, the principle issues being when to add a unit, and how to initialise its weights (see for example [Honavar & Uhr 1988], [Kruschke 1988], [Ash 1989], [Thacker & Mayhew 1990], [Smith 1990]). There are also some gradient descent type algorithms which actually use the incremental nature of the algorithm in defining weight changes, notably [Sun, Lee & Chen 1988] where a tree of units is built based on an information theoretic measure, and the 'Cascade-correlation' algorithm (Fahlman & Lebiere 1990) where units are trained to maximise their correlation with the output unit's errors. This results in new units which are responsive to the errors made by the existing output. The output unit can then use the response of the new unit (by learning the appropriate connection weight) to decrease its own errors.

There are other procedures, generally referred to as 'geometrical' methods (Rujan & Marchand 1989; Ramacher & Wesseling 1989; Hao *et al.* 1990), which are able to guarantee convergence by constructing units but are not implemented using local information transfer and hence cannot be said to learn in the usual neural network sense.

There are two aspects to the convergence properties of all the constructive algorithms for perceptrons discussed here. When a new unit is added, it needs to be shown firstly that there *exist* weights associated with this unit which will result in an improvement in the network's performance, and secondly that such weights can be *learned* by the unit. Although not always explicitly stated in the literature, the existence of useful weights depends on the following property of sets of patterns, which I will call *convexity*[7]. A set of patterns will be described as convex if and only if each pattern in the set can be separated by an appropriately positioned hyperplane from all the rest. Clearly this is true of binary patterns, since they are corners of a hypercube, and any such corner can be 'sliced off' from the rest of the hypercube. For instance the weights

$$\begin{aligned} W_i &= 2\xi_i^\mu - 1 \qquad\qquad \text{for } i = 1..N \\ \text{with a bias weight} \quad W_0 &= -\sum_{j=1}^{N} \xi_j^\mu \; + \; 1 \end{aligned}$$

define a perceptron which is ON for pattern ξ^μ and OFF for all other binary patterns. Another frequently used class of patterns which are convex arises when the individual inputs ξ_i are real-valued and each pattern is normalised such that

$$\sum_{i=1}^{N} \xi_i^2 = 1$$

Such patterns lie on the surface of a hypersphere in N-dimensional space. Any individual point on such a surface can be separated from the rest by a hyperplane which just touches the hypersphere at that point. Once it can be shown for a given algorithm that a useful set of weights does exist for each unit added, convergence to zero errors is guaranteed *provided the weights algorithm finds such a set in the*

[7] A convex region is usually defined as one in which a line between any two points in the region is itself entirely within the region. This is similar but not identical to the definition used here for a set of points to be convex.

worst case. However this must be tempered by the fact that there is no known upper bound on the time required for the Pocket algorithm to find such a set of weights: the probabilistic guarantee of the Pocket algorithm is also true of an algorithm which simply assigns random weights to a perceptron. In practice either the Pocket algorithm must be run for a very long time to ensure weights which distinguish at least one pattern from the rest, or an explicit check can be made and training continued if the weights aren't good enough. The second option amounts to a 'hack', so the first was adopted in the simulations reported here.

In the algorithms studied in the following chapters the networks are all feed-forward, since the inclusion of feedback greatly complicates network analysis. The learning tasks are all simple two-way discriminations, and the units themselves are all simple perceptrons as defined in equation 1.1. In summary, the following discussion concerns *feed-forward networks of simple perceptrons, with a single output unit, learning convex patterns.*

1.6.1 The Tower algorithm

[Gallant 1986b] introduces this method and describes its convergence property. [Nadal 1989] arrived at the same algorithm independently, as a special case of the Tiling algorithm (Mézard & Nadal 1989) which is discussed shortly.

The idea behind the Tower algorithm is very simple. Given a set of input patterns and target outputs, a unit (call it A) is generated with weights from each of the inputs. This unit is trained on all the patterns using the Pocket algorithm. If this unit succeeds the task is solved, but otherwise the Pocket algorithm produces a 'good' set of weights, given sufficient training. Then a second unit (say B) is generated which is also connected by weights to the inputs, but also has a weighted connection from A, and all weights are trained in the same way. If this unit doesn't succeed a third unit (C) is added, and so on as shown in figure 1.1. In the simple Tower algorithm each new unit receives weights from the inputs and only its immediate predecessor, while in a second version it has contact with *all* previous units. It is easy to show that each new unit can make at least one fewer errors on the training set than did any of its predecessors, in the case where the patterns are convex. For instance, if the first unit, A, is wrongly OFF for pattern ξ^μ, the second unit, B, can have weights from the inputs which guarantee it is active for this pattern alone as shown above. If B also has a large weight from A, it behaves just as A does except it gets the μ^{th} pattern correct. A similar argument applies to patterns for which A is wrongly ON.

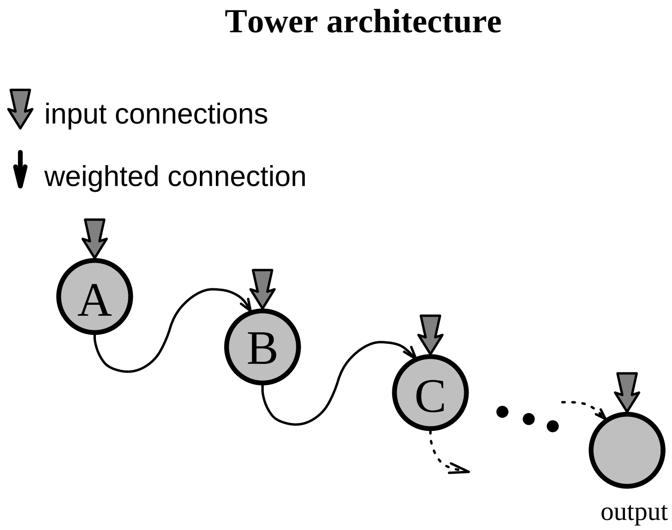

Figure 1.1: The architecture constructed by the Tower algorithm. In this case only the immediately previous unit is available to its successor. The output unit is simply the first such unit which makes no errors.

1.6.2 The Tiling algorithm

Mézard and Nadal published this algorithm in 1989. The core idea of the Tiling algorithm is what they call *faithfulness*, which they observed to be a necessary property of each layer in a feed-forward network. The representation at a given layer is said to be faithful provided that both targets are never assigned to the same pattern at that layer. Thus *un*faithfulness is simply the presence of contradictory data. Obviously if the raw input is contradictory it is impossible for any system to get every pattern correct, since there is at least one pattern which tries to map to two different targets at once. Mézard and Nadal call the set of patterns at layer l which cause the same pattern at layer $l+1$ a *class*, so for the representation at that layer to be faithful each class must include patterns of only one target. Assuming consistency in the input patterns then, a neural network must maintain faithfulness at each successive layer. Mézard and Nadal's idea was to invent a learning algorithm which constructed units in such a way that

- one unit in each layer makes fewer errors than a corresponding unit in the previous layer.
- the set of patterns produced in each completed layer is faithful.

This combination ensures eventual convergence to zero errors in a number of layers which is at most equal to the number of errors made by a single perceptron attempting to learn the task.

In the Tiling algorithm two different types of unit are required: *master units* and *ancillary units*. Each new layer has a single master unit, whose role is to make fewer errors than the master unit in the preceding layer, and as many ancillary units as are required to ensure faithfulness of the representations formed in that layer, as shown in figure 1.2. The layer $l = 0$ is taken to be the input layer. Assuming the representation at layer l is faithful then, layer $l + 1$ is constructed as follows:

Step 1. Generate a master unit M_{l+1} in the new layer. Train the weights to M_{l+1} from all the units in l using the Pocket algorithm. It is easy to show that M_{l+1} can make fewer errors than M_l ($l \geq 1$), provided the patterns in layer l are faithful. If $l = 0$ the number of errors is just that made by a single perceptron trying to learn all the input patterns.

Step 2. Examine the representations of input patterns as they appear in layer $l+1$ (initially there are just two, master ON and master OFF): if any class at this layer contains patterns of both targets[8] the representation is unfaithful. In this case the largest offending class is assigned to a new ancillary unit. This ancillary unit tries to learn only on this restricted set of patterns, using the Pocket algorithm.

Step 2 is repeated until the representation at layer $l + 1$ is faithful. In this way the two conditions above are satisfied for each successive layer, so eventually there will be a layer in which the master unit makes no errors. This is then taken to be the output unit and the problem is solved.

In fact the conditions can be satisfied without any ancillary units at all, provided each master unit is connected directly to the (assumed faithful) inputs (Nadal 1989), and to the preceding master unit as before. This then is an alternative view of the Tower algorithm.

[8]Note that this step is not truly local at all since a class is defined using the activities of *all* units in a layer. However it can easily be made so, as will be shown in section 1.6.5.

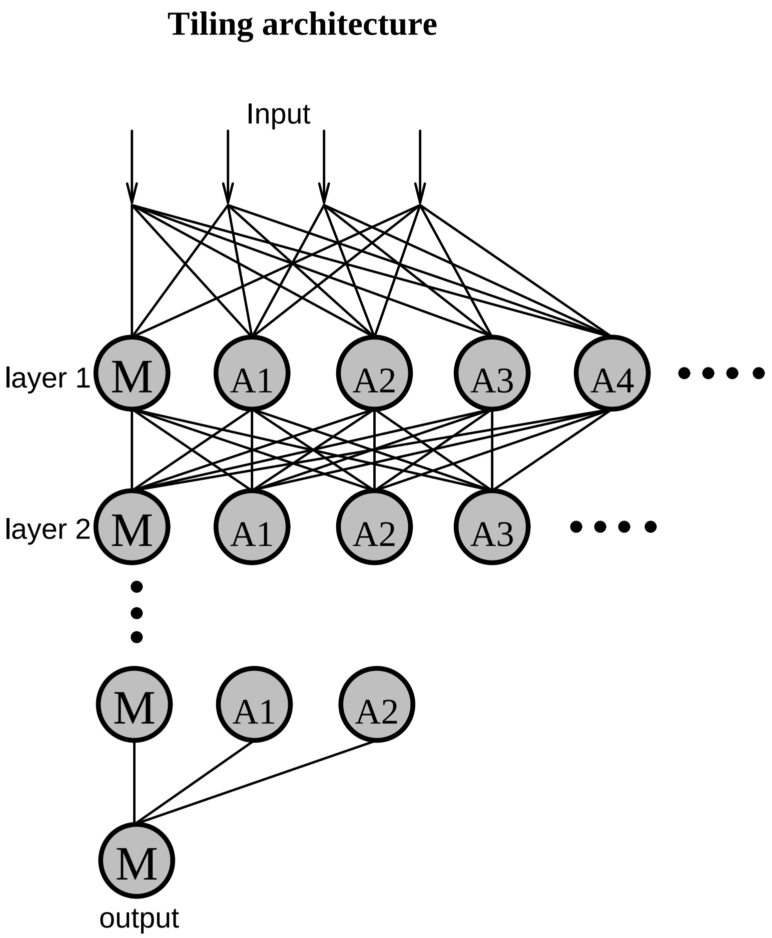

Figure 1.2: The architecture constructed by the Tiling algorithm. Master units are labelled *M* and ancillary units *A1, A2* and so on. The output unit is the first master unit which makes no errors.

1.6.3 A different approach to this problem.

The problem of how to train hidden units is usually thought of as how to get the output units to give the correct answer. However, for an output unit to succeed, the internal or 'hidden' representations of the input patterns must be of the right form. Hence the search for the weights which solve the problem can be recast in terms of a search for appropriate hidden representations. This view is the basis for various variants and extensions of backpropagation learning (see for example [Le Cun 1985], [Plaut *et al.* 1986], [Rohwer 1990], [Krogh *et al.* 1990]) and also perceptron learning (Grossman *et al.* 1989, Nabutovsky *et al.* 1990) for networks with fixed architectures. Indeed, one view of the Tiling algorithm is that it is a method for constructing progressively more useful and compact hidden representations.

We can take this idea one step further: the output unit can only succeed if the patterns it sees are actually linearly separable, so the hidden layer has a 'role', namely to produce a separable representation of the input patterns. There are many dichotomies which are linearly separable (even though in proportion to the total number of dichotomies this number is tiny). There is also a large degree of redundancy among hidden representations. For example [Denker, Schwartz *et al.* 1987] point to an 'ordering' symmetry and a 'polarity' symmetry. The ordering symmetry arises because the ordering of hidden units within a layer is arbitrary, so any given representation across H hidden units is one of a family of $H!$ essentially equivalent representations. The polarity symmetry refers to the fact that if all the weights associated with a given hidden unit are multiplied by minus one and the output's bias is adjusted appropriately the effect on the output is unchanged, giving a further 2^H-fold symmetry. Moreover, for perceptron units the magnitude of the weights vector of any given unit is irrelevant since it is only the sign of ϕ which determines the output response.

Since a great many different internal representations can be used to solve the same problem, one way to get at the problem of learning in networks is to remove part of this degeneracy. This involves biasing the network towards one or other particular representation instead of allowing it freedom to choose between lots of essentially similar solutions. The simplest way to do this is to preassign some linearly separable representation and get the hidden layer to produce it. One of the simplest hidden representations is where *all* the patterns in one class map onto a single pattern in the hidden layer, and *none* of the patterns in the other class map onto this same pattern. This is a separable representation since a perceptron (in this case the output unit) can always respond to this binary pattern and no other. That is, we aim to group all the patterns of one target class together in the space of patterns across hidden units ('H-space'), and merely exclude the other class from this pattern. The particular pattern chosen here is the origin;

the object is to map all target 0 input patterns onto the origin in *H*-space, and all the target 1 patterns elsewhere. What is needed is a set of hidden units which are all OFF for any target 0 pattern in the learning set, but at least one of which is ON for every target 1 pattern. Therefore two properties are essential:

Property 1. that each relevant hidden unit should get at least one target 1 pattern correct, but be OFF for *every* pattern which is target 0. In the space of patterns this amounts to positioning a hyperplane so that it 'slices off' a portion of the hypercube which has only target 1 patterns on it.

Property 2. that each target ON pattern activates *at least one* such hidden unit.

Note that in this representation the polarity symmetry is removed. The output effectively does a logical *OR* operation over the outputs of the hidden units, so this might be termed 'disjunctive'. From now on it is just referred to as the *OR*-representation. Particular appeals are firstly that the link with boolean logic is apparent, secondly that this is in a sense the 'natural' representation if the training set is itself composed of the conjunction of several features, and thirdly that it is simple enough to suggest constructive algorithms.

Two new constructive algorithms will now be discussed that incrementally build such *OR* representations. The first generates this in a single hidden layer in a manner reminiscent of 'whittling' off one class from another, and the second uses hidden units in a tree structure to recursively split up the training set until a set of hidden units is arrived at which form the *OR* representation.

1.6.4 Producing *OR* by Whittling

Consider the following trivial incremental network method. We have an input layer, an initially empty hidden layer and a single output unit with no connections. Now a pattern is presented. If the target is 0, there's no change since the output was OFF anyway. If the target is 1, a hidden unit is generated and given weights such that it is ON for *this pattern alone*, and its weight to the output is set positive. The output unit's bias is zero, so this pattern now turns the output ON and corrects the error. This process is repeated for all patterns in the training set. Obviously every new unit corrects one error without causing any others to occur, so the hidden layer is effectively a look-up table of the target ON patterns, and its size is just the number of such patterns in the training set.

In the above, no use is made of the relationships amongst the patterns themselves. To do better, each hidden unit should attempt to get *at the very least* one target 1 pattern correct, and preferably many more than one where this is possible. This can be seen as an optimisation problem: minimise the number of incorrect target 1's (a 'soft' constraint) subject to ensuring that the number of incorrect target 0's is zero (a 'hard' constraint). One way to produce these units by using the Pocket algorithm is as follows.

Start. Begin with the training set consisting of all the patterns.

Step 1. Use the pocket algorithm to produce a good set of weights for this training set. If the weights make no errors on this set then STOP.

Step 2. From the training set of this unit, remove the target ON patterns which give lowest (*ie.* most negative) ϕ. A good way to do this is to remove some fixed proportion (typically 10-50%) of the target ON patterns. Then repeat Step 1.

In this way the training set is eventually made linearly separable solely by removing target 1 patterns, so the final set of weights gets *every* target 0 pattern correct and at least one pattern correctly ON (since in the worst case the last target 1 pattern is certainly separable from all the other patterns). One drawback is the relatively poor performance of the Pocket algorithm, and hence the possibility that the set of weights which are found will actually turn the perceptron OFF for every pattern. This happens because if there are relatively few target 1 patterns, the number of errors made by never being active is relatively small, hence these weights are likely to be pocketed. Obviously a unit which is off for every pattern is of no use. Another drawback is that the removal of patterns from the training set in this way effectively requires that the unit have access to a listing of the training set complete with 'tags' denoting patterns to be ignored[9].

Having successfully produced such a unit satisfying the first property given above, the construction of units one by one deals with the second property, provided each new unit ignores any pattern which activates any of the earlier hidden units. In terms of signals passing between units, whenever a completed unit is ON it broadcasts a 'don't learn' signal to the new cell which is being trained. Algorithmically then, the patterns that turn a unit ON are removed from the training set of all future hidden units. The next unit then follows the same steps, slicing off a further subset of the (remaining) target 1 patterns, and removing them from the training set of any later units. This process may be continued until no target 1 patterns

[9] Both of these drawbacks can be dealt with by a simple variant of perceptron learning, described in the final section of chapter 3.

remain, in which case the layer of added hidden units forms an *OR* representation, and an output unit can easily succeed at the task simply by having positive weights incident from each hidden unit. In effect we've 'whittled' all the target 1 patterns off the hypercube, leaving all the target 0 patterns behind.

1.6.5 Producing *OR* by Splitting.

This section presents another method for developing an *OR* representation amongst hidden units. There is no need to be restricted to a single hidden layer with no connections except to input and output: it's just that there must be *some set* of hidden units for which the above two conditions hold. How this set is arrived at is another matter, and could involve other hidden units. In this case it is achieved by constructing a tree of units.

There are three stages involved in the development of this method. The first requirement is that the hidden representation must be faithful. This is dealt with by a very simple algorithm for dividing up the training set which produces a binary tree of constructed units. Secondly, the representation must ultimately be *separable*, in this case by forming *OR*. This is readily achieved with no extra training from the existing faithful representation if a control structure is assumed which allows only a selected part of the tree to respond. Thirdly the network should work without any such external 'controller' looking on. This is achievable using greater connectivity between hidden units, and at some cost in either algorithmic complexity or the size of training set each unit must learn.

Producing a faithful representation.

The preliminary aim is to split the training set up into smaller and smaller regions until each region contains patterns of either target 1 or 0 but not both. The boundaries of regions in pattern space are of course hyperplanes corresponding to hidden units, and the positions of these may be learned by the Pocket algorithm or other perceptron learning method which converges to satisfactory weights.

The first unit, say A, is trained for some amount of time on all the patterns, and effectively splits the training set $\{S\}$ into two subsets:

$$\{S_A\} \quad \text{for which } A \text{ is ON}$$

$$\{S_{\overline{A}}\} \quad \text{where } A \text{ is OFF}$$

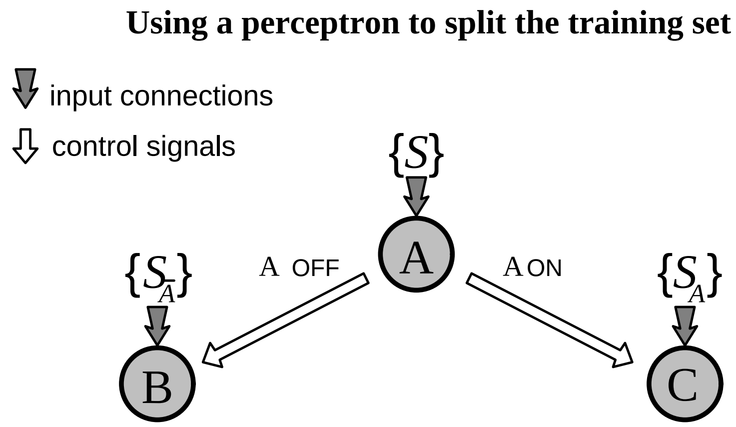

Figure 1.3: Recursive element for producing a faithful representation by splitting up the training set. The signal which is passed from A to its daughters is 'learn now'.

Note that $\{S_A\}$ now tends to contain a higher proportion of target 1 patterns to the total set size than did the original set $\{S\}$, while $\{S_{\overline{A}}\}$ contains a lower proportion. Sets consisting entirely of patterns of one target will be referred to as *homogeneous*. Therefore $\{S_{\overline{A}}\}$ is homogeneous if A is never wrongly OFF, and conversely $\{S_A\}$ is homogeneous if A is never wrongly ON.

The weights from the input into A are then frozen. If $\{S_{\overline{A}}\}$ is not homogeneous, a new unit, B is generated. Similarly if $\{S_A\}$ is not homogeneous, a unit C is included. For the purposes of training, B sees only the training set $\{S_{\overline{A}}\}$ and C sees $\{S_A\}$, but otherwise they are trained just as unit A was. This training scheme is summarised in figure 1.3. Notably the information that flows from A to the new units is a 'learn now' (or alternatively a 'don't learn now' to the other branch) signal, but the targets are the same as for A.

After training, unit B effectively splits $\{S_{\overline{A}}\}$ into two subsets: $\{S_{\overline{A}B}\}$ and $\{S_{\overline{AB}}\}$. B may now generate two daughter units of its own to deal with these subsets if they are not already homogeneous. This splitting process continues in a recursive fashion while any existing subset contains patterns of both targets, producing a binary tree of dependencies. If a given unit sees a training set which is separable, then the resulting split consists of two homogeneous subsets and no further splitting occurs. Hence such 'terminal' units make no errors on the patterns they are trained on. To show that the recursive splitting terminates, it is enough to show that each split produces two non-empty subsets. Because each of these contains fewer patterns than the original, there will eventually be pattern sets which are separable. Clearly the perceptron must not be ON for all patterns it is trained on, nor OFF; that is, the hyperplane defined by the perceptron's weights must actually cut through the convex hull formed by the patterns. Consider a perceptron with weights such that it responds OFF to every pattern in its training set. The number of errors it makes is just the number of patterns with target 1. However provided the patterns are convex, the perceptron could instead respond ON to (at least) a single target 1 pattern, and thereby make one fewer errors than before. Given that the Pocket algorithm can be used to find an optimal set of weights with arbitrary probability, the training set will always be (eventually) split in a useful way. Hence the tree will eventually terminate. Note that when the tree terminates, there is no pattern of activity amongst the hidden units which is engendered by input patterns of both targets, since this would prompt further splitting. Hence when the branching terminates the accompanying hidden representation is guaranteed to be faithful.

The above method can be used to build the ancillary units in the Tiling algorithm instead of the method used by Mézard and Nadal which requires non-local information to train these units. In the Tiling algorithm at this stage a new layer is constructed, however as the next section shows, no further units are actually required. Hence in the Tiling algorithm all the layers of hidden units beyond the first are in fact expendable.

Using a 'go-left, go-right' controller.

The representation formed by the above method is faithful but not necessarily separable, because units which were never even trained on a given input pattern may still respond to it. Hence there is no way to ascertain which active units are the significant ones. However, the same units can be used to produce a separable representation by means of a *'react now'* signal which propagates down the tree in an identical way to the 'learn now' signal used during construction of the tree.

If upon inputting a learned pattern μ, unit A is ON, then we know that unit C was trained on that pattern, whereas B was not. Similarly if C then responds OFF we know that C's left-hand daughter unit was trained on μ but its right-hand daughter was not. Therefore, of all the terminal nodes in the tree, only one saw μ during training. We would like to have a way of locating the one which was actually trained on μ since if it is a terminal node its output on presentation of μ is the target it was trained on, which is what the network is supposed to produce.

This can be accomplished by means of another kind of signal which is transferred down the tree when a pattern is presented, and which merely enables the current unit to react to the input. If A responds with ON, control should be transferred to C, which is to say that C should be able to react whereas B should be silent.

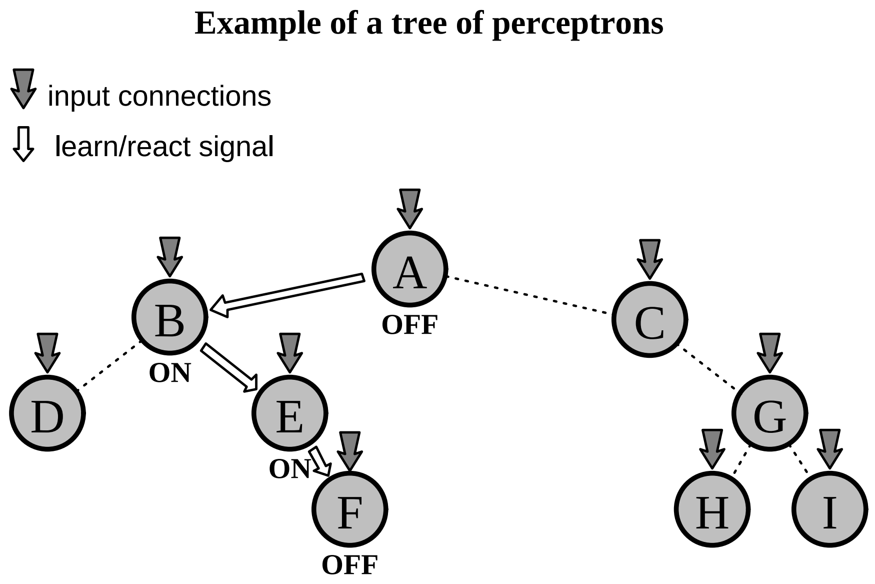

Figure 1.4: Descending the tree of perceptrons by passing a flag.

Essentially if such a 'go-left, go-right' controller operates at each branch, only one terminal unit is able to respond to the input, and this node is the unit which learned the pattern if the pattern was in the training set. This is the same as if a flag were being passed down the tree, going to the left daughter if the parent is inactive and to the right if the parent is active as shown for example in figure 1.4. This flag enables a unit to respond to the input, so all units which do not get flagged are OFF. An output unit has only to be connected to the terminal units and to detect if any of them is ON in order to respond correctly to any learned pattern (Frean 1989). That is, the representation on the terminal nodes is *OR*.

The 'react' signal must be passed down the tree *irrespective of the parents response*, which only determines which daughter it is passed to. Therefore this signal cannot be simply a function of the response of the parent unit.[10]. So if we insist on passing something down the tree in the way that trees usually work,[11] the only way to achieve this operation is by introducing another unit-to-unit signal besides output activity. Hence the network is no longer made up solely of perceptrons.

[10]This isn't true in the case of the 'learn now' signal, because only the latest terminal nodes learn, the rest being frozen, so nothing needs to be passed down the tree, and the unit responds merely to whether its parent is ON or OFF as the case may be.

[11]The algorithm at this stage amounts to a *classification tree* method (Breiman *et al.* 1984) with perceptrons as the classifiers.

Weighted connections within the hidden layer can replace the controller.

How can the above be implemented by a network of simple perceptrons? Without a controller or a 'react' signal, many of the hidden units can respond ON to a given pattern besides the correct one. The basic operation required is to turn OFF the 'don't react' side of the entire tree below each successive branch point. Eventually the entire tree except for a single terminal node must be inactive following presentation of an input pattern. Consider A,B and C, where both B and C may have subtrees below them. If A responds ON, then B *and the whole tree below it* should be strongly inhibited. On the other hand if A is OFF then C and its entire subtree should be inhibited. Then by the time the terminal node is reached[12], it is the only unit in the tree which isn't crippled by inhibition: its output is the 'decision' of the network as to the classification of the input pattern.

To achieve this requires that every hidden unit have incoming weights from all its ancestors as well as the inputs, as shown for example in figure 1.5. Supposing that unit A unit is ON, its left-hand branch is easily inhibited by large negative weights from A to all the appropriate units. If A is OFF however, the right-hand branch cannot be inhibited directly. Instead, the weights from A to the units in this branch must be positive, and each unit must acquire a negative bias such that it is definitely OFF (*ie.* inhibits itself) when A is OFF, and exactly counteracts the excitation from A when A is ON.

The weights and biases required to implement this could in principle be 'hard-wired' at the time the unit is introduced, but it is also of interest to consider how they might be learned just as the weights from the input are. To do this it is sufficient to feed the 'learn now' daughter the usual target, but set the other daughter's target to be 0. Then, since appropriate weights do exist as described above, the Pocket algorithm can be used to learn them. In this view there is no need for a 'learn/react' signal at all, since all that is required is for the right-hand [left-hand] daughter's target to be reset to 0 if the parent is OFF [ON]. However this means every unit must now learn the entire training set. This scheme for a single unit and daughters is shown in figure 1.6.

[12]this terminology suggests that the tree is traversed sequentially: in fact everything could react at once, and because hidden units influence each other in a purely feed-forward manner the process is equivalent to 'final values' moving down the tree.

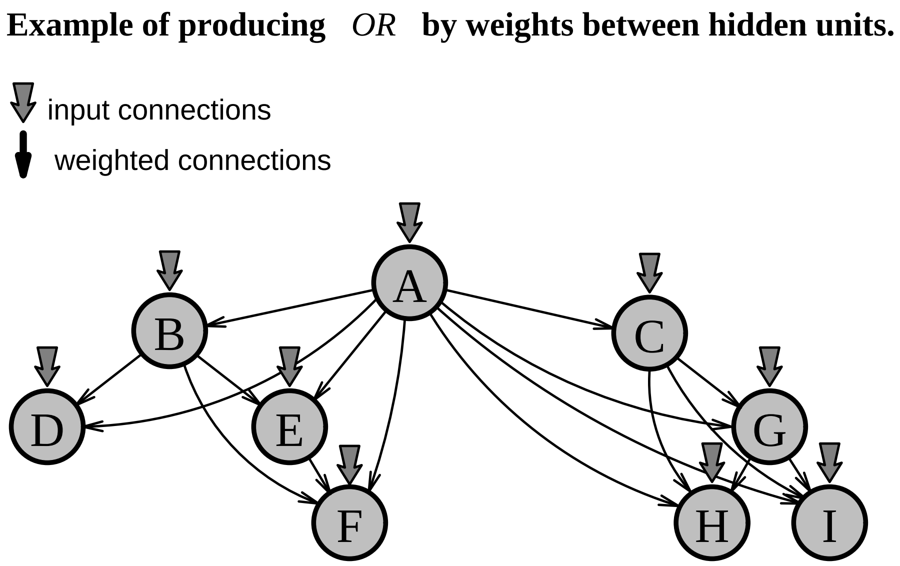

Figure 1.5: Producing *OR* by weighted connections between hidden units.

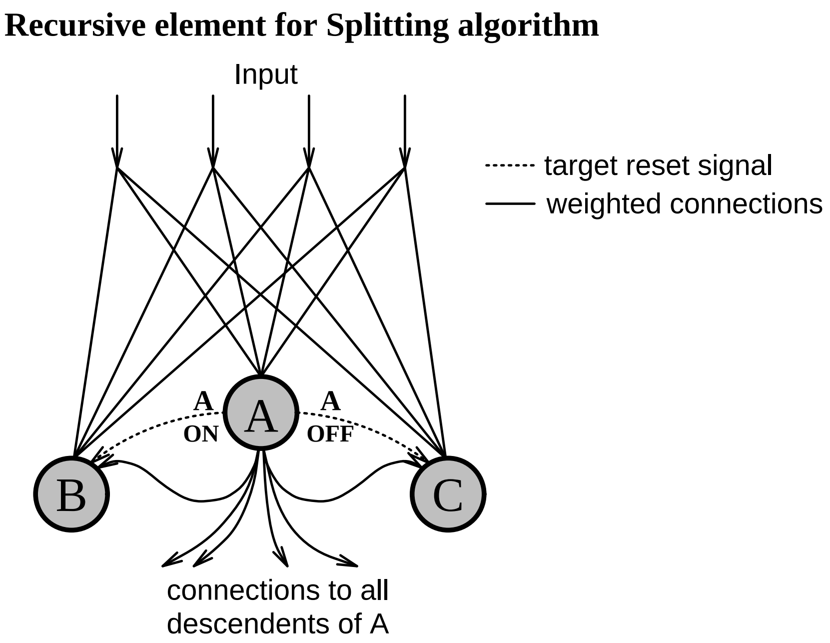

Figure 1.6: The recursive element of the Splitting algorithm.

Concurrency

It turns out that at the same time as this work was done, no fewer than four groups elsewhere independently arrived at similar algorithms. [Personnaz *et al.* 1990] generate units according to the splitting procedure, which are used as input to a further hidden layer which produces a separable representation by another (non-local) method. [Sirat & Nadal 1990] use perceptrons in a tree with a 'controller' as described above, but use the Pocket algorithm to maximise a measure derived from information theory. [Sun, Lee & Chen 1988] have also presented an algorithm (although with no convergence guarantee) in which individual units perform gradient ascent of an information measure and are used to split the training set into two parts, each of which is dealt with by a separate unit. The recursive application of this same procedure to these new nodes results in a decision tree. Again, this cannot be said to be a neural network in the usual sense because of the necessity of control other than by the units themselves. [Golea & Marchand 1990] have concurrently developed the algorithm I have described above, including feedback from all ancestors. Like Sirat and Nadal, they use the Pocket algorithm to minimise a quantity other than the total number of errors.

1.7 A Simulation: learning random mappings

In this section, the four algorithms described above are compared on a particular task: that of learning random mappings from input to output. For N inputs, the training set consists of all $P = 2^N$ possible binary patterns (this is called 'exhaustive learning' by [Solla *et al* 1990]). Each pattern is assigned its target output of 0 or 1 at random and with equal probability. Hence approximately $N/2$ patterns have target 1. This is a difficult problem, as there is no external structure to the patterns which a classifier could hope to exploit.

The weights themselves are trained by the Pocket algorithm (with ratchet). One training *epoch* is taken as the presentation of P patterns, each being chosen at random (with replacement) from the training set.

It must be noted however that this apparently even-handed method of training does not put all the algorithms on an equal footing. For example, the Whittling algorithm needs to form Pocket weights several times over in order to eliminate enough patterns to make the problem separable. In this case the number of epochs per elimination cycle is 100, and 10% of the incorrectly classified target ON patterns are eliminated after each such cycle, until convergence. For the other

other algorithms, each unit is trained for 1000 epochs.

The number of units constructed by each of the algorithms in solving the problem (that is, getting the entire training set correct) is shown in figure 1.7, plotted against the number of patterns in the training set (2^N). In their paper Golea and Marchand quote an average network size of 20.5 units for the case where $N = 6$, $P = 64$. It is possible that the Tiling algorithm might do better if the number of *weights* was considered instead of the number of units, because in this algorithm the number of units per layer tends to decrease as layers are added. However figure 1.8 shows that this is not the case for the random problem.

In this case the Tower algorithm has very limited contact between successively trained units: each new unit has a weighted connection from its immediate predecessor only (apart from those from the input). Figure 1.9 shows the mean performance of the Tower algorithm on the random problem with $N = 6$, as the number of predecessors available to new units is increased. There is no strong benefit obtained in terms of the number of units constructed by having greater numbers of contacts in the random case.

1.8 Comparisons between constructive methods.

In this section a number of key points on which constructive algorithms may be compared and contrasted are suggested. The Tiling algorithm is omitted from the discussion, as it is essentially a combination of the Tower and Splitting methods.

Optimal performance on parity. It is not difficult to show how many units each algorithm will generate in solving the parity problem if it finds "perfect" weights at every stage. In solving N bit parity, the Tower algorithm builds $\frac{N+1}{2}$ units if N is odd, and $\frac{N}{2}+1$ if it is even. The Splitting algorithm is less efficient, building $N+1$ units. Whittling can do no better than 2^{N-1} units, one for each target ON pattern. The Tower algorithm's success arises from its ability to use previously constructed units to deal simultaneously with patterns of *both* targets, which neither of the other methods can achieve. Whether this property has advantages or disadvantages for other problems is an open question.

Complexity of the architecture. Whittling and Splitting distinguish hidden units from the output unit, whereas the Tower algorithm does not.

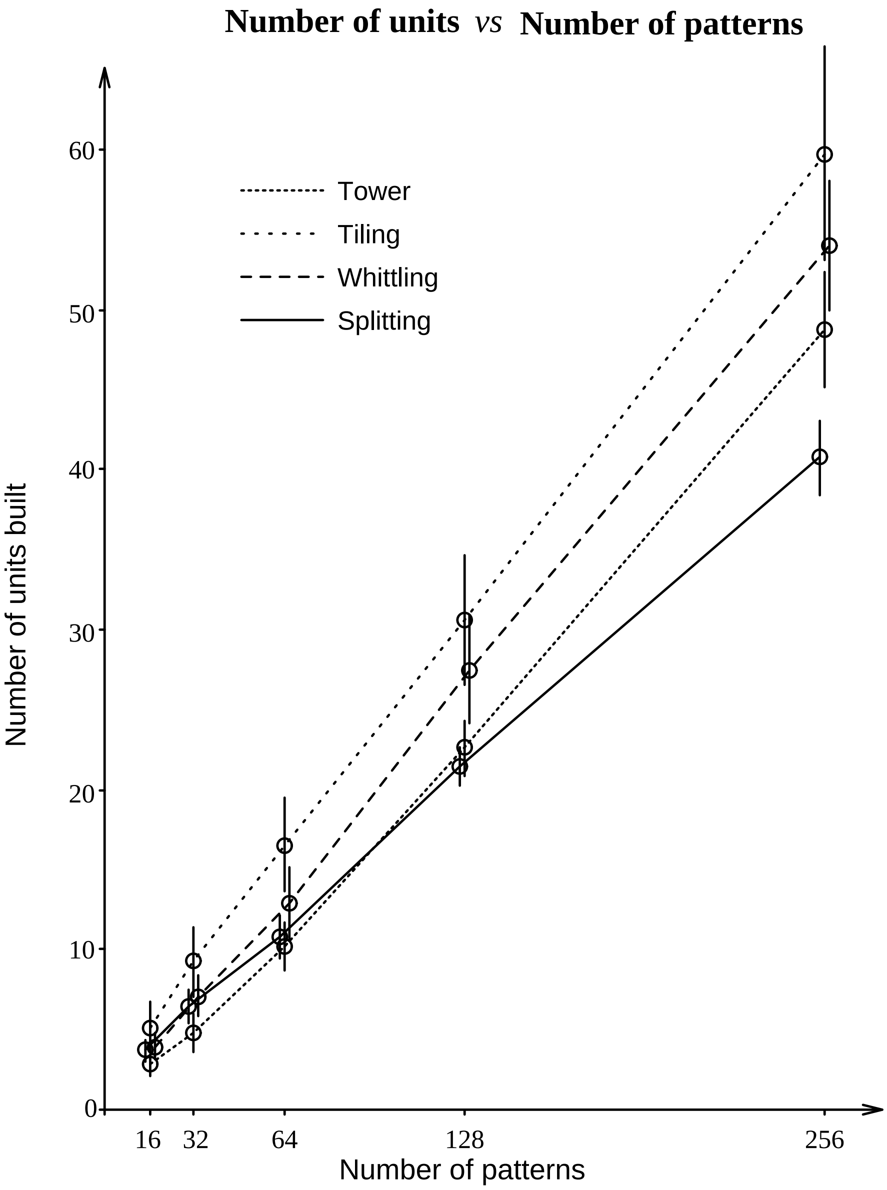

Figure 1.7: The number of units constructed by each algorithm is shown, plotted against the size of the training set. The circles and vertical bars show the mean and standard deviation respectively over 25 separate trials, each on a different training set.

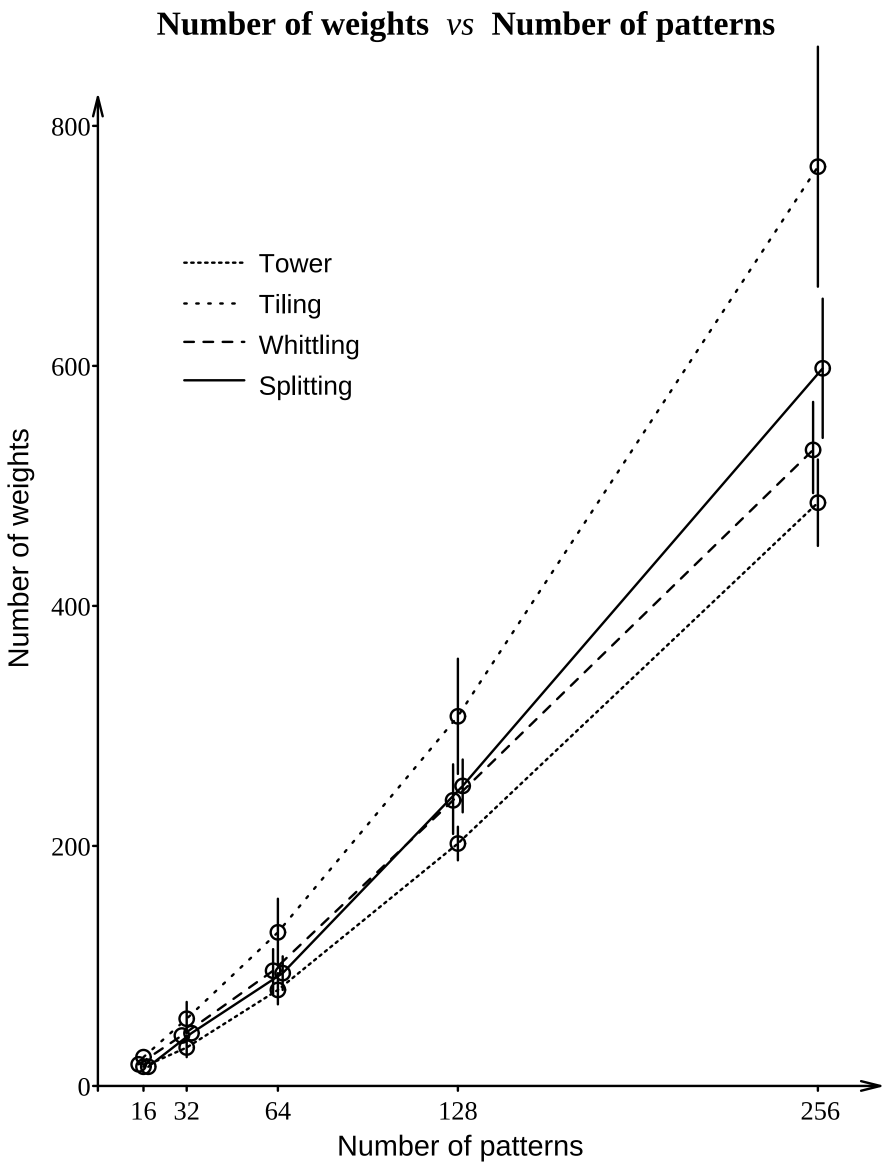

Figure 1.8: The number of weights in networks constructed by each algorithm is shown, plotted against the size of the training set. Statistics are over 25 separate trials, each on a different training set.

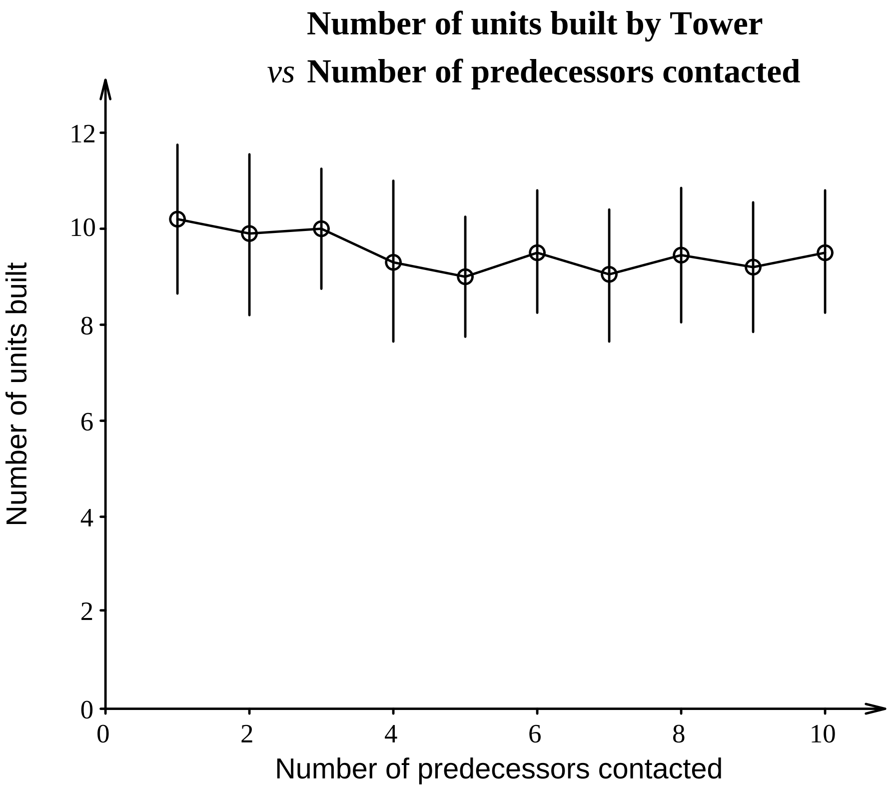

Figure 1.9: Network size generated by the Tower algorithm for the case of $N = 6$. The abscissa shows the number of previously trained units that each new unit can have weights from (in addition to those from the input).

Signals passed between units during learning. The Tower algorithm is very simple, since each unit only needs the external target signal. Whittling requires a *don't learn* signal, to be communicated between active hidden units. Units in Splitting must interfere with targets of other units deeper in the tree, effectively *over-riding* external target 1 signals for some patterns.

Number of units which learn separable training sets. The difficulty of the training problem presented to units may differ between methods. For example in Tower and Whittling, no unit except the last one sees a separable training set, whereas in Splitting many units solve linearly separable problems.

Exploitation of reduced training sets. If the classification to be learned is available as a training set rather than arriving "on line" from the environment, some methods can train units on successively smaller training sets. This is not true of the Tower algorithm. Whittling can potentially eliminate some patterns from future training sets. Splitting can heavily exploit reduced training sets, but only if the weights between hidden units and the adjustments to the biases do not need to be learned.

Parallelism in learning. One possible advantage of any tree-building method is that several units can learn at once. In Splitting the number of units that can learn in parallel potentially rises exponentially with the tree depth.

Parallelism in recall. Given a network of say H units, the different architectures will engender different "response times". The Tower architecture requires H time steps before the output unit can be read. Splitting will require a number of time steps equal to the depth of the tree. Since Whittling constructs a single hidden layer, it takes only two time steps to produce an output.

Sequence of training of the units. If connections between constructed units are disrupted, the unit giving fewest errors will be the unit which was trained on the original targets, and saw only the inputs. In none of the methods so far discussed is this the output unit.

All the constructive algorithms discussed in this chapter apparently have broadly similar performance. The number of hidden units built is not dramatically different from the number of patterns being learned; for larger sized problems the number of units may reach unmanageable proportions. In addition, problems of larger scale cannot be attempted because the training time required to give convergence becomes prohibitive. This demands a more efficient constructive algorithm. Such an algorithm is described in the next chapter.

Chapter 2

Upstart algorithm

This chapter describes a different method for constructing and training feed-forward networks of perceptrons which is based on error correction not at the level of *weights* but of *units*. The core idea is that a unit may recruit and train other units specifically in order to correct its mistakes. The Upstart algorithm is described and is found to generate networks which have small size and can generalise to novel input patterns. This performance is significantly better than the constructive algorithms described earlier.

2.1 Rationale

In making a binary classification, any unit (say Z) can make two kinds of mistake, by being

$$\text{"wrongly ON "} \quad (o_Z^\mu = 1, \text{ but } t_Z^\mu = 0)$$

$$\text{"wrongly OFF"} \quad (o_Z^\mu = 0, \text{ but } t_Z^\mu = 1)$$

where o_Z is the perceptron's actual output and t_Z is its desired or target output. Suppose there exists a unit (say X) which is ON for every pattern for which Z is wrongly ON, but is otherwise OFF. Similarly suppose there is a unit (say Y) which is ON when Z is wrongly OFF but not otherwise. By means of a large negative weight from X and a large positive weight from Y, all of Z's errors could be corrected. The presence of the two extra inputs from X and Y makes the problem linearly separable (since it is now possible for Z to make no errors), so the appropriately large weights can easily be learned by the PLR. X and Y's role is to effectively *override* whatever Z's "raw" response to the input pattern was. The problem then becomes how to produce units X and Y.

2.2 Training units to act as correctors

Assuming the two new units are also connected to the input layer by variable weights, they can be trained using targets which depend on Z's response. These units might be called "daughters" since they are generated by the established "parent" unit, Z. Note however that the direct effect of *activity* proceeds from daughter to parent. Consider, for example, the targets we should assign to X, the unit whose role is to inhibit Z. We would like X to be active if Z was wrongly ON, and silent if Z was correctly ON. Similarly X should be silent if Z was wrongly OFF (to avoid further inhibition of Z). Finally, X could be silent if Z was correctly OFF, although if X is active in this case, the effect is merely to reinforce Z's response when it was *already correct*. This doesn't itself cause an error, meaning that it doesn't matter how X responds. Therefore these patterns can be eliminated from X's training set. This elimination makes the problem easier and faster to solve, but is not essential for the error-correcting property described below. Similarly, Y should be trained to be ON only when Z is wrongly OFF, but if Z is correctly ON the pattern can be eliminated from Y's training set. These target assignments are summarised in Figure 2.1.

An important point is that the "raw" output of unit Z is used to set the daughter's targets, rather than the value of Z after the daughters have exerted any effect, since this would introduce feedback. To achieve this the sum from the true inputs alone must be available even though the daughters are exerting their effect (as indeed it is, since the inputs from daughters arrive in the next time step). Alternatively the connections from daughters can be built up only after the direct input weights have been learned and "frozen".

Two useful results follow immediately from this training method, because it essentially gives daughters (X or Y) an easier problem to solve than their parent (Z).

Property I: Daughters can always make fewer errors than their parent.

Denote Z's errors by

$$e(Z) = e(Z)_{\mathrm{ON}} + e(Z)_{\mathrm{OFF}}$$

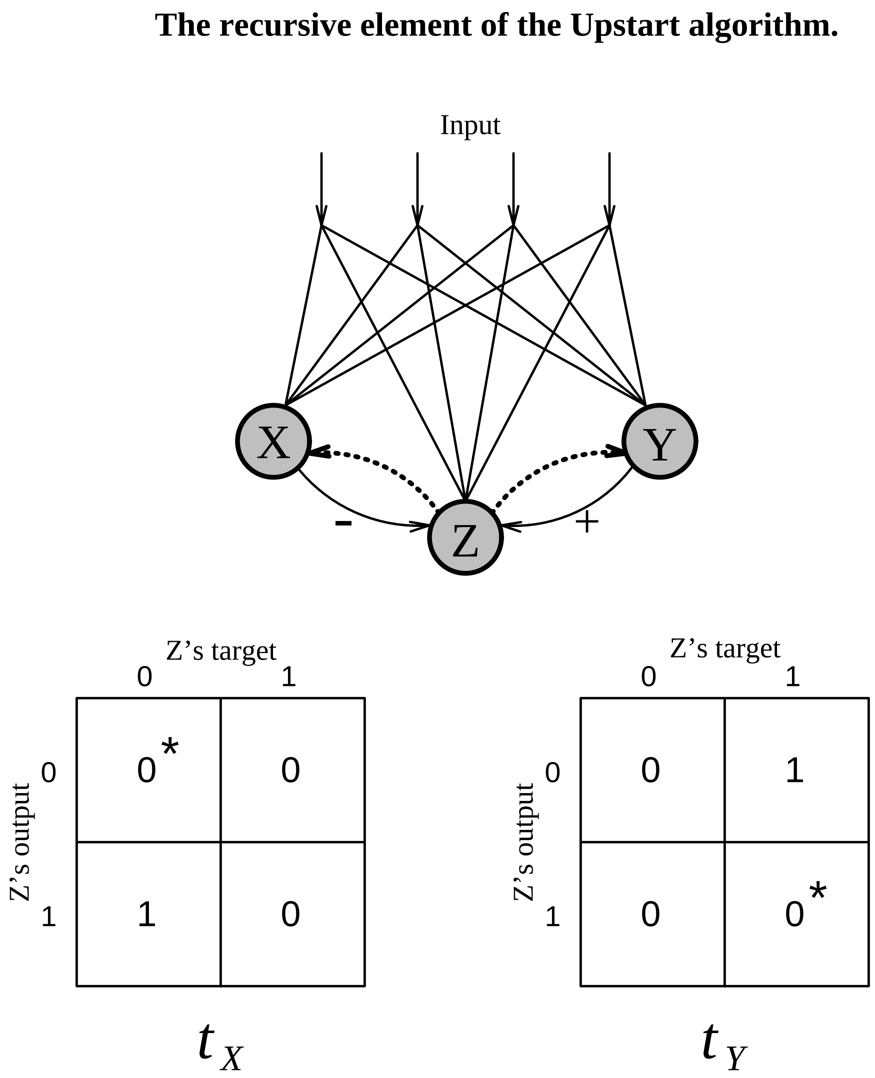

Figure 2.1: Correcting a parent unit: the left hand table gives the targets, t_X, for the daughter unit X for each combination of (o_Z, t_Z). For example, the lower left-hand entry assigns t_X to be 1 when $o_Z = 1$ and $t_Z = 0$: the 'wrongly ON' case. Similarly the right hand table gives the values of t_Y for the daughter unit Y. The dotted line represents the flow of this target information. The "starred" entries correspond to cases where the pattern could be eliminated from the daughter's training set.

where $e(Z)_{\text{ON}}$ is the number of patterns for which Z is wrongly ON.
If X responded OFF to every pattern, it would make as many errors as there were patterns of target $t_X = 1$. However, X can always do better than this, provided the training set is convex. Given that the Pocket algorithm can find the optimal weights visited by a perceptron with any given probability, at the very worst weights could be found such that a single pattern for which $t_X = 1$ turns unit X ON whereas all other patterns turn it OFF. Therefore

$$e(X) \ < \ e(Z)_{\text{ON}} \ \leq \ e(Z) \tag{2.1}$$

A similar argument applies to Y.

Property II: Connecting daughter to parent with the appropriate weight can always reduce the errors made by the parent.

It follows from the above that Z's errors are reduced by X, since

$$\begin{aligned} e(Z \text{ with } X) \ &= e(X)_{\text{ON}} \ + \ e(X)_{\text{OFF}} \ + \ e(Z)_{\text{OFF}} \\ &= e(X) \ + \ e(Z)_{\text{OFF}} \\ &< e(Z) \end{aligned} \tag{2.2}$$

and similarly for Y on its own. When the joint action of X and Y is considered, the same result holds, i.e. $e(Z \text{ with } X{,}Y) \ < \ e(Z) - 1$.

In the next section an algorithm which uses the first of these results is described. Other possibilities are discussed in section 2.5.

2.3 Upstart as a binary tree.

Assume we already have a unit Z which sees input patterns $\xi_i^\mu : i = 1, .., N$ and has associated targets t_Z^μ. The weights from the input layer to Z are trained to minimise the discrepancies between Z's output and target and once trained, these weights remain frozen. This "first" unit is actually the eventual output unit, and its targets are the classification to be learned. The following two steps are then applied:

<u>UPSTART AS A BINARY TREE.</u>

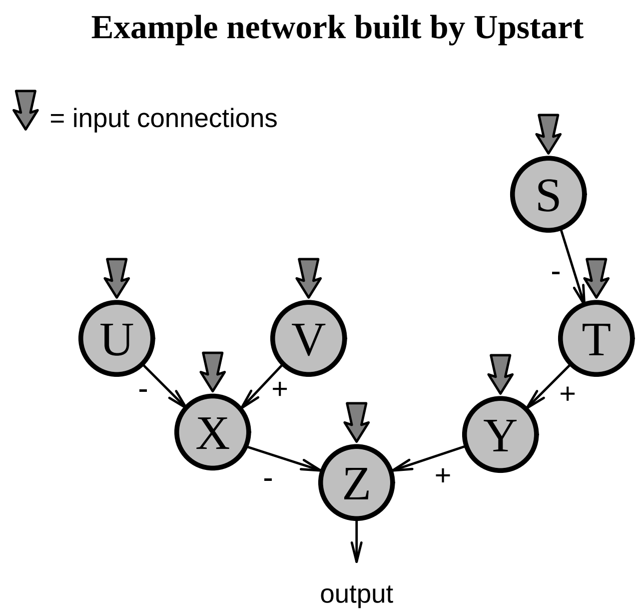

Figure 2.2: Example of a binary tree of units constructed by the Upstart method.

Step 1. If Z makes any "wrongly ON" mistakes, it builds a new unit X, using the targets given in Figure 2.1. Similarly if Z is ever "wrongly OFF" it builds a unit Y. Apart from the different targets these units are trained and then frozen just as Z was.

Step 2. The outputs of X and Y are connected as inputs to Z. The weight from X is large and negative whilst that from Y is large and positive. The size of the weight from X [Y] needs to exceed the sum of Z's positive [negative] input weights, which could either be done explicitly or by using the Perceptron learning rule to learn these weights.

Steps 1 and 2 are now applied recursively to X and Y in place of Z. Thus daughter units behave just as Z did, constructing daughter units themselves if they are required. In this way a binary branching tree of connected units is constructed, as shown for example in figure 2.2. Each unit gets targets from its parent rather than from an external signal, and each is capable of acting both as "pupil" and "teacher".

New units are only generated if the parent makes errors, and the number of errors decreases at every branching. It follows that eventually none of the terminal daughters makes any mistakes, so neither do their parents, and neither do *their* parents and so on. Therefore every unit in the whole tree produces its target output, including Z, the output unit. Hence the classification is learned.

2.3.1 Equivalence of the tree and layer architectures.

The architecture generated by this procedure is unconventional in that it has a hierarchical tree structure. However in the case where we choose not to eliminate any training patterns there is an equivalent structure with the same units arranged as a single hidden layer.[1] To see this, consider two daughters (say X,Y) and their parent (Z). With primes denoting "corrected" values, the corrected value o'_Z is always equal to $o_Z - o'_X + o'_Y$. Now compare these two values in each of the eight possible combinations of o_Z, o'_X and o'_Y:

CASE	o'_X	o'_Y	o_Z	o'_Z	$o_Z - o'_X + o'_Y$
i	0	0	0	0	0
ii	0	0	1	1	1
iii	0	1	0	1	1
iv	1	0	1	0	0
v	1	0	0	0	-1
vi	0	1	1	1	2
vii	1	1	0	undetermined	0
viii	1	1	1	undetermined	1

In cases *v-viii* the values in the last two columns don't agree, but none of these cases actually occurs. X would never be correctly ON if Z was OFF (case *v*), and similarly Y would never be correctly ON if Z was ON (case *vi*). Finally, X and Y would never be correctly ON together (cases *vii* and *viii*).

Since this equivalence between o'_Z and $o_Z - o'_X + o'_Y$ holds for *every* unit in the tree, the final output is simply a sum of the "raw" responses. For example in the case of the network shown in figure 2.2:

$$\begin{aligned}
\text{output} &= o'_Z \\
&= o_Z - o'_X + o'_Y \\
&= o_Z - (o_X - o_U + o_V) + (o_Y + o'_T) \\
&= o_Z - o_X + o_U - o_V + o_Y + o_T - o_S
\end{aligned}$$

Imagine the tree units disconnected from one another and placed in a single layer. A new output unit connected to this "hidden" layer can easily calculate the appropriate sum by, for example, having weights of +1 from each unit which adds to the sum and −1 from each unit that subtracts, with a bias of zero. In effect we can convert a binary tree into a single hidden layer architecture which implements the same mapping, at the expense of adding one unit and being unable to exploit pattern elimination. Figure 2.3 shows the example network converted in this way. The algorithm for constructing a single hidden layer architecture is:

[1] I am grateful to Peter Dayan for pointing this out.

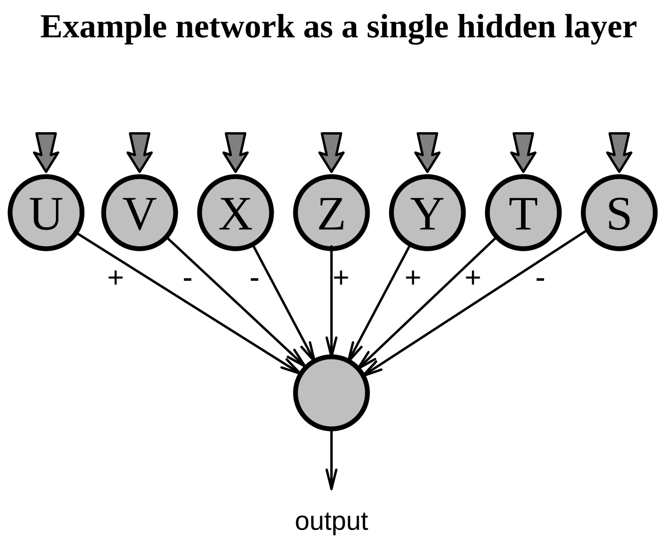

Figure 2.3: Example of conversion to a single layer. The hidden units are the same as those shown in the earlier figure *ie.* their weights to the input are the same.

<u>UPSTART AS A SINGLE LAYER (I).</u>

Construct units as before, omitting Step 2 (where they are connected into a feed-forward tree). Then connect all the units so constructed to a new output unit. The weights to this unit can be learned by PLR (since in this representation the patterns are clearly linearly separable), or can be inferred from the tree structure: there is a sign reversal for every "X-type" daughter.

2.4 Simulations

In all the simulations shown here the "starred" entries in Figure 2.1 were not included in a daughter's training set. If the whole training set is used in every case, the number of units produced is relatively unaffected for the problems investigated here, but the training time (a combination of the time per epoch and number of epochs required to generate a comparable network) is approximately doubled.

A learning procedure called the *Thermal perceptron learning rule* was used to learn the weights. In this, the weight changes given by the usual PLR are simply multiplied by

$$\frac{T}{T_0} \exp\left(\frac{-|\phi|}{T}\right)$$

Unless otherwise stated the "temperature" T_0 was initially set at 1.5 and reduced to zero linearly over 1000 epochs. A full discussion of this rule is deferred until chapter 3, where it is dealt with in detail.

Parity

In this problem the output should be ON if the number of active inputs is odd, and OFF if it is even. Parity is often cited as a difficult problem for neural networks to learn. It is also of interest because there is a known solution consisting of a single layer of N hidden units projecting to an output unit (Minsky & Papert 1969). It is easy to see how the Upstart Algorithm tackles parity (see figure 2.4). Essentially the same structure as that shown for N=3 would arise for any N, although for large problems the optimal weights become much harder to find. For parity up to N=10, in all cases N units are constructed, including the output unit. In all cases except N=10, a thousand epochs were sufficient to generate the minimal tree. For 10-bit parity, the figure was 10,000.

Random mappings on the complete set of binary patterns

In this problem the binary classification is defined by assigning each of the 2^N patterns its target 0 or 1 with 50% probability. Again this is a difficult problem, due to the absence of correlations and structure in the input for the network to exploit. The networks obtained for N up to 10 are summarised in Figure 2.5. The Tower algorithm requires a prohibitively large number of training epochs by the Pocket algorithm to converge for this problem for $N > 8$. Instead, results for the Tiling algorithm are included for comparison, where the Pocket algorithm is used to learn the weights.

Random mappings on random patterns lying on a hypersphere.

This experiment looked at the classification of patterns with real-valued components. Such a set of patterns do not in general have the essential property of convexity, however this property holds if the patterns are constrained to lie on the surface of a hypersphere. In the first case up to 1000 real-valued patterns across $N = 10$ inputs were generated, and in the second case up to 5000 patterns across 100 inputs. Each of the patterns was generated as follows. Firstly, each of the N inputs is chosen at random, uniformly in the interval [-1,1]. This gives patterns distributed uniformly in the unit hypercube. Secondly, each such pattern is normalised to make $\sum_i \xi_i^2 = 1$, and is assigned a target 0 or 1 with equal probability. Note that although these pattern vectors lie on the surface of the unit hypersphere in N dimensions, they are not distributed uniformly over this surface.

The size of the networks generated in solving this problem to zero errors using the

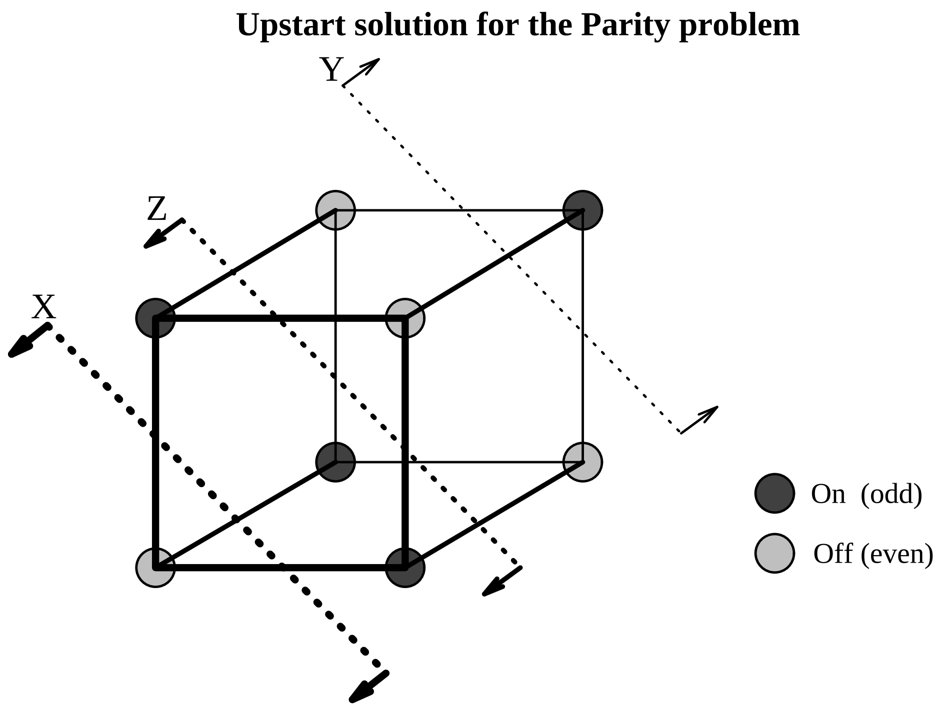

Figure 2.4: Solution for 3-bit Parity. The output unit Z on its own can clearly make a minimum of two mistakes, when the plane defined by its weights cuts the cube as shown. X corrects the wrongly ON pattern by responding to it alone, and similarly Y corrects the wrongly OFF pattern.

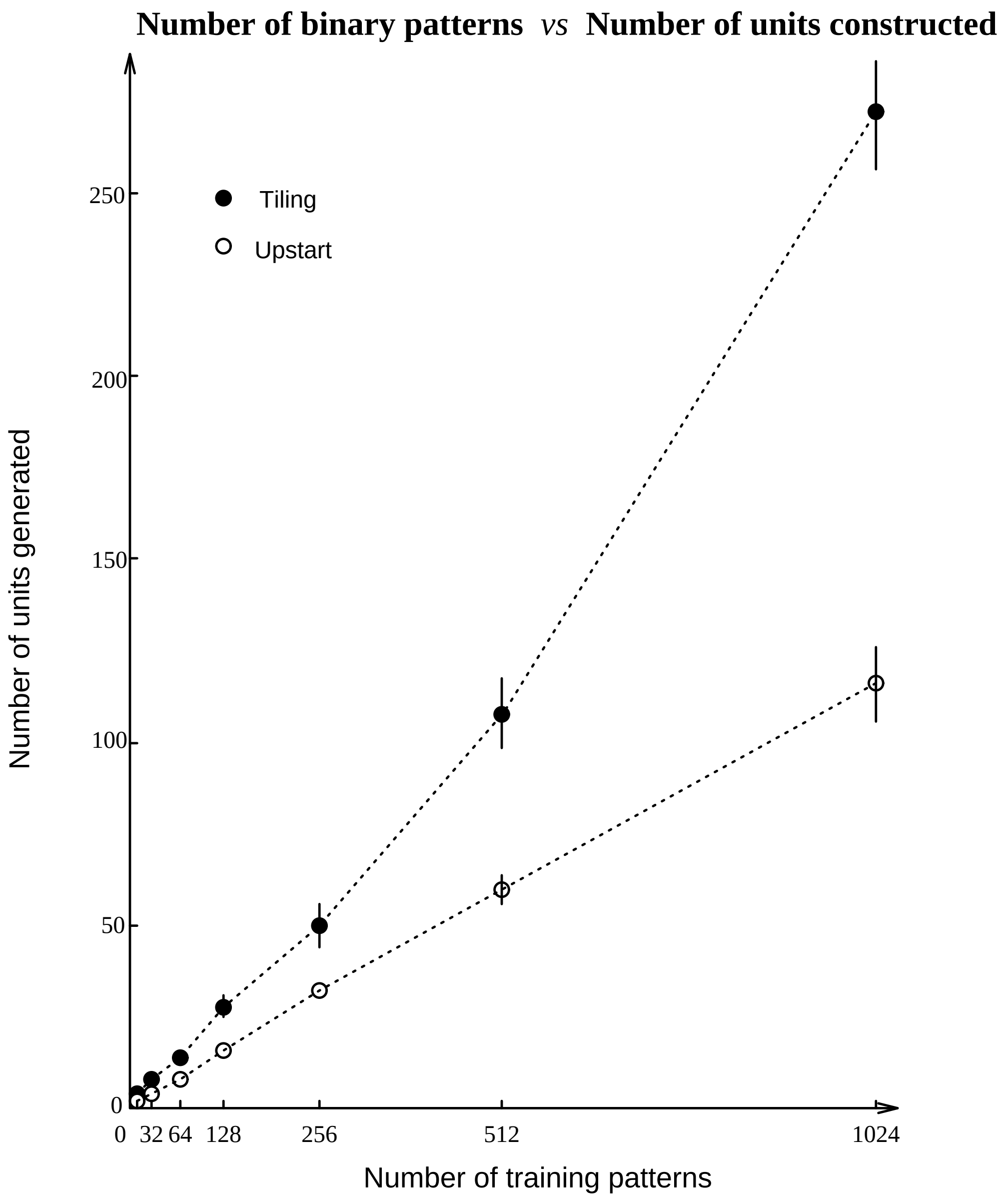

Figure 2.5: Number of units built vs the number of patterns (2^N) for the random mapping problem. The slope of the Upstart line is approximately 1/9. Each point is an average of 25 runs, each on a different training set.

Upstart algorithm are shown in figure 2.6. Even training with large numbers of patterns, the slope of the line remains constant at approximately 1/15 for $N = 10$, and 1/100 for $N = 100$. In other words the "pattern capacity" is respectively fifteen and one hundred patterns per unit, under the training conditions described. This capacity may be compared with the theoretical result that $2N$ is the largest number of such patterns which can be expected to be separable.

Using a related method called stereographic projection and a truly parallel implementation of the algorithm, [Saffery 1990] has solved the "two spirals problem" (Fahlman & Lebiere 1990) using Upstart.

Generalisation: the "two-or-more clumps" problem

Neural networks are often ascribed the property of generalisation: the ability to perform well on all patterns taken from a given distribution after having seen only a subset of them. Several workers (Denker et al. 1987; Mezard and Nadal 1989) have looked at generalisation using the "2-or-more clumps" predicate. The problem is this: given an input pattern, respond ON if the number of clumps is 2 or greater and OFF otherwise, where a "clump" is a group of adjacent 1's bounded on either side by 0's. Circular boundary conditions are used: input 1 is "adjacent" to input N. As with parity, there is a solution consisting of a single hidden layer of N units which would solve the problem exactly. For instance each hidden cell can be associated with one of the N adjacent pairs of inputs, and simply detects the presence of a (leading) edge by means a positive weight from the lower indexed input and a negative weight from the higher one. The number of active hidden units is then just the number of clumps present in the input pattern, and the output unit need only detect if this number is 2 or greater (for instance by setting all hidden-to-output weights at $+1$ with a bias -1). Following Mezard and Nadal, the patterns were generated by a Monte Carlo method (Binder 1979) such that the mean number of clumps is 1.5. I used $N = 25$ inputs, for which there are $N(N-1)+2 = 602$ possible patterns with less than two clumps. Training sets consisted of up to 600 patterns. The set used to test the resulting net's ability to generalise was a further 600 patterns. The results, with comparisons to the Tiling Algorithm, are summarised in Figure 2.7. [Nadal 1989] compared the performance of the Tower algorithm with Tiling on this problem, and found very similar performance in terms of numbers of units and generalisation ability, but with Tower generating approximately twice the number of weights. This is in contrast to the random binary problem investigated in chapter 1, where the number of weights is not dissimilar in the two methods.

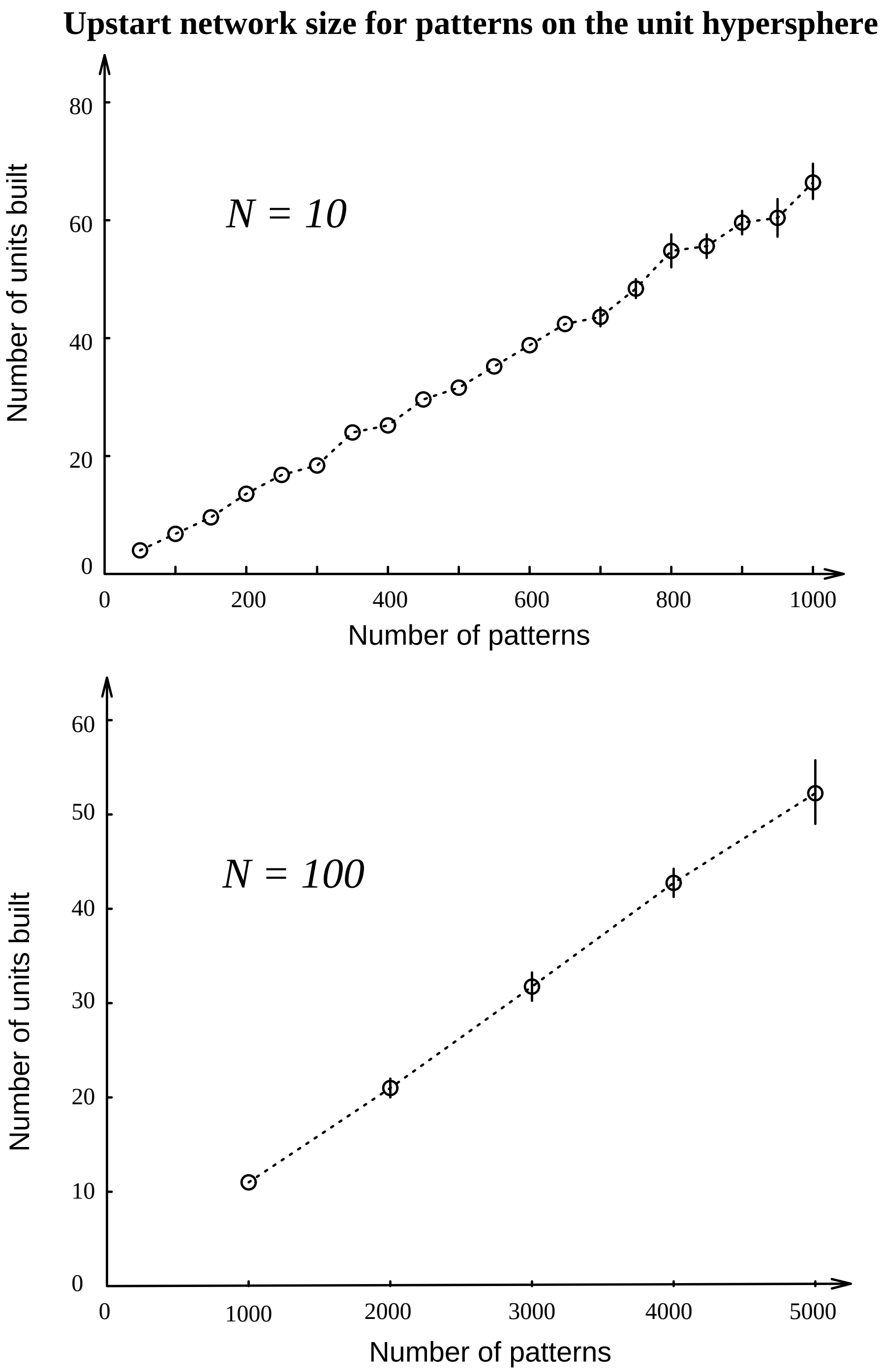

Figure 2.6: Number of units built by the Upstart algorithm *vs* the number of patterns for the random mapping problem on the unit hypersphere. Each point is an average of 25 runs, each on a different training set. T_0 is 0.5

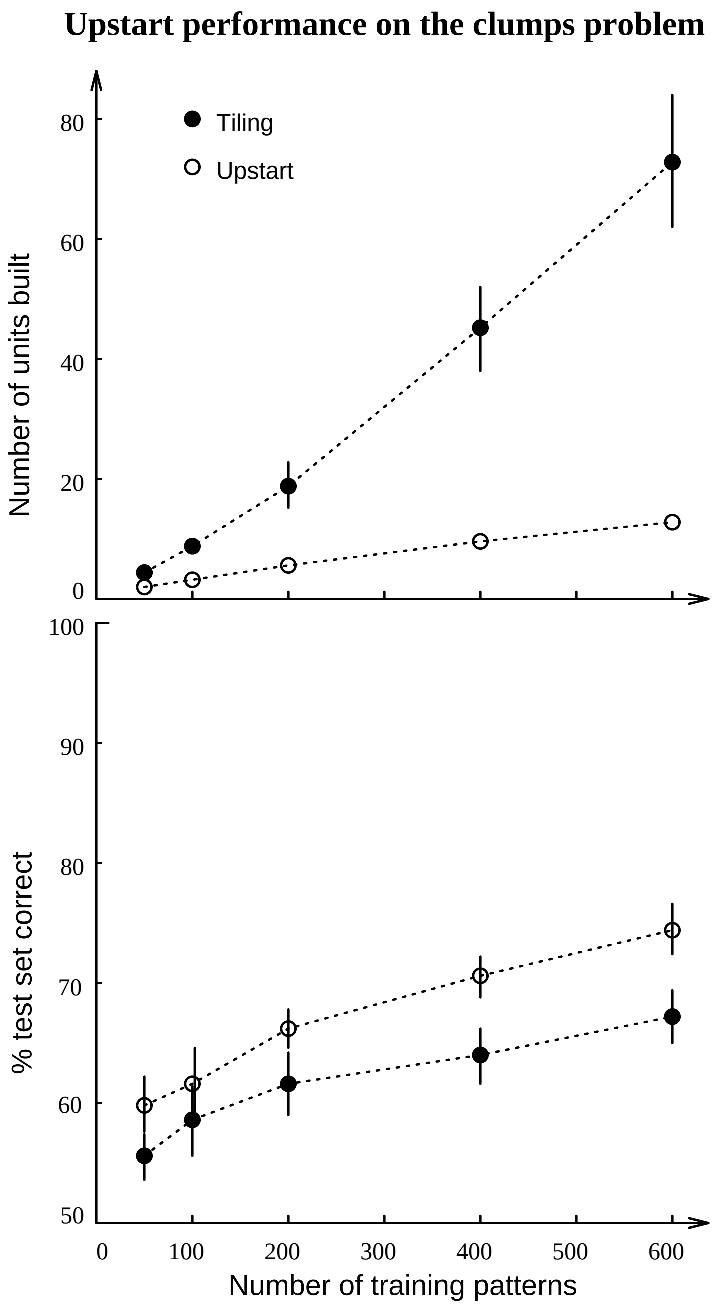

Figure 2.7: Performance of the method on the "2-or-more clumps" problem. The lower graph shows the % generalisation as the size of the training set is increased. Plotted above this and with the same abscissa is the size of the corresponding network. $T_0 = 4.0$. There were 25 runs per point, each on a different set. Where not shown, error bars are smaller than the points.

2.5 Alternative architectures

The algorithm described above constructs a binary tree of units, and uses only the first result given in section 2.2, namely that daughters can make fewer errors than their parent. However the second result can be used to generate alternative architectures using the same idea.

Consider the following algorithm:

<u>UPSTART AS A SINGLE LAYER (II).</u>

Start. Begin with an output unit with no connections to the inputs. Trivially this unit makes only "wrongly OFF errors and as many errors as there are target 1 patterns.

Step 1. Evaluate the errors made by the output unit. If $e(out) = 0$ then STOP.

Step 2. If $e(out)_{\mathrm{OFF}} > e(out)_{\mathrm{ON}}$ generate a Y unit in the hidden layer with connections to the inputs, and train it as described previously. Otherwise do the same for an X unit.

Step 3. Build a weight from the new hidden unit to the output unit, positive for a Y, negative for an X. The weight must be large enough to override any other input which the output unit may receive. Return to step 1.

Hence a single hidden layer is constructed. Since the errors made by a *parent* are reduced by an appropriately trained daughter (even though the daughter may itself still be in error), each new hidden unit can reduce the output unit's errors by at least one.

The third step can be accomplished by explicitly assigning a weight of the appropriate sign whose magnitude is sure to be greater than the sum of the hidden to output weights of the opposite sign. Alternatively the new weight can be learned in the usual way.

Note that this method results in an apparently exponential rise in the magnitudes of hidden to output weights, due to the "override" operation that each unit must perform. However this need not be a restriction since hybrid methods will also work. Since each unit has an explicit target, we can decide at any stage which unit to correct: a single hidden layer is generated by always correcting the parent, and a binary tree is built by always correcting the (terminal) daughters. Trees of variable width can be built by including both these possibilities. For instance an initial layer of say h units could be constructed as above, and then each of these hidden units could build (up to) h units of its own, and so on.

An important potential advantage of building such "wide" trees concerns the parallelism of the learning process: each of the first generation can go about training daughters independent of the others, so there can be h units learning in parallel (*ie.* $h = 2$ in the binary tree version). For the second generation of daughters there is a potential for training h^2 units in parallel, and so on.

2.6 Extension to multiple outputs

These algorithms can be extended to problems involving multiple output units. A good method should build considerably fewer units than would be obtained by treating each output separately (especially if the output targets are correlated); in other words, maximum mutual use should be made of hidden units. The following algorithm could be used, where steps 1 and 2 are repeated until every output unit makes no mistakes :

MULTIPLE OUTPUTS.

Start. There are no hidden units and no connections, so the output units are always OFF.

Step 1. Choose an output unit (say, the one which makes the most errors). Build the appropriate hidden unit to correct some of the mistakes being made by this output unit, as described above. Connect this new unit to *all* the output units.

Step 2. Train the weights from each unit in this enlarged hidden layer to each of the output units. Re-evaluate the numbers of errors made by each output unit.

Hence a single hidden layer is constructed. This is one possible method for the case of multiple outputs, and represents work to be done in the future.

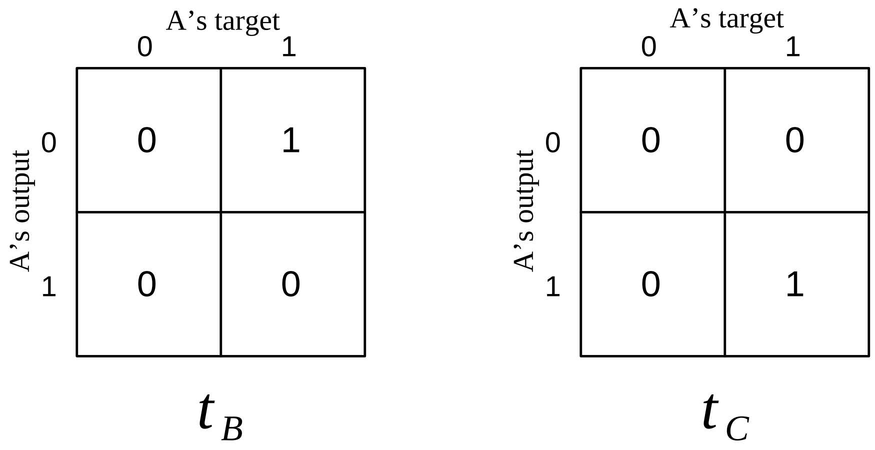

Figure 2.8: Targets assigned to daughter units by the Splitting algorithm: the left hand table gives the targets, t_B, for the daughter unit B for each combination of (o_A, t_A). Similarly the right hand table gives the values of t_C for the daughter unit C.

2.7 Are the tree building methods the same?

The Upstart algorithm and the procedure for obtaining *OR* by Splitting are clearly very similar, even though their motivations are very different: the strategy of Splitting is to "divide and conquer" whereas that of Upstart is "enlist help". Notably the criterion determining inclusion of new units is identical. However the two algorithms are not the same. Notably,

- the training sets learned by units in the two methods are different. This can be readily seen by putting the targets learned by daughter units in Splitting into the same form as those for Upstart (figure 2.1), as is shown in figure 2.8. The two are not equivalent.
- the connectivity between constructed units is very sparse in Upstart (one connection per unit), whereas it needs to be relatively dense in Splitting.
- if both methods are trained without any elimination of patterns, then the Upstart network can be transformed into a single layer, whereas the Splitting network cannot.

2.8 Conclusion

In this chapter the Upstart constructive algorithm has been described, and seen to perform well on large problems, and produce fewer units than other types of constructive algorithm. For example, in the random mapping problem Upstart builds half as many units as the Tiling algorithm, and in the clumps problem the ratio is one fifth.

Aside from the simulation results, Upstart may be compared with other algorithms along the lines suggested in section 1.8 in the previous chapter. In solving N bit Parity, N units are constructed. Upstart does not distinguish algorithmically between hidden and output units: these follow exactly the same training procedure. The signals passed between units are *targets*, rather than "don't learn" or "over-ride" signals. Because Upstart alters the training sets seen by successive units, many units learn separable training sets. Another advantage is the potential to eliminate patterns from the training set of successive units. In addition, many units can learn in parallel, especially if trees of appropriate width are developed as indicated in section 2.5. Regarding the time required for the output to respond to an input pattern, Upstart as a tree will take a number of time steps equal to the maximum depth of the tree. However, when implemented as a single layer architecture only two time steps are needed. Finally, the tree version of the Upstart algorithm is unique among the constructive algorithms considered, in that the output unit is the first unit to be trained. This feature is intuitively appealing, and also confers a degree of robustness to the networks built; if all the connections between constructed units were lost, the unit which would make the smallest number of errors on the training set is in fact the output unit. This is not true of the constructive methods described earlier.

Just as for all constructive algorithms, a suitable rule for learning the weights of the individual units is required. During the course of the work on Upstart, a new learning rule was developed which may play a crucial role in the success of the method. In the next chapter this learning rule is looked at in more detail, and the spectrum of possible rules is investigated.

Chapter 3

Learning in single Perceptrons

3.1 Introduction

Suppose we are given a binary classification to be learned by a single perceptron unit, wherein each input pattern of N binary values has an associated target, being zero or one, which is the desired output of the perceptron.

A very simple learning rule is the associative rule:

$$\begin{aligned} \Delta^\mu W_i &= \alpha \hat{t}^\mu \, \xi_i^\mu \\ \text{where } \hat{t}^\mu &= \begin{cases} 1 & \text{if} \quad t^\mu = 1 \\ -1 & \text{if} \quad t^\mu = 0 \end{cases} \end{aligned} \tag{3.1}$$

Alpha is a small positive constant controlling the rate of learning. This rule builds an association between pattern $\vec{\xi}^\mu$ and its (signed) target $\hat{t}^\mu$. Starting with all the weights at zero strength, after one presentation of each pattern, the i^{th} weight corresponds to the correlation between the i^{th} input and the target. Subsequent pattern presentations do not improve the weights vector any more than this, since the weight change is independent of the unit's history. More powerful learning rules involve the actual output of the unit, which effectively enables previous learning to be taken account of. For example, there is no need to change the weights if the unit's output is already correct on a given pattern. Historically there are two ways that this has been done: rules derived to minimise a global error measure, and the Perceptron Learning Rule.

3.2 Rules derived from global error minimisation.

On presentation of pattern ξ^μ, a unit sums up its weighted inputs to give ϕ^μ, and its output is some function $y(\phi)$ of this quantity. If the output function is differentiable, then one approach to learning is to consider the total error attributable to the unit in its present state,

$$E = \sum_\mu E^\mu$$

where E^μ is some measure of how well the unit's actual output y^μ matches the desired output t^μ. The most common forms used for E are the *squared* error

$$E^\mu = (t^\mu - y^\mu)^2$$

from classical statistics (giving rise to "least-mean-squared" or *LMS* methods), and the *cross entropy* error

$$E^\mu = t^\mu \log_2 y^\mu \; + \; (1 - t^\mu) \log_2 (1 - y^\mu)$$

from information theory (Pearlmutter & Hinton 1986, Hinton 1987). The most obvious way to reduce the error is to move down the gradient of E with respect to $\vec{W}$. That is,

$$\Delta W_i = -\alpha \sum_\mu \frac{\partial E^\mu}{\partial W_i}$$

If the rule is used "on-line", where the weights are altered in response to each input pattern as it is presented (rather than "batch" mode where the above sum can be calculated explicitly and then the weights altered), the error measure isn't decreased at every step, but tends to do so over many steps, provided α is sufficiently small. This is known as *stochastic gradient descent.*

[Widrow & Hoff 1960] and [Kohonen 1977] considered the case of the LMS error measure for linear units, where the output function is just $y^\mu = \phi^\mu$. This gives the so called "adaline" or "delta" or "Linear LMS" rule,

$$\Delta^\mu W_i = \alpha \, (t^\mu - y^\mu) \, \xi_i^\mu$$

which is the same as Rosenblatt's "non-quantized error-correction procedure" except that in this case it is applied for *every* presented pattern. Notably E has no local minima that are not also global minima, because the contribution E^μ is a quadratic well in weight space, and the sum of any number of such wells is just another quadratic well (Hinton 1987). Although this may be a sensible criterion for training units with linear output functions, such units are not particularly interesting, because the linear mappings they implement can only map similar input patterns onto similar output patterns, and are thus not sufficiently powerful to enable arbitrary classification tasks to be achieved. Furthermore, multiple layers of such units are no more computationally powerful than single layers: each layer performs a linear transformation of the input pattern, and the result of a linear transformation of *this* is equivalent to a single linear transformation of the original pattern.

If the units have a differentiable, nonlinear but monotonic output function then if there exist weights such that $E = 0$, then likewise E has no local minima[1] and "perfect" gradient descent is therefore guaranteed to find this minimum. However if no such weights exist then local minima can arise (Hinton 1987; Sontag 1988). [Brady *et al.* 1988] show that even if the training set is separable, the LMS minimum may not separate the patterns. [Wittner & Denker 1988] give a simple example of this for the linear case, noting that it is equally true for smooth, monotonic non-linear output functions. They further suggest a modification to E so that gradient descent of this measure *does* succeed on a separable training set. [Sontag and Sussmann 1988a] suggest a very similar modification also, but show that even with this criterion, local minima may be generated if the training set is not separable. In a later paper (Sontag & Sussmann 1988b) they present an example of such a local minimum with exclusively binary inputs and outputs.

The smooth nonlinear function most commonly used is the "sigmoid" or "logistic function":

$$y = 1 \;/\; (1 + e^{-\phi})$$

Use of the LMS measure with this output function gives the following "sigmoidal LMS" rule:

$$\Delta^\mu W_i = \alpha \; (t^\mu - y^\mu) y^\mu (1 - y^\mu) \; \xi_i^\mu$$

If the cross-entropy error measure is used instead of LMS, we have the "sigmoidal cross-entropy" rule

$$\Delta^\mu W_i = \alpha \; (t^\mu - y^\mu) \; \xi_i^\mu$$

as in the original delta rule. Notably, a great deal of added generality arises because E can be differentiated with respect to *any* weight in a whole network of interconnected units. This idea in conjunction with the sigmoidal LMS rule forms the basis of the *Generalised delta* or *backpropagation* rule[2] (Werbos 1974; Rummelhart Hinton & Williams 1985).

[1] This is because altering the unit's output function in this way cannot build "walls" in weight space where there were none before.

[2] The generality doesn't arise in the case of linear units because there is no computation possible in multiple layers of such units that cannot be achieved in a single layer.

For perceptron units with threshold output functions, E cannot be differentiated to obtain a rule as it stands. However the differential can be done for a modified E which is "threshold LMS", such as in [Hinton, 1987]:

$$E^\mu = \begin{cases} 0 & \text{if output is correct and } |\phi^\mu| > m \\ (m - \phi^\mu)^2 & \text{if } t^\mu = 1 \text{ but } \phi^\mu < m \\ (m + \phi^\mu)^2 & \text{if } t^\mu = 0 \text{ but } \phi^\mu > -m \end{cases}$$

This predates but satisfies the conditions suggested in [Wittner and Denker 1988] and [Sontag and Sussmann 1988a] as necessary for gradient descent to separate where this is possible. Differentiation of this measure gives essentially the same rule as Rosenblatt's "non-quantized error-correcting rule" except that the learning rate α should be a small constant, to move the weights a small amount in the right direction, rather than $1/\|\xi^\mu\|^2$, the amount required to actually correct the output due to pattern ξ^μ.

Finally, it can be argued that if the units are indeed linear threshold perceptrons and the goal is to misclassify as few patterns as possible, minimising a sum of squared errors in ϕ is of no consequence, since the actual output for pattern ξ^μ bears no relation to how large ϕ^μ is in error. Hence this may in fact be a bad strategy for minimising the total number of perceptron errors.

3.3 The Perceptron Learning Rule (PLR)

On presentation of pattern ξ^μ, the Perceptron Learning Rule (Rosenblatt 1962) alters the weights only when the target t^μ differs from the actual output[3]

$$\Delta W_i^\mu = \alpha\, (t^\mu - o^\mu)\xi_i^\mu \tag{3.2}$$

where α is the overall learning rate and is a constant. Thus a correct response engenders no change, making this a strictly *error correcting rule*. Note that this is equivalent to the associative rule of equation (3.1) except that now the change is only made in the event of an error. This rule constitutes a powerful but restricted procedure for learning to distinguish between classes of input patterns. It is powerful, in that the Perceptron Convergence Theorem (Minsky and Papert 1969) states that *if* a set of weights exists for which the perceptron makes no errors, the PLR will converge on such a set after a finite number of pattern presentations. It is restricted however with respect to learning, in that if such a perfect solution does not exist, the PLR never stabilises the weights.

[3]Notation: although here and elsewhere learning rules are stated with an index indicating the particular (i^{th}) input, the rule is assumed to be implemented for all inputs (i=0..N) convergent on the unit, where the zeroth input is understood to provide the necessary bias.

3.3.1 Which learning rules will converge on separable patterns?

Theorems of perceptron convergence on separable sets of patterns have been proved and improved in several different ways over the years. In the initial perceptron paper (Rosenblatt 1958) there's no hint of such a theorem at all. [Block 1961] gave an early and somewhat complicated version. In [Rosenblatt 1962] it is split into two proofs, one for the classical PLR (Rosenblatt calls this the "quantized error-correcting rule"), and another for what he calls the "non-quantized error-correcting rule". Here, the magnitude of the weight change is set so that the current (incorrect) response is only just corrected. That is, the weights are altered so that the summed input ϕ itself exactly equals the target output, implying that the actual output obtained by thresholding ϕ will now be corrected. If an error is made on presentation of $\vec{\xi}$ the appropriate weight change is $\Delta W_i = \alpha(t-\phi)\xi_i$ with α set at $1/||\vec{\xi}||^2$. [Nilsson 1965] and [Duda & Hart 1973] discuss these convergence results in some detail.

The best known and most elegant proof of convergence of the PLR appears in Minsky and Papert's book "Perceptrons", a *tour de force* of novel analysis into what perceptrons can represent (and to a lesser extent what they can learn). The proof works by assuming there exists a set of weights $\vec{W}^\star$ which would solve the problem and then showing the cosine of the angle between $\vec{W}^\star$ and the actual weights $\vec{W}$ will exceed 1 after a finite number of weight changes. Since the above cannot occur and weight changes are always made in response to errors, the algorithm must converge on a correct set of weights after a finite number of updates. Hence the PLR essentially learns by decreasing (an upper bound on) an angle in weight space. In this section the Perceptron Convergence Theorem is reviewed, closely following the proof given in "Perceptrons" (Minsky and Papert 1969) with some added generality to see which attributes of the PLR are flexible without losing its convergence property. The conditions turn out to be quite flexible.

For convenience we begin by defining $\vec{X}^\mu = \hat{t}^\mu \vec{\xi}^\mu$, so the learning task becomes "find weights $\vec{W}$ such that $\vec{W} \cdot \vec{X}^\mu > 0$ for all patterns μ".

Learning Algorithm:

START: Set $\vec{W} = \vec{0}$; all the weights are zero.

TEST: Choose an input, $\vec{X}^\mu$

if $\vec{W} \cdot \vec{X}^\mu > \epsilon$ go to TEST.

else go to ADD.

ADD : $\vec{W} \rightarrow \vec{W} + f(\phi^\mu)\vec{X}^\mu$

Theorem:
The algorithm given above will go to ADD only a finite number of times if the following conditions hold.

1. There exist weights $\vec{W}^\star$ for which $\vec{W}^\star \cdot \vec{X}^\mu > 0$ for all $\vec{X}^\mu$. In other words the patterns $\vec{\xi}^\mu$ must be linearly separable by some hyperplane in pattern space.
2. Every $\vec{X}$ must be able to be chosen an arbitrarily large number of times.
3. $\epsilon \geq 0$ and is bounded above.
4. Whenever weights are altered $f(\phi^\mu) > \epsilon$ and is bounded above.

Proof:
Assume there exist weights $\vec{W}^\star$ and $\delta > 0$ such that $\vec{W}^\star \cdot \vec{X}^\mu > \delta$ for all patterns μ. Also assume the values of f are bounded such that

$$\epsilon < f_{min} \leq f \leq f_{max}$$

Define G as the square of the cosine[4] of the angle between $\vec{W}^\star$ and the actual weights $\vec{W}$,

$$G(\vec{W}) = \frac{(\vec{W}^\star \cdot \vec{W})^2}{\|\vec{W}^\star\|^2 \, \|\vec{W}\|^2} \leq 1$$

Now examine how this measure changes each time the learning algorithm goes to ADD. For convenience another index is introduced. This index, τ, starts at zero and is incremented after each ADD.

<u>Numerator:</u>

$$\begin{aligned}
\vec{W}^\star \cdot \vec{W}_{\tau+1} &= \vec{W}^\star \cdot (\vec{W}_\tau + f_\tau \vec{X}_\tau) \\
&= \vec{W}^\star \cdot \vec{W}_\tau \; + \; f_\tau \vec{W}^\star \cdot \vec{X}_\tau \\
&\geq \vec{W}^\star \cdot \vec{W}_\tau \; + \; f_\tau \delta
\end{aligned}$$

[4]Minsky and Papert use the cosine itself, showing that it must eventually increase above 1. However, the square is used here because it simplifies presentation.

because of the assumption that $\vec{W}^{\star} \cdot \vec{X}_{\tau} > \delta$. Therefore, after n ADD's we have

$$\begin{aligned}
\vec{W}^{\star} \cdot \vec{W}_{n} &\geq \delta \sum_{\tau=0}^{n-1} f_{\tau} \\
\Rightarrow \left(\vec{W}^{\star} \cdot \vec{W}_{n}\right)^{2} &\geq \delta^{2} \left(\sum_{\tau=0}^{n-1} f_{\tau}\right)^{2} \quad \text{(since } \delta > 0\text{)} \\
&\geq \delta^{2} n^{2} f_{min}^{2} \qquad \text{(since } f \geq f_{min} > 0\text{)}
\end{aligned}$$

Denominator:

Note first that going to ADD means we must have encountered a pattern for which $\vec{W} \cdot \vec{X}_{\tau} < \epsilon$.

$$\begin{aligned}
\|\vec{W}_{\tau+1}\|^{2} &= \vec{W}_{\tau+1} \cdot \vec{W}_{\tau+1} \\
&= (\vec{W}_{\tau} + f_{\tau}\vec{X}_{\tau}) \cdot (\vec{W}_{\tau} + f_{\tau}\vec{X}_{\tau}) \\
&= \vec{W}_{\tau} \cdot \vec{W}_{\tau} \; + \; 2 f_{\tau} \vec{W}_{\tau} \cdot \vec{X}_{\tau} \; + \; f_{\tau}^{2} \vec{X}_{\tau} \cdot \vec{X}_{\tau} \\
&\leq \|\vec{W}_{\tau}\|^{2} \; + \; f_{\tau}^{2} \|\vec{X}_{\tau}\|^{2} \; + \; 2 f_{\tau} \epsilon
\end{aligned}$$

Therefore, after n ADD's we have

$$\begin{aligned}
\|\vec{W}_{n}\|^{2} &\leq N \sum_{\tau=0}^{n-1} f_{\tau}^{2} \; + \; 2\epsilon \sum_{\tau=0}^{n-1} f_{\tau} \\
&\leq N n f_{max}^{2} \; + \; 2 \epsilon n f_{max} \quad \text{(since } 0 < f \leq f_{max}\text{)}
\end{aligned}$$

Since both numerator and denominator are positive, we have a lower bound for G after n ADD's of

$$G_{n} \geq \left(\frac{\delta^{2} f_{min}^{2}}{\|W^{\star}\|^{2} (N f_{max} + 2\epsilon) f_{max}}\right) n$$

The expression in the brackets is constant, so there is an n for which G_n exceeds 1. However we know this cannot occur. Therefore, the algorithm must go to ADD only a finite number of times. Thus convergence is guaranteed under the conditions given in the theorem. *QED.*

Note that ϵ need not be a constant, but must be non-negative[5] and bounded above, meaning there must be some value of $\vec{X}^{\mu} \cdot \vec{W}$ above which no change

[5] From the learning algorithm, this is required in order that the weights change at all from their initial values of zero.

occurs. Also note that the conditions given above are sufficient only. In particular the condition that f be bounded above may be unnecessary, since for example Rosenblatt's "non-quantized error-correcting rule" is guaranteed to succeed on separable patterns yet f is a linear function of ϕ. The classical PLR corresponds to $f_{min} = f_{max} = 1$ with $\epsilon = 0$.

In fact convergence of the the "threshold LMS" algorithm can be deduced from the Perceptron Convergence theorem. To guarantee success on a separable problem it is sufficient that the algorithm be *able* to present each pattern an arbitrary number of times: there is no restriction on the order of presentation. Therefore every time an error is made the PLR step can be implemented repeatedly some arbitrary, but finite, number of times before another pattern is chosen, which (for sufficiently small α) is trivially the same as the threshold learning rules above, and the perceptron convergence result still holds. In a sense this gives "stronger" convergence, because proofs of convergence using the Perceptron Convergence Theorem do not fall prey to the following problem associated with any gradient descent, that the finite step size α makes any real algorithm only an approximation to gradient descent. If α is too large the minimum is no longer guaranteed to be found, while on the other hand making α too small slows convergence down.

The perceptron with PLR never stabilises for pattern sets which aren't linearly separable because it alters the weights every time an error is made. If the patterns are separable we know from the Perceptron Convergence theorem that a lower bound on the agreement between the actual weights and a set of desired weights is increased, but perceptron behaviour in the non-separable case is not well understood. Minsky and Papert (among others) proved a theorem which they called the Perceptron Cycling Theorem, stating that a perceptron with linearly separable patterns will never visit the same weights vector twice, whereas there is no limit to the number of times that this will occur for the non-separable case.[6]

3.4 A "thermal" perceptron learning rule.

The PLR on its own does not work for non-separable patterns: the weights are neither stable nor "good on average". One rationale goes as follows: the trouble is that the PLR does the same thing for *every* error made. Instead, the benefit from improving ϕ^ν should be tempered by the possibility that the new weights

[6] In principle this can be used as a positive test for non-separability of the patterns, but in practice requires enormous amounts of memory.

now misclassify patterns they previously got right[7]. Firstly since (given an error has been made) the change in ϕ is independent of the value itself, an error with a large associated ϕ is less likely to be corrected in this step than an error where ϕ is small. The weight changes necessary to correct a large error are themselves large, and hence much more likely to corrupt the existing correct responses of the unit. We can say that errors due to small $|\phi|$ are more likely to be cured without altering the response to other patterns than those due to large $|\phi|$, and the weight changes made should be biased accordingly towards errors where $|\phi|$ is small.

A simple way to do this is to make the PLR weight changes tail off exponentially for large $|\phi|$:

$$\Delta W_i = \alpha \left(t^\nu - o^\nu\right) e^{-|\phi|/T} \, \xi_i^\nu \tag{3.3}$$

The "temperature" T controls how strongly the changes are attenuated for large $|\phi|$. At high T the PLR is recovered, since the exponential becomes almost unity for any input. Also, at any given T the conditions for convergence given in section 3.3.1 are satisfied.

One picture of the way this rule works is given by considering the patterns as points in a space and the weights as defining a decision surface which is a hyperplane in this space. This hyperplane moves every time an error is made. In the usual PLR, it moves by approximately the same amount whenever there is an error, whereas in the thermal PLR (henceforth just called "Thermal") it moves by an amount which is large if the pattern causing the error is close to the hyperplane, and small if the pattern is distant. As an approximation, one can imagine a zone immediately to either side of the hyperplane, within which an error will cause movement of the hyperplane. The perceptron will be relatively stable if there are no errors occurring in this zone. A natural extension to this is to anneal the thermal effect by gradually reducing the temperature from high T where the usual PLR behaviour is seen to $T = 0$, where there are no more weights changes. The annealing of the temperature is then the gradual reduction of the extent of the "sensitive" zone. In the limit of $T \to 0$ the zone disappears altogether and the perceptron is stable. This gradual freezing is particularly desirable because it stabilises the weights in a natural way over a finite learning period. One possible problem which could arise is that patterns for which $|\phi|$ is very small continue to cause comparatively large weight changes even at low temperatures, so the effect of gradually reducing the learning rate α was also investigated.

[7] Of course this check cannot be made exact without going through all the patterns and counting the numbers of errors. This is precisely the function of the "ratchet" in the Pocket algorithm. In that case the cost of doing it *all* the time is avoided by only checking if the run length exceeds the best so far. We would like to avoid this, and see what can be done within the simple perceptron framework itself, rather than inventing flourishes to add onto it.

The effects of annealing one or both of the temperature and the rate are plotted in figure 3.1. In this case the problem used is highly non-separable: all 1024 of the binary patterns across 10 inputs are used, and each is assigned a target 0 or 1 with equal probability. Exactly half the patterns are target 1, so a unit with no weights at all gets 50% of the patterns correct by default. This is also the average level attained by the threshold LMS algorithm. In the annealed cases the relevant quantity was reduced from its starting value down to zero linearly over the entire training time. This time was 1000 epochs; that is, each pattern was presented an average of 1000 times although the actual order of presentation was random. It should be remembered that the Pocket algorithm with ratchet involves much heavier computation per epoch (of the order of ten times longer in real time for this problem) than either the Pocket algorithm alone or Thermal. The latter two run at approximately the same speed.

First consider the simplest case where there is no annealing: both the temperature and rate are held fixed at their starting values. Clearly including the exponential weighting has greatly improved the perceptron's performance. The Thermal rule can perform much better than the Pocket algorithm (which has double the number of parameters and a "longest run" checker), and considerably better than even the Pocket algorithm with ratchet method (which has still more parameters and an explicit check of the entire training set at each update of the pocketed weights).

Since the weights are not stable (they still change every time there is an error and since the problem is non-separable this is always occurring), the observed improvement implies that good sets of weights are being visited preferentially, or in other words the perceptron using the Thermal rule spends far more time in the good regions of weight space than the pure PLR. Also, it can potentially locate good regions of weight space more quickly due to the use of real-valued weight changes. Although integer weights as used in the Pocket algorithm can approximate a real-valued weights vector arbitrarily closely (because the magnitude of a weights vector is irrelevant for threshold units) and require less storage on a computer, they may take much longer to build up from initial weights of zero strength. However there is a strong temperature dependence: in this case the performance peaks at around $T = 1.0$ and, as expected, reduces to that of the standard PLR as T grows large.

Annealing the rate alone gives good results provided the temperature is just right, but is very sensitive to this parameter. Apart from a small improvement for low temperature, the score in this case is considerably worse than the "no annealing" case at the same temperature, so annealing the rate is actually detrimental.

However the performance seen by annealing the temperature alone not only approaches that of the "no annealing" case for low temperatures, but only drops

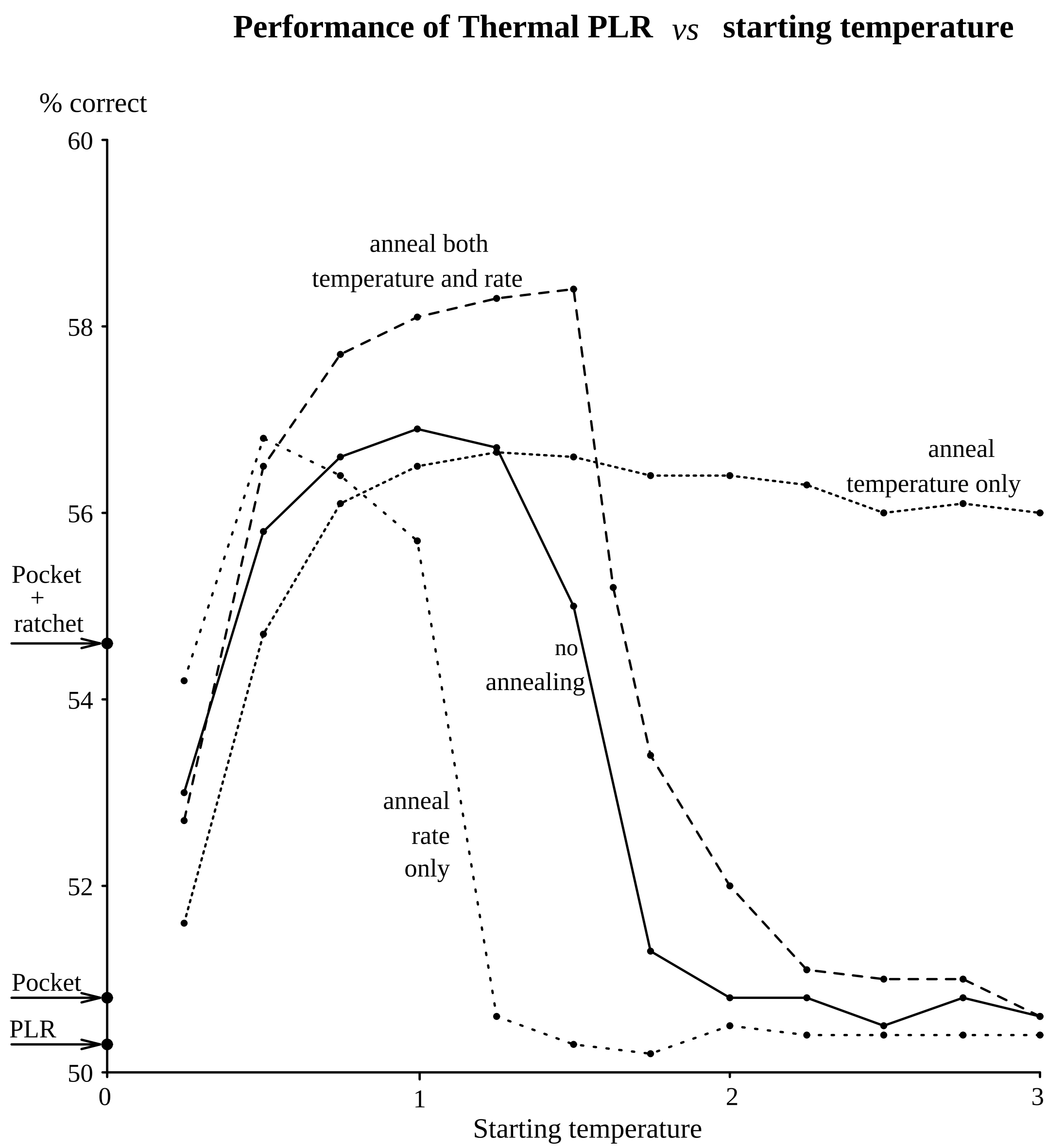

Figure 3.1: Performance of various algorithms on a highly non-separable problem produced by assigning targets at random to binary patterns. The abscissa denotes the starting temperature, T_0, while the starting value for α is always 1. The ordinate denotes the percentage of patterns being classified correctly at the end of 1000 epochs of training. Each plotted point is the average of 100 independent trials. Also shown are the levels attained by the standard PLR, the Pocket algorithm and the Pocket algorithm with ratchet after 1000 epochs.

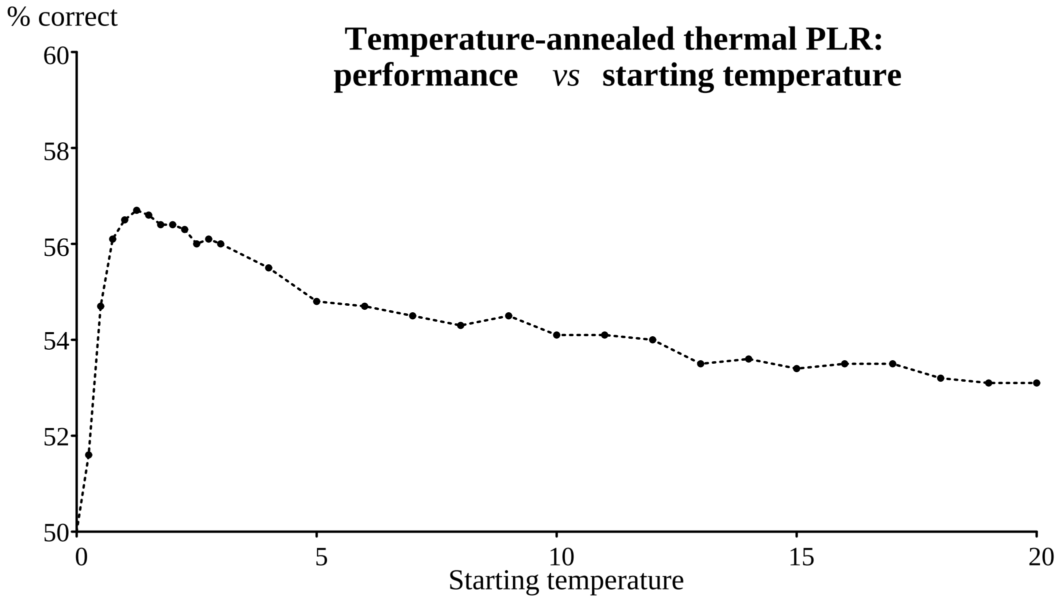

Figure 3.2: The curve shows the performance of the temperature-annealed Thermal rule over a wider range of starting temperatures.

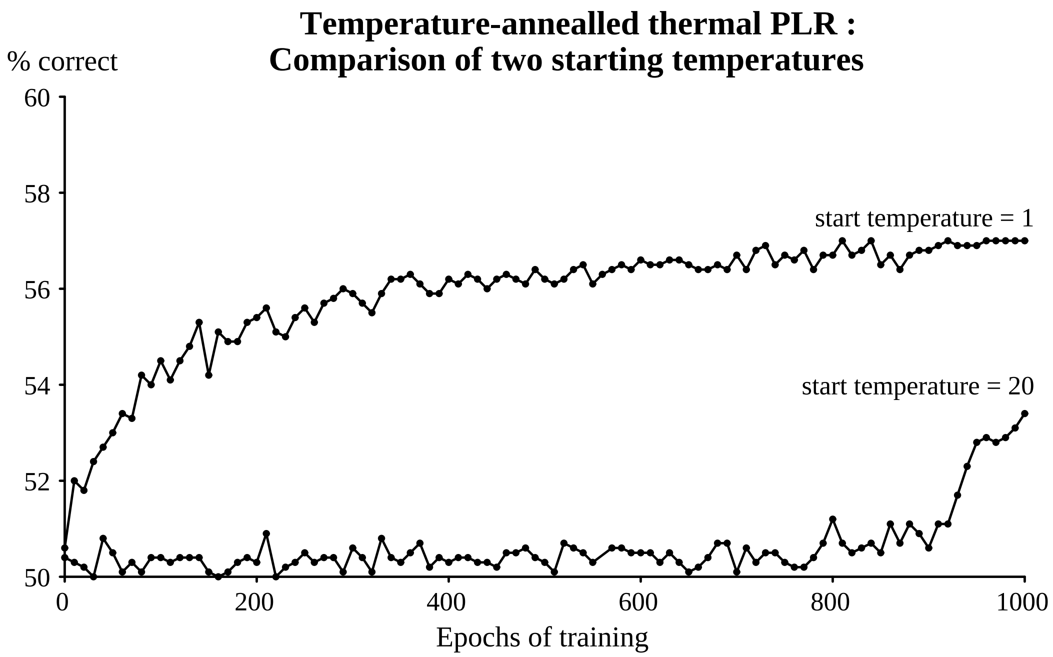

Figure 3.3: The curve shows the performance of the temperature-annealed Thermal rule over 1000 epochs for two different starting temperatures. Each point is an average over 100 trials.

away gradually above this region. Figure 3.2 confirms this effect over a wider range of T_0. Ultimately this curve must return to the level of the PLR, since annealing from higher and higher temperatures means spending more time effectively implementing the PLR, and less time exploiting the exponential weighting in the useful region (in this case $T < 2.0$, from the "no annealing" curve). The fact that the curve drops so slowly therefore suggests that the benefits of annealing the temperature arise from the low temperature region, and that the time spent in this region can be quite brief. Figure 3.3 shows the time course of learning by annealing the temperature alone for two different starting temperatures, $T_0 = 1$ and $T_0 = 20$. Indeed, the improvement in the weights where $T_0 = 20$ only occurs after 900 epochs, which is as temperature goes through the region $T < 2$. This relative independence from T_0 is an important characteristic, because the optimal value to use for T_0 may vary from problem to problem (although this is generally close to unity for training sets of all binary patterns with $N \leq 10$) for the other cases.

Annealing the temperature *and* the rate however gives the highest scores of all. In this case, compared to the "break even" point of 50%, performance is a full ten times better than the Pocket algorithm, and almost double that where a ratchet is included. The temperature dependence roughly follows that of the no-annealing curve, and in this case both curves peak at temperatures close to 1.

The time course of the learning is shown for $T = 1.0$ in figure 3.4. Although the Pocket algorithm (with ratchet) reaches a moderate set of weights quickly, it fails to continue the improvement, whereas the perceptron using the Thermal rule goes on increasing considerably further. The main advantage of annealing lies in being able to "force" a good solution quickly. For example, annealing over 200 epochs reaches the same level as not annealing over 1000 epochs, as shown in the figure.

Finally, the performance *vs* training time on a linearly separable training set is shown in figure 3.5, again with $T_0 = 1$. If there is no annealing, the scores for Thermal and the usual PLR (or equivalently, Thermal at high temperature) are similar. The low temperature case scores slightly lower, although the convergence time is almost the same. However a striking speed-up is possible by annealing the rate and temperature together: in this case the solution can be reliably found in approximately a tenth of the usual training time.

It is interesting to compare the rationale behind Thermal with that of the LMS procedures. In the former, it is argued that large errors (in ϕ) should be penalised lightly, since endeavouring to correct these errors means corrupting existing weights to a large degree. In the latter, large errors are supposed to be more heavily penalised than small ones[8], since these large errors contribute proportion-

[8] It is often said to be a "drawback" of the sigmoidal LMS compared to the cross-entropy rule

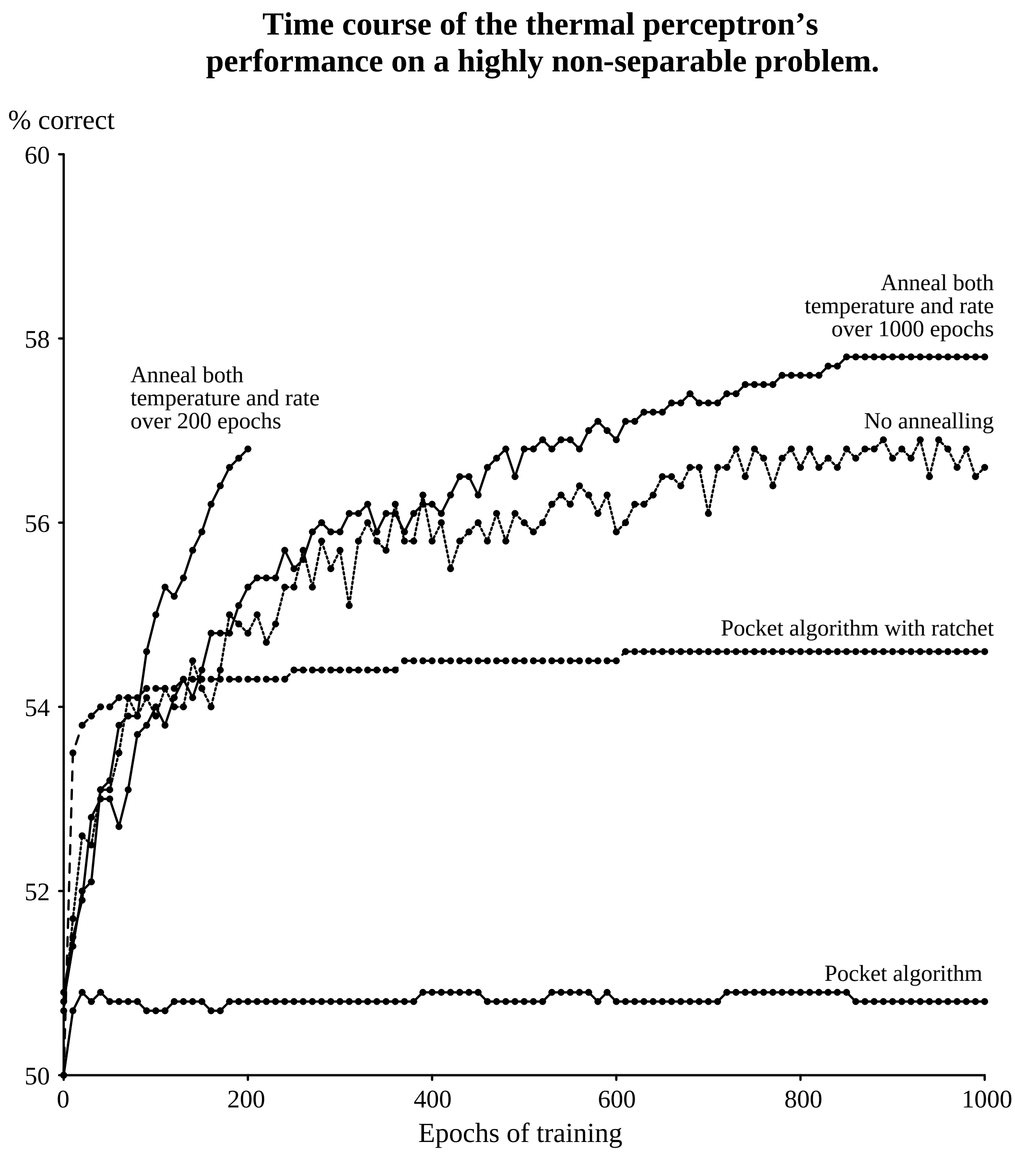

Figure 3.4: Performance on a highly non-separable problem plotted *vs* the training time for the $T_0 = 1$ case over 1000 epochs. Each point is an average over 25 trials.

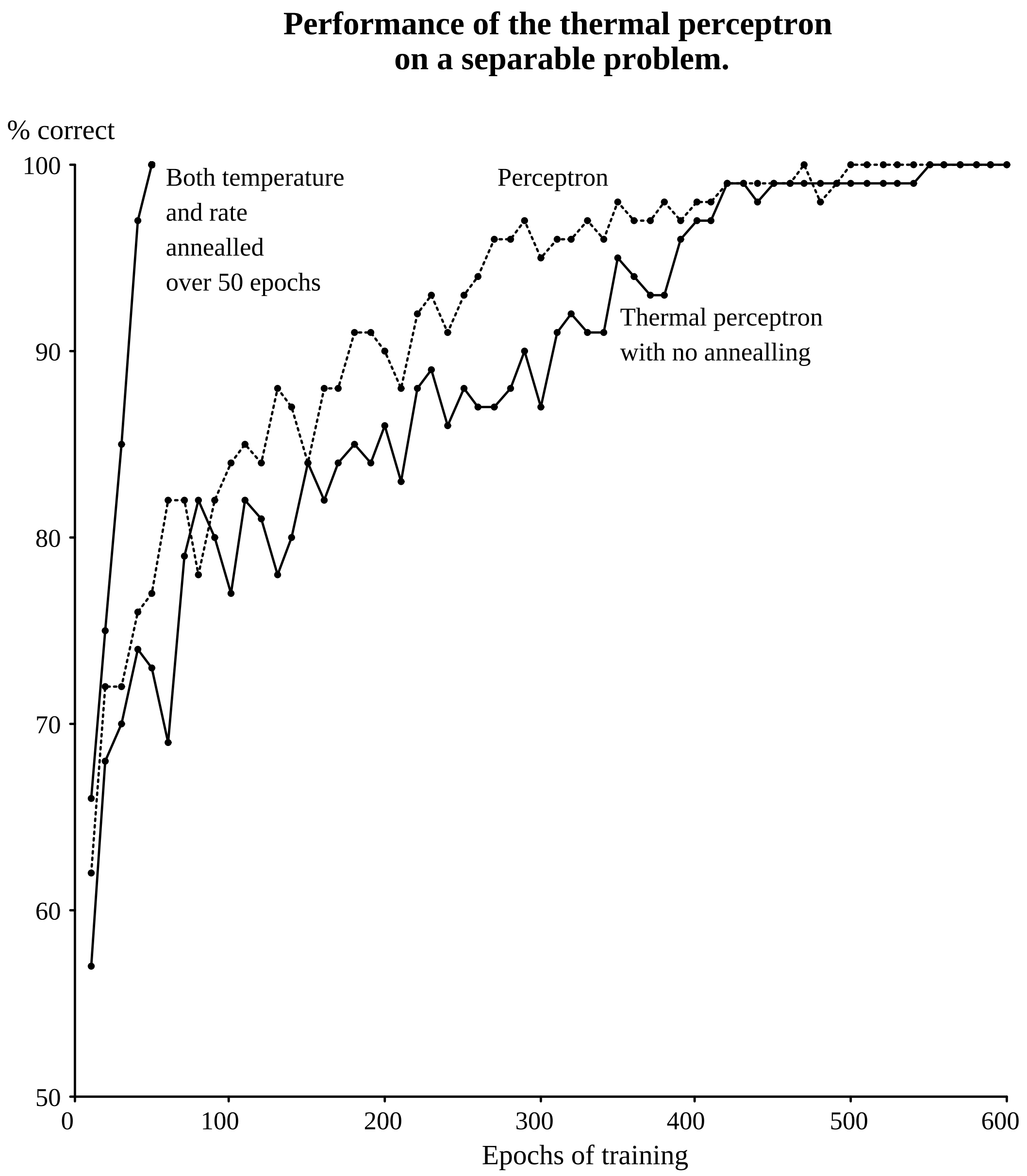

Figure 3.5: Performance on a linearly separable problem, plotted *vs* the training time for the $T_0 = 1$ case over 1000 epochs. Each point is an average over 25 trials.

ally more to the quantity being minimised (the sum of squared errors in ϕ). Hence these two approaches have opposite motivations.

3.5 Expressing learning rules as curves.

In the remainder of this chapter, other learning rules for the perceptron architecture are considered. One approach would be to derive rules which minimise some measure of the total error, as was outlined in section 3.2. Another is to define learning rules in terms of a small number of parameters, and then investigate the merits of particular combinations of parameter values. This is the approach adopted here.

How might the possible learning rules for the Perceptron architecture be parameterised? A learning rule for a perceptron is simply a single-valued function $\Delta^\mu W_i$, the change to be made to weight W_i upon presentation of the μ^{th} input pattern. This could depend on

- W_i, the existing weight value.
- ϕ^μ, the sum of weighted inputs.
- ξ_i^μ, the input activity.
- t^μ, the target output.

and also an overall factor specifying the learning rate.

In most accounts of synaptic error-correcting learning rules, the output activity of the post-synaptic unit is present explicitly. In general this is some monotonically increasing function of ϕ^μ alone. However for perceptrons an error is understood as being made whenever the *sign* of ϕ^μ is positive [negative] where the target is OFF [ON], so it is superfluous to include the intermediary step of evaluating the actual output since this is itself a single valued function of ϕ^μ. The following are true of the supervised learning rules commonly in use:

1. the existing weight value is *not* taken into account. That the weight's magnitude should not matter in learning is no surprise, since the unit's output

that really large errors do *not* engender large weight changes (eg. Hinton 1987).

is only a function of a linear sum of weights. If bounds on weights are to be taken seriously, the existing weight may be included as a way of preventing their size growing beyond some limit. However for most purposes it can be ignored.

2. the weight change contains a multiplication by ξ_i^μ. Therefore changes are only made to the i^{th} weight if ξ_i is active.

3. errors of both types (wrongly ON and wrongly OFF) are treated symmetrically. The only difference is in the sign of the weight change incurred: weights should increase [decrease] if ϕ is too low [high]. That is,

$$\begin{aligned}
\Delta^\mu \vec{W} &= \begin{cases} f(\phi^\mu)\vec{\xi}^\mu & \text{if } t^\mu = 1 \\ -f(-\phi^\mu)\vec{\xi}^\mu & \text{if } t^\mu = 0 \end{cases} \\
\text{or equivalently,} \quad \Delta^\mu \vec{W} &= \hat{t}^\mu \; f(\hat{t}^\mu \phi^\mu) \; \vec{\xi}^\mu
\end{aligned} \tag{3.4}$$

The advantage of these restrictions is that they cut down the number of free parameters controlling learning, so it is possible to ignore many of the possible functions of input, output and target, and concentrate on those that conform to the general features of known rules such as PLR, sigmoidal LMS and Cross-entropy. In other words not much generality is lost if the weight change is treated as a single function of ϕ alone together with equation (3.4). This function, f, is the weight change the perceptron makes if the target $t^\mu = 1$ and the input ξ_i is active.

Figure 3.6 shows this function for the Perceptron learning rule, Thermal perceptron rule, sigmoidal LMS, threshold LMS and sigmoidal cross-entropy rules for comparison.

Which curves correspond to rules which are guaranteed to converge on separable data? The convergence conditions can be stated graphically, since the function f introduced above is the same as that used in section 3.3.1. Weights must only change if $\vec{X} \cdot \vec{W} \leq \epsilon$, so $f(\phi)$ must go to zero above some positive value of ϕ, and below this value $f(\phi)$ should be greater than zero and bounded above. Note that this implies a discontinuity in the weight change function at ϵ. From figure 3.6, clearly the PLR and Thermal meet these conditions.

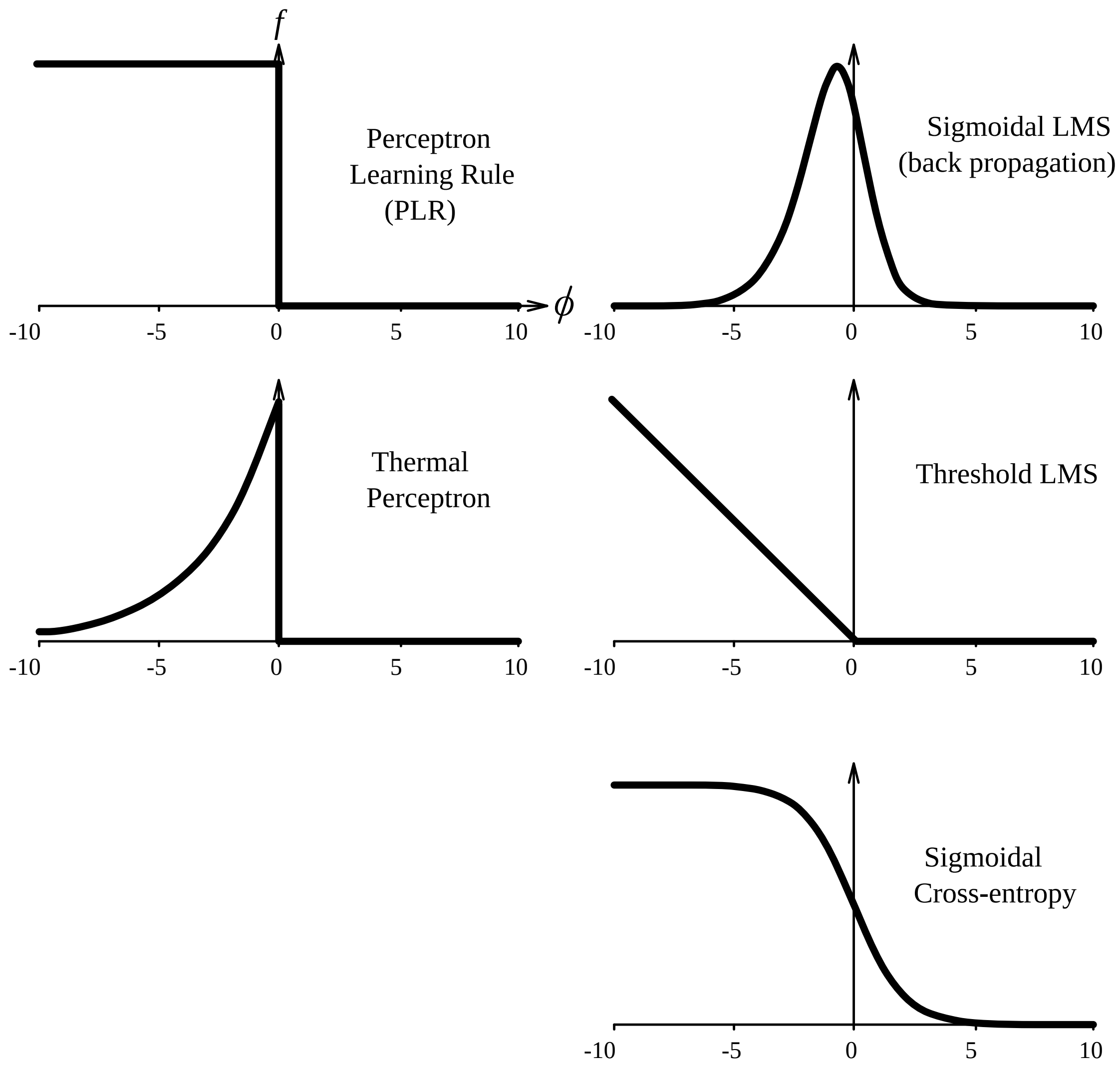

Figure 3.6: The plotted function f is the change to be made where the target is 1 and the input is active, shown for several learning rules. The vertical scale has been normalised.

3.6 Learning rules considered as points in space.

In this section the learning function f is given a specific form controlled by just four parameters. The possible learning rules may then be viewed as points in a four-dimensional parameter space. Whereas associative rules can be parametrised and investigated analytically (Willshaw & Dayan 1990), for error-correcting rules this is much more difficult. Instead, an essentially empirical approach is adopted here, where rules are evaluated by testing them on real problems. There are many possible rules and the ability of each to optimise a very simple criterion (the number of errors made on the training set) is simply measured. We may expect to see some structure, in that nearby points in this space of rules should have similar learning properties. The discussion which follows presents the general features of this space, rather than attempting an exhaustive evaluation.

3.6.1 Restrictions on the form of the curves.

Attempting to evaluate the whole range of possible curve shapes is out of the question. However there is no need to do this, since the curves we know of already are really of a simple form: they are all single-valued, with a single maximum. In fact they can all be closely approximated as "hump-shaped" curves, where "hump-shaped" is taken to mean $f(\phi)$ is positive definite and differentiable with a single maximum, tending to zero in the limit of large $|\phi|$. The PLR and Thermal perceptron rules are not differentiable but the curves can be approximated by a smooth function, to an arbitrary degree if necessary. The PLR and Cross-entropy curves do not return to zero as ϕ tends to $-\infty$, but again this can be approximated arbitrarily closely if need be[9]. Further, it seems evident that only a very simple function is expected to perform well, given the inherent variability possible in even well defined learning tasks.

Given the above, we can immediately define a parameter for such curves, namely the value of ϕ at which $f(\phi)$ is maximum, hereafter referred to as the OFFSET. The actual value of this maximum is another parameter, referred to as MAX. A way of varying the curves to either side of OFFSET is also required. The simplest form is definable by two parameters, controlling the rate of decrease at either side of $\phi =$ OFFSET. The particular function chosen here is the gaussian since it is simple, and flat at $x = 0$ and $\pm\infty$. Following the analogy of temperature used

[9] In support of this, the Perceptron Cycling Theorem tells us that ϕ is bounded in the non-separable case (because the weights themselves are, since the number of possible configurations is finite), so a curve may always be found which is virtually flat for negative ϕ above this bound.

earlier (although now there's nothing so appealing as the Boltzmann distribution) these are named T_{left} and T_{right}. The family of functions considered are given by:

$$f(\phi) = \begin{cases} \text{MAX} \times \exp[-\beta(\phi - \text{OFFSET})^2 / T_{left}^2] & \text{if } \phi \leq \text{OFFSET} \\ \text{MAX} \times \exp[-\beta(\phi - \text{OFFSET})^2 / T_{right}^2] & \text{if } \phi > \text{OFFSET} \end{cases}$$

If β is set to $\log_e 2$, then the "temperatures" are just the half-height half-widths of the gaussians.

The four parameters OFFSET, MAX, T_{left} and T_{right} correspond to a learning rule. These parameters are shown in figure 3.7. For instance, the learning rules depicted in figure 3.6, with the exception of threshold LMS, can be approximated by the following parameter given below.[10]

Parameter values approximating known rules				
LEARNING RULE	OFFSET	MAX	T_{left}	T_{right}
Perceptron (PLR)	0	1	∞	0
Thermal	0	1	variable	0
sigmoidal LMS	$-\log_e 2 = -0.693$	4/27	2.0	1.3
sigmoidal cross-entropy	-3.5	1.45	∞	3.5

3.6.2 Evaluating the curves.

Having defined a "learning rule" by setting the four parameters, the rule's performance may be evaluated on a given learning problem. Learning problems are either (linearly) "separable" or "non-separable" (that is, patterns which are not separable by a single hyperplane). The PLR's behaviour undergoes a sudden change between these two regimes: convergence of the weights is guaranteed in the separable case, cycling is guaranteed otherwise. For separable problems, learning rules similar in character to the PLR are expected to perform well. These correspond to curves where the ratio T_{right}: T_{left} is small. For non-separable problems one intuitively expects "smoother" curves to outperform those with sudden changes near $\phi = 0$, on account of a capacity to "balance out" effects in the way that the sigmoidal LMS rule does in contrast to the PLR, for example.

For each learning problem, a given learning rule has a performance characterised by a number of measures, most directly the ratio of correctly classified patterns

[10] Note however that the sigmoidal rules are supposed to implement stochastic gradient descent and therefore generally have MAX at a much smaller value than that given here, which is that obtained by differentiating the global error without any multiplying factor.

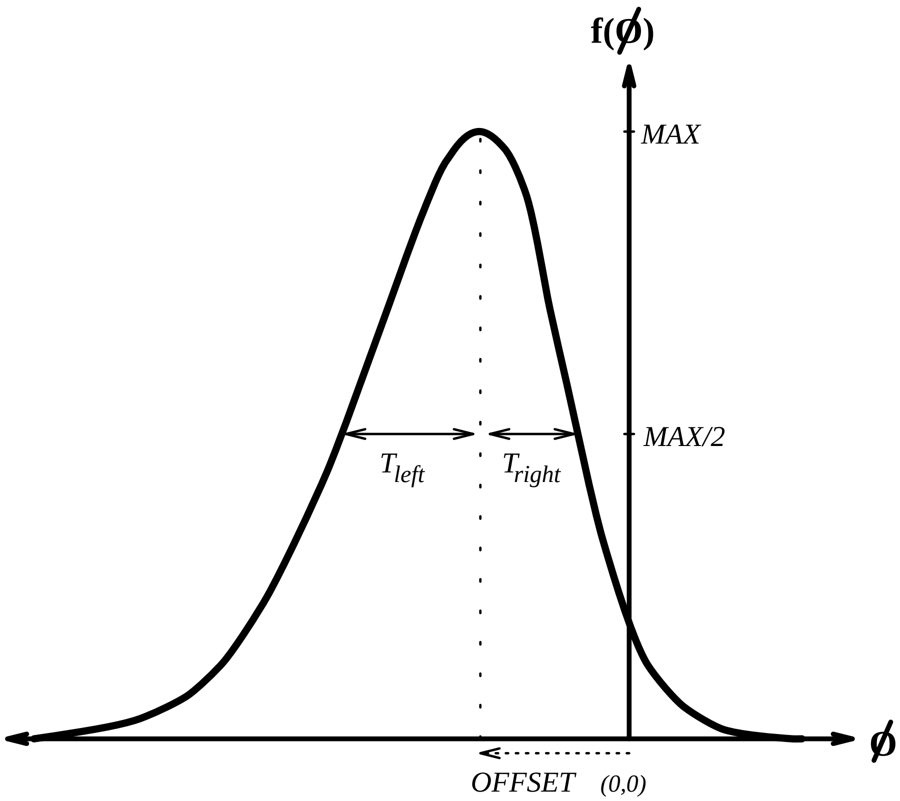

Figure 3.7: Parameters specifying a family of learning rules as "hump-shaped" curves. The curve is the change to be made where the target is 1 and the input is active.

to the total number of patterns in a given problem. In this study the training sets consist of all 2^N possible patterns across the N inputs, and the $N = 8$ case is studied. The performance as the number of iterations is increased is not examined in detail except for separable patterns; rather, the weights are evaluated after an average 100 presentations of each pattern. It should be noted that the "good" rules may well be of a different character if the evaluation is performed after (for instance) a single presentation of each pattern, and there could be interesting effects that occur at lower densities of patterns than those studied here.

The procedure is as follows. A set of test patterns and associated targets is generated. If there are P patterns in the training set then P pattern presentations constitutes one *epoch*. There is some evidence that neural nets learn faster if an epoch instead consists of exactly one presentation of each pattern, that is, the patterns are chosen at random but without replacement. This seems somewhat artificial (*ie.* nothing in the real world is so obliging) so here the patterns are chosen at random, with replacement. At each presentation of a pattern the summed input ϕ is calculated and the weights updated according to the curve and equation (3.4).

The pattern sets used here are attributed either linearly separable or random targets. A useful way to see these two extremes in the same framework is the following. One method of generating sets of patterns is to construct a feed-forward network of perceptrons with one hidden layer of say H units and a single output unit. The weights from inputs to hidden units are set at random values, and each hidden unit's bias is set so that some non-trivial number of input patterns will turn the unit ON. The output unit simply calculates the 'majority function' of the hidden units' activities: it is ON only in the case where at least half the hidden units are ON. The output response of this network to a given input pattern is then taken to be the target for that pattern. If $H = 1$ a linearly separable training set is produced since the network is effectively just a perceptron. If $H = 2$ the target ON class is composed of the union of two half-spaces in pattern space. Also, whatever the mapping produced from inputs to output by this network is, H is the upper bound on the number of hidden units required to learn the mapping using a strictly feed-forward architecture where there are no weights between non-adjacent layers[11]. In the limit of large H the mapping becomes effectively random (but still consistent), as if each input pattern were assigned its target 0 or 1 entirely at random.

The original idea was to use the methods of Genetic Algorithms to "evolve" good curves in a population where survival depends on the ability to learn a training

[11] If this is relaxed then the network is computationally more powerful and fewer units may be necessary. For example a single hidden unit becomes sufficient in order to solve XOR, whereas two are required in the restricted geometry.

set. In this analogy the four parameters (on which the usual genetic algorithm operators act) constitute the "genotype", a perceptron endowed with the curve they represent is the "phenotype", and the world in which they thrive or otherwise is one of inputs and targets, with a perceptron's "fitness" being its performance on the learning task. However the much more straightforward approach of directly searching the parameter space is adopted here because the computation is tractable and the information obtained covers the whole space rather than being dependent on the particular properties a genetic algorithm might or might not have.

Each plot shows T_{left} versus T_{right}, for constant OFFSET and MAX. Hence each grid point represents a particular learning rule. Note that points on the diagonal $T_{right} = T_{left}$ with OFFSET $= 0$ are symmetric curves. The size of the circle at a grid point always represents the relative degree of success at the task. Circles are discretised to one of five sizes (or zero). Every data point is an average taken over twenty-five trials on the same task.

3.6.3 Performance on non-separable training sets.

The most obvious way to assess the curve's performance after some number of learning epochs is to count the number of correctly classified patterns in the training set, relative to the total number of patterns in that set:

$$\frac{\#\text{ patterns correct}}{\text{total }\#\text{ patterns}}$$

A plot using this measure is shown in Figure 3.8. The training set consists of all 256 patterns, with their targets assigned 0 or 1 at random with 50% probability. Exactly half the patterns are target 1. In this plot OFFSET $= 0$ and MAX $= 1$.

Firstly, rules where T_{right} is appreciably greater than T_{left} do not perform well at all, but those where $T_{right} = T_{left}$ are very successful. The most successful of all rules lie on this line, and virtually any point above the line has some success at this task.

Secondly, the curves do not appear to be getting much worse along this line. It turns out that even the rule corresponding to (∞, ∞) performs reasonably well. This corresponds to the "curve" becoming a horizontal line at $f = 1$. This is surprising, because this rule is now just

$$\Delta^{\mu} W_i = \hat{t}^{\mu}\ \xi_i^{\mu}$$

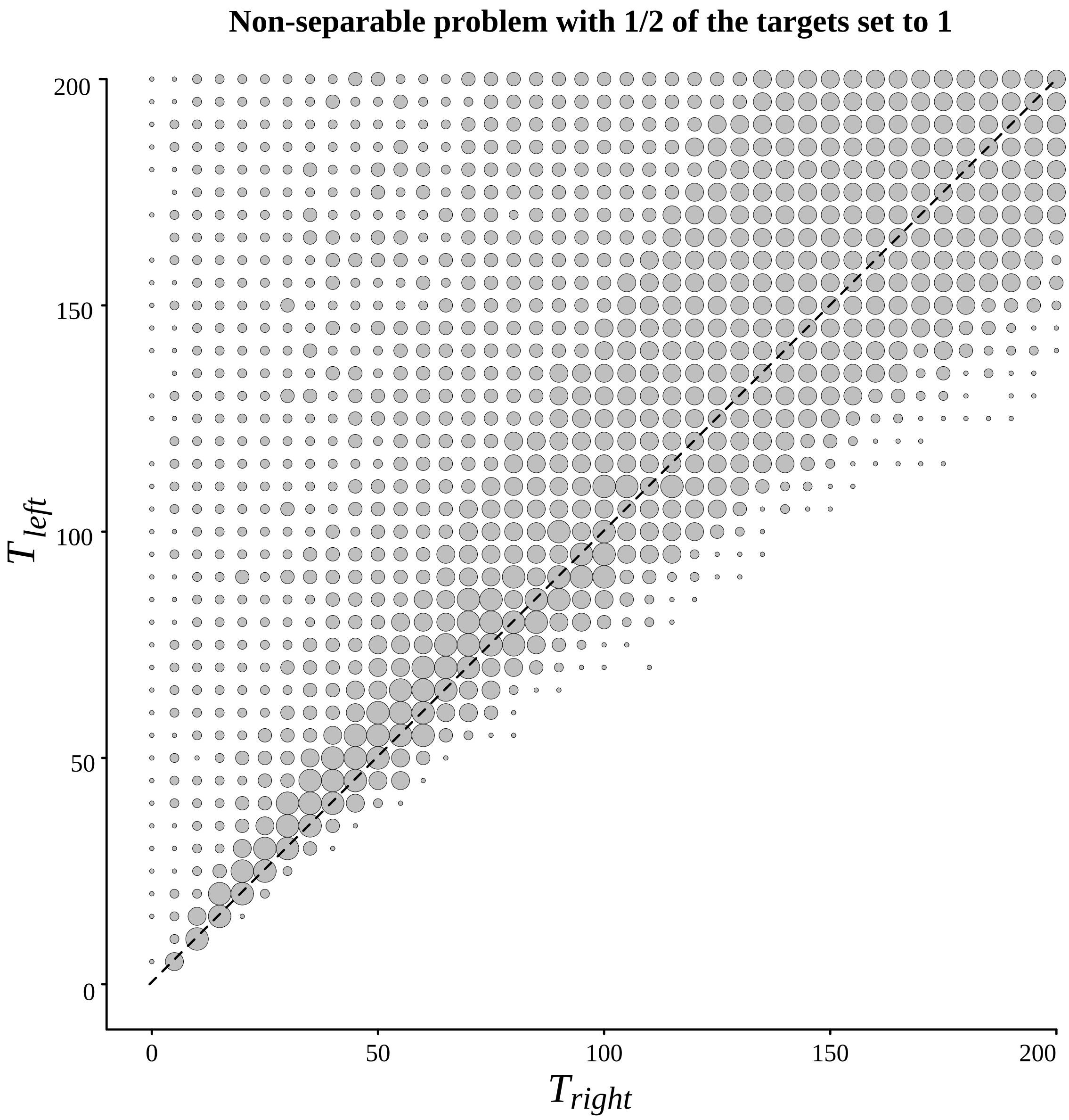

Figure 3.8: Performance on the random target problem using 256 patterns across 8 inputs with 50% of the targets set to 1. The range denoted by circles is from 50% to 64% of the training set correct after 100 epochs of training. OFFSET = 0, MAX = 1.

which is independent of the value of ϕ. This is exactly the associative rule, where the association being formed is between the input vector and the target output. The relative success of such a simple rule strongly suggests that it is somehow "cheating": perhaps it is the high degree of symmetry in this problem which makes this possible. Figures 3.9 to 3.12 show the same type of plot for the same problem except that only a $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$ and $\frac{1}{32}$ respectively of the input patterns are targeted to 1, with the remaining targets set to 0.

Two differences from the 50:50 case are evident. Firstly, although the most successful rules lie on a line, this line is no longer that of the symmetric curves as it was in the 50:50 case. The presence of the line tells us that it is the ratio of T_{left} to T_{right} which dominates performance over a fairly large range in their absolute magnitudes. Its slope confirms that T_{left} should be larger than T_{right}. Why should symmetric curves be best for the 50:50 case, and left-skewed ones be best otherwise? The answer to this lies in the fact that these rules treat patterns of either target the same way, yet there are now more patterns of one target than the other. Consider what happens for a symmetric rule ($T_{left} = T_{right}$) in the "skewed" case. Initially all the weights are zero. The weights are decreased when t^μ is 0 and increased when t^μ is 1. Since there are many more $t = 0$ patterns being presented to the network, the average values of the weights are decreased, and hence the average value of ϕ decreases. The downward trend in the weights will continue, unless a sufficient number of the $t = 1$ patterns elicit high ϕ (that is, close to zero); however since this is a difficult problem, the patterns of both targets remain more or less evenly distributed across the range of ϕ elicited. The result is that the values of ϕ for all patterns continue to decrease, with no pattern having $\phi > 0$. Therefore all rules to the right of the line give "default" scores by responding OFF to every pattern.

Secondly there is a definite fading as this line moves too far from the origin. This is comforting, since the implication otherwise is that the limit situation of a virtually flat curve at $f = 1$ is a successful rule even though it is essentially only associative. Although the ratio T_{left}/T_{right} dominates performance up to a point, eventually the curves get too wide and begin to degrade. This effect is more and more pronounced as the density of target 1 patterns is decreased.

3.6.4 Performance on linearly separable training sets.

The usual score measurement is shown in figure 3.13, after training on a separable problem for 100 epochs. The training set was produced as described in section 3.6.2 using $H = 1$, and consists of the complete set of binary patterns (*ie.* all $2^N = 256$) for $N = 8$. Half the patterns have target 1, so the range of score

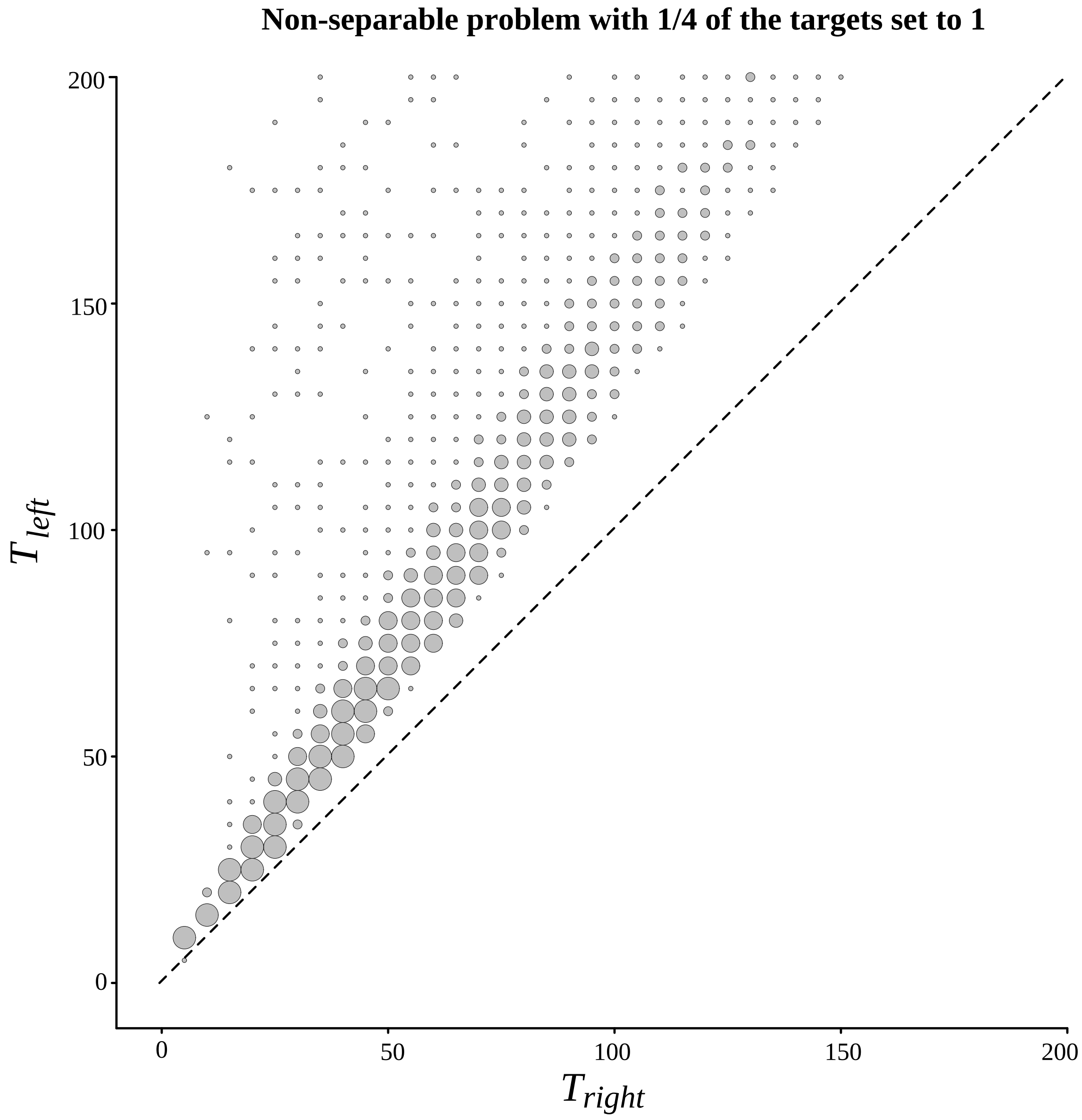

Figure 3.9: Performance on the random target problem with a quarter of the targets set to 1. The range denoted by circles is from 75% to 79.1% of the training set correct after 100 epochs of training. OFFSET = 0, MAX = 1.

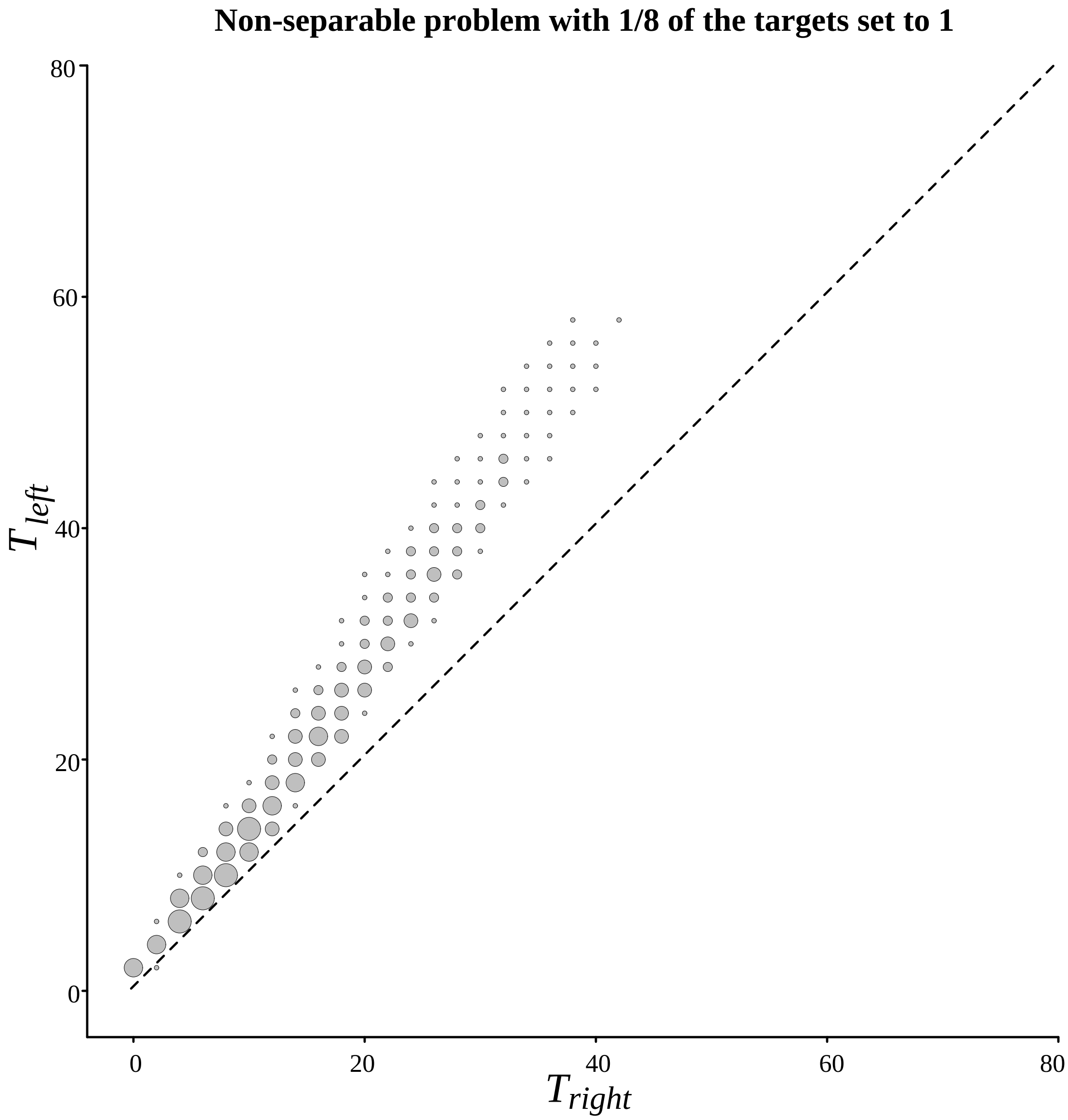

Figure 3.10: Performance on the random target problem with an eighth of the targets set to 1. The range denoted by circles is from 87.5% to 89.2% of the training set correct after 100 epochs of training. OFFSET = 0, MAX = 1. Note that the scale has changed from that used in the previous figure.

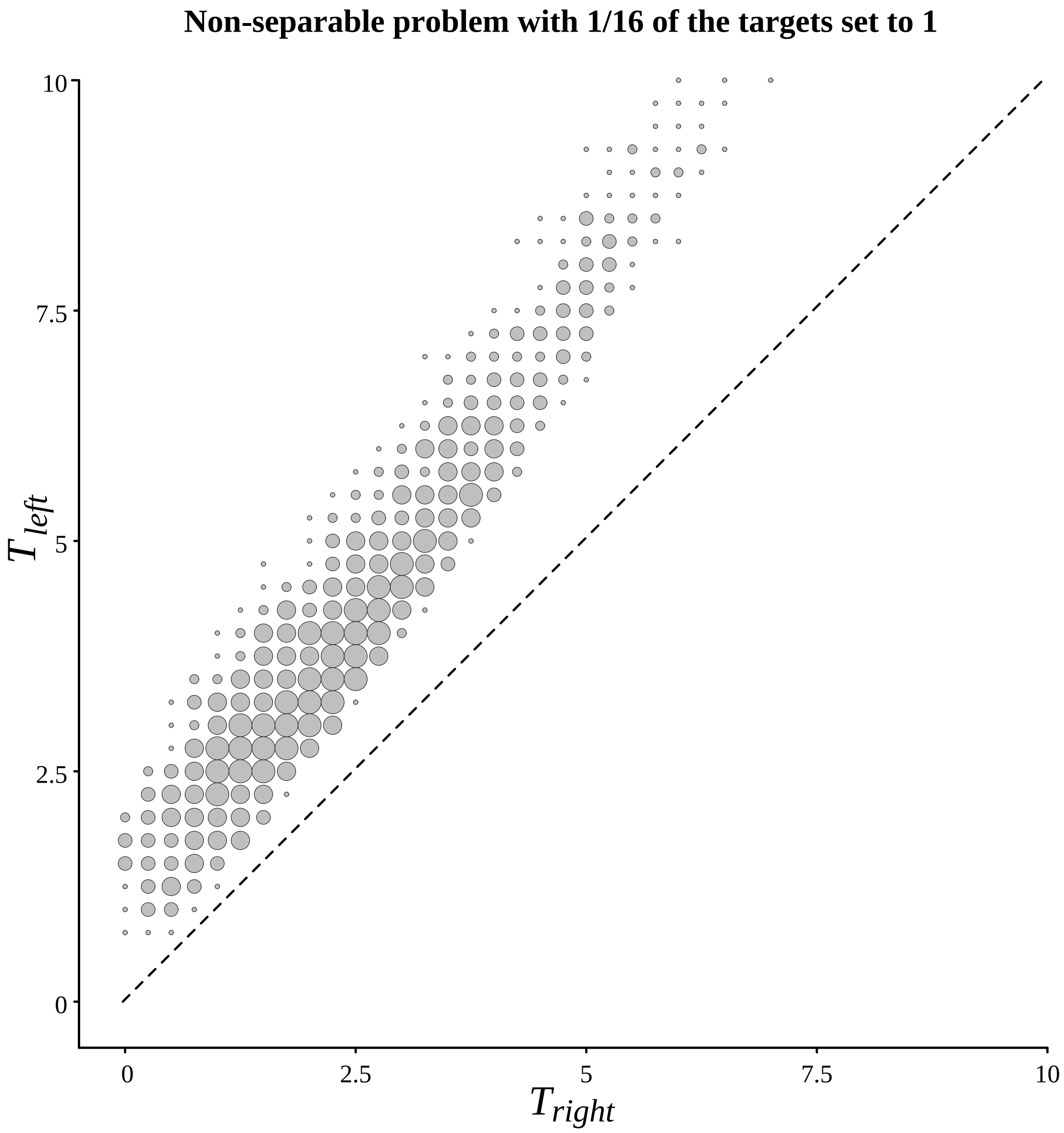

Figure 3.11: Performance on the random target problem with one sixteenth of the targets set to 1. The range denoted by circles is from 93.75% to 94.53% of the training set correct after 100 epochs of training. OFFSET = 0, MAX = 1. Note that the scale (0-10) is very much smaller than that of the 50:50 case (0-200).

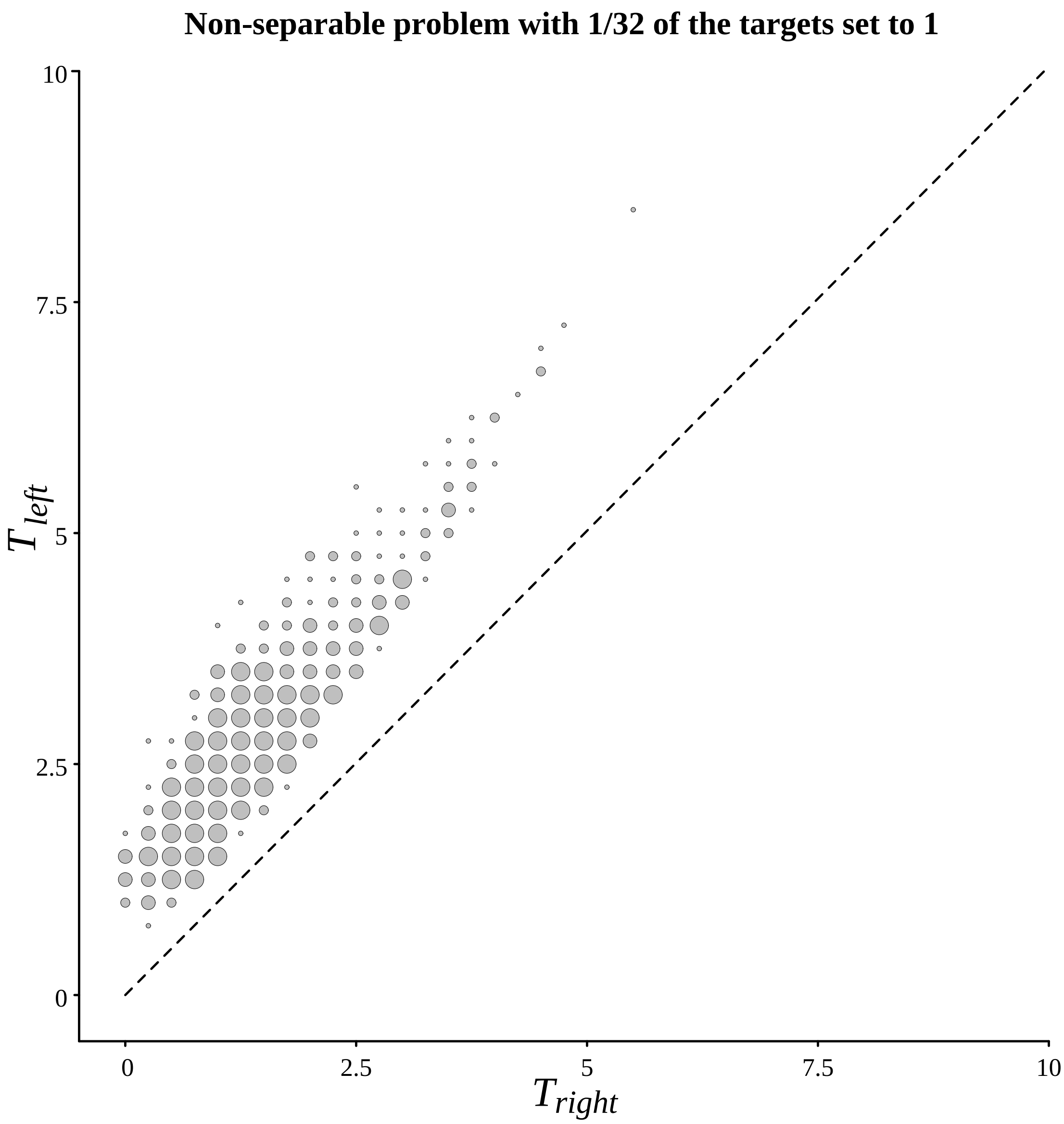

Figure 3.12: Performance on the random target problem with 1/32 of the targets set to 1. The range denoted by circles is from 96.875% to 97.27% of the training set correct after 100 epochs of training. OFFSET = 0, MAX = 1.

is from 50%, representing no improvement over a random choice of weights, to 100%, which corresponds to successful learning of the entire training set.

Figures 3.14 to 3.17 show the corresponding plots for linearly separable problems with only a $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$ and $\frac{1}{32}$ respectively of the input patterns targeted to 1 and the remaining targets set to zero.

However, this doesn't tell us how quickly this level of performance was achieved. A more descriptive measure for this task would be the number of epochs taken to find a separating hyperplane. There are actually two such measures of interest:

1. the earliest time that weights occur which give no errors
2. the latest time that any error occurs

These turn out to be very different in character, because a rule which quickly finds a separating solution is not necessarily stable. Consider figure 3.18, which shows the "earliest separate" case compared to the "last not separate" case. Firstly, the speed with which the first "perfect" plane is found looks very much like the usual score shown in figure 3.13. Secondly, even though a broad range of curves find separating hyperplanes quickly, only those with $T_{right} = 0$ are actually stable at this point.

3.6.5 The effect of MAX.

Recall that the parameter MAX plays the part of a learning rate for a curve of given OFFSET, T_{left} and T_{right}. Figure 3.19 shows the non-separable problem for $N = 8$ as MAX is varied (with OFFSET $= 0$). It's not obvious what the effect is - is it a broadening or simple translation of the curves as MAX is increased, or something else? Figure 3.20, which shows the performance for different rates with rescaled axes, makes it apparent that the effect is simply one of scaling the widths linearly with the rate: the effect seen in the previous figures is actually due simply to magnifying the pictures up. This makes perfect sense, because the two axes are linearly related: since ϕ is just a linear sum of weights, doubling the weight changes doubles the rate of build up of ϕ. Therefore by blowing up both dimensions of the curve, for example by doubling MAX, T_{left} and T_{right}, precisely the same weights vector would be generated given the same order of patterns, but with twice the magnitude. Since the unit calculates its output by a threshold operation the relative magnitude is of no consequence, so the scores are the same. In the simulations shown the order of presentation of the patterns is random and different in each trial, but on average the performance is the same.

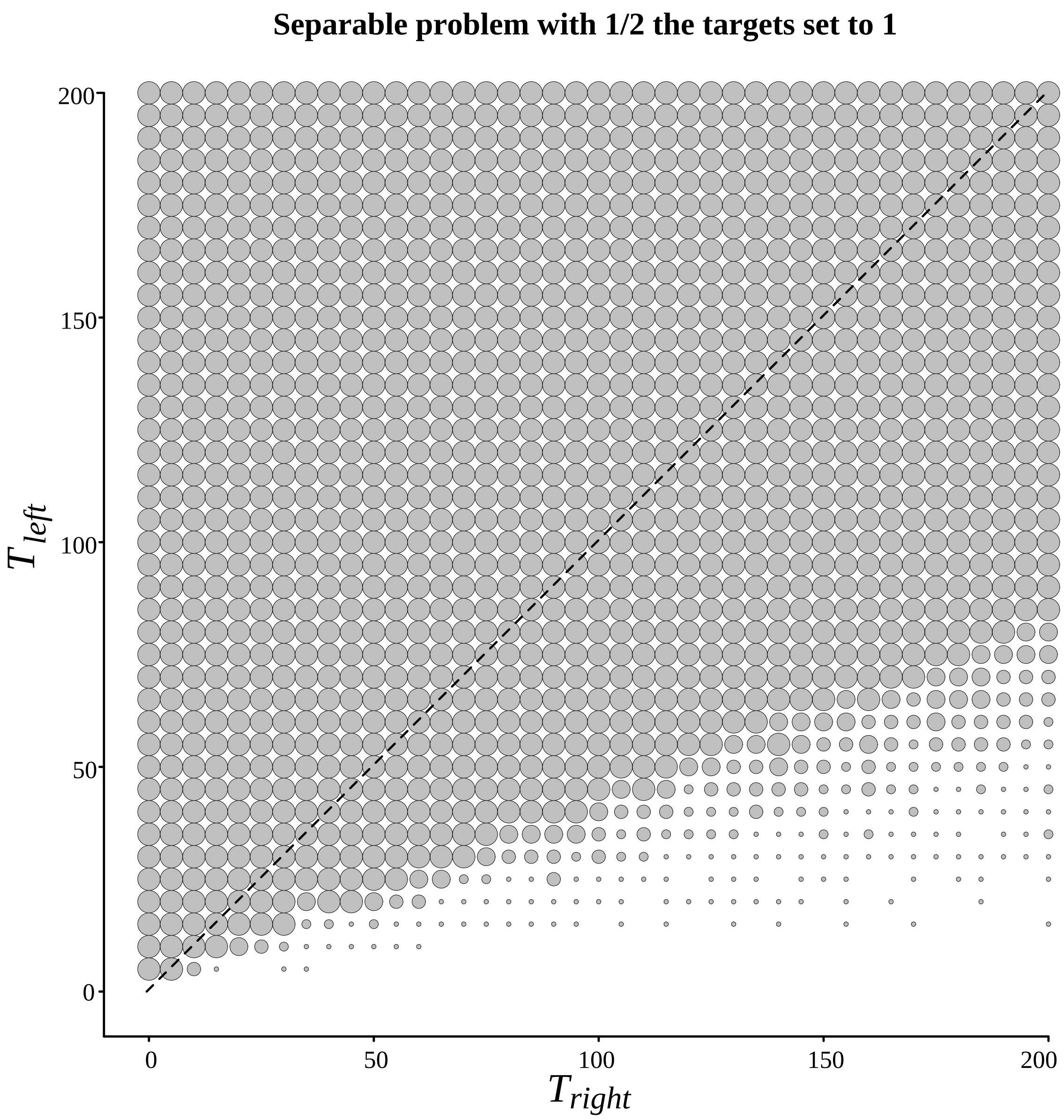

Figure 3.13: Performance on a linearly separable problem with half of the targets set to 1. The range denoted by circles is from 50% to 100% of the training set correct after 100 epochs of training. OFFSET = 0, MAX = 1.

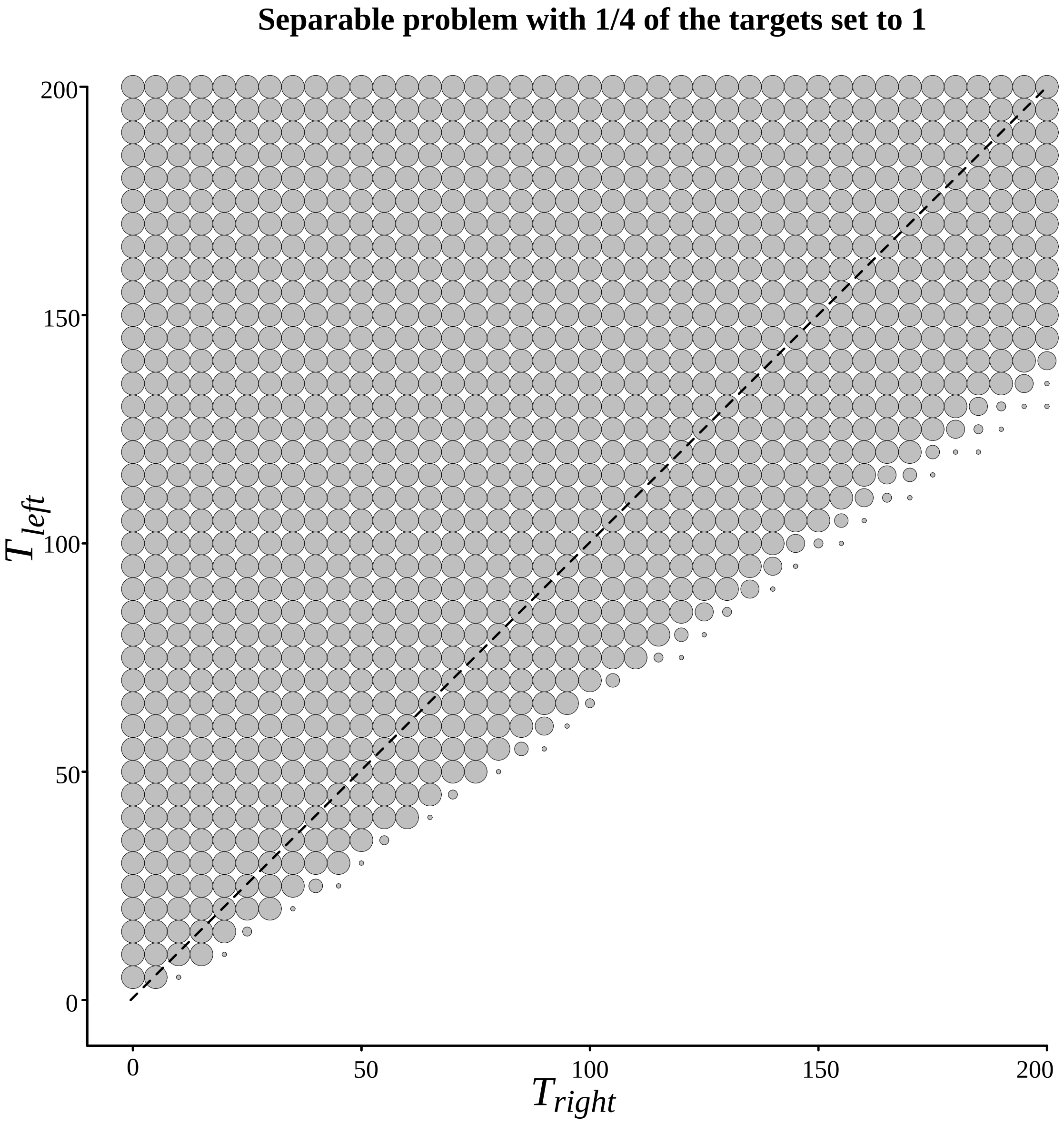

Figure 3.14: Performance on a linearly separable problem with 1/4 of the targets set to 1. The range denoted by circles is from 75% to 100% of the training set correct after 100 epochs of training. OFFSET = 0, MAX = 1.

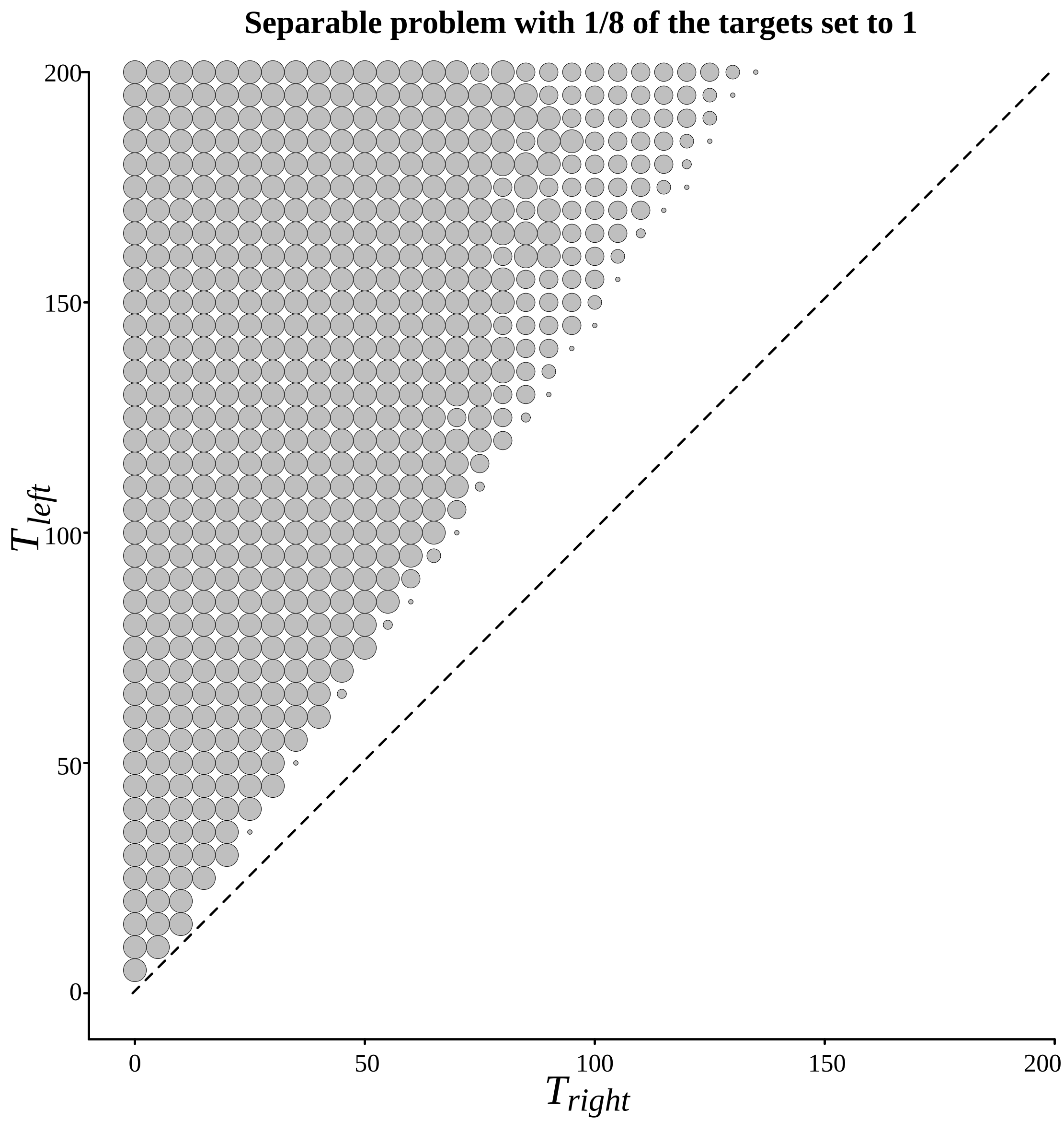

Figure 3.15: Performance on a linearly separable problem with 1/8 of the targets set to 1. The range denoted by circles is from 87.5% to 100% of the training set correct after 100 epochs of training. OFFSET = 0, MAX = 1.

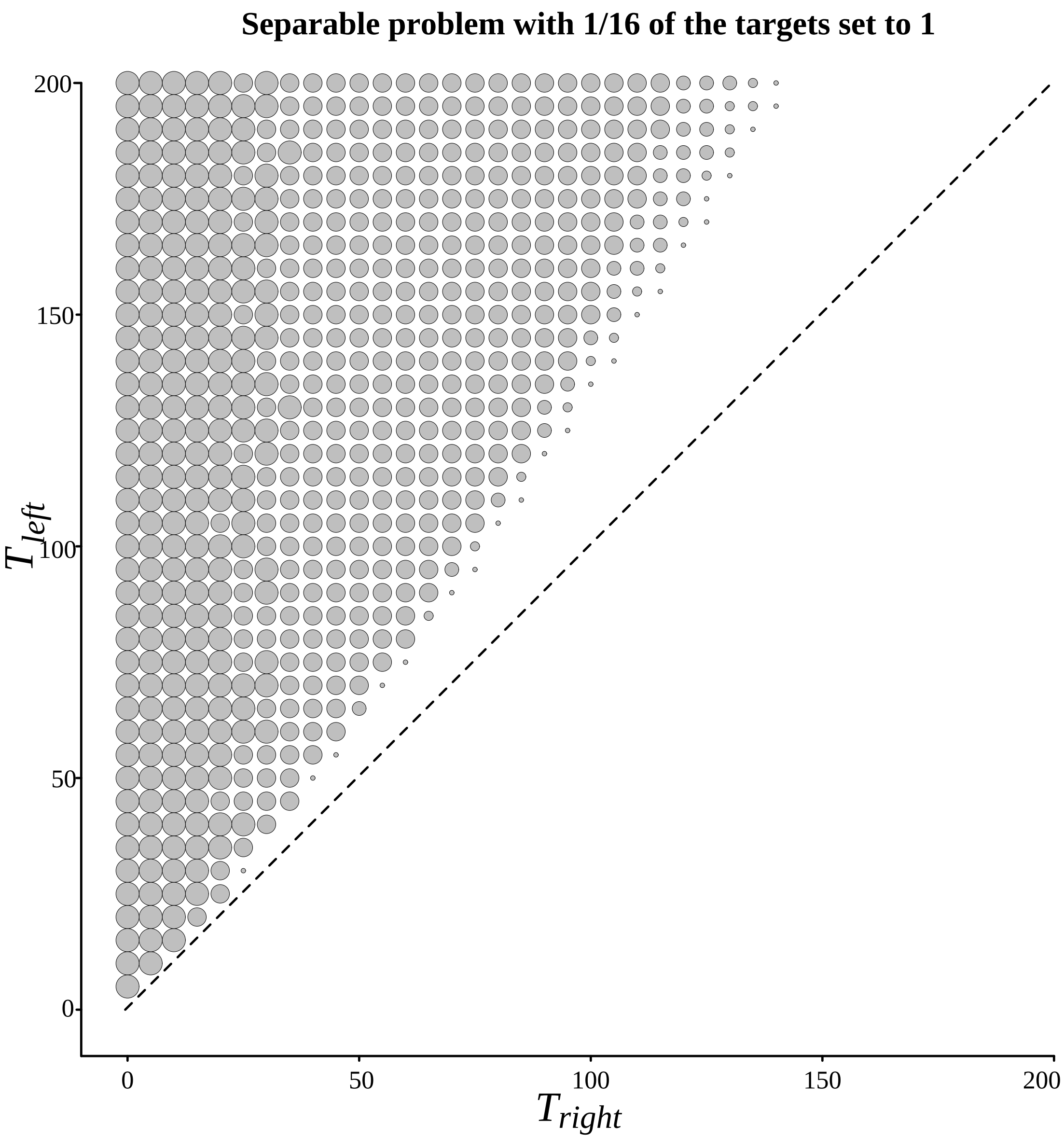

Figure 3.16: Performance on a linearly separable problem with 1/16 of the targets set to 1. The range denoted by circles is from 93.75% to 100% of the training set correct after 100 epochs of training. OFFSET = 0, MAX = 1.

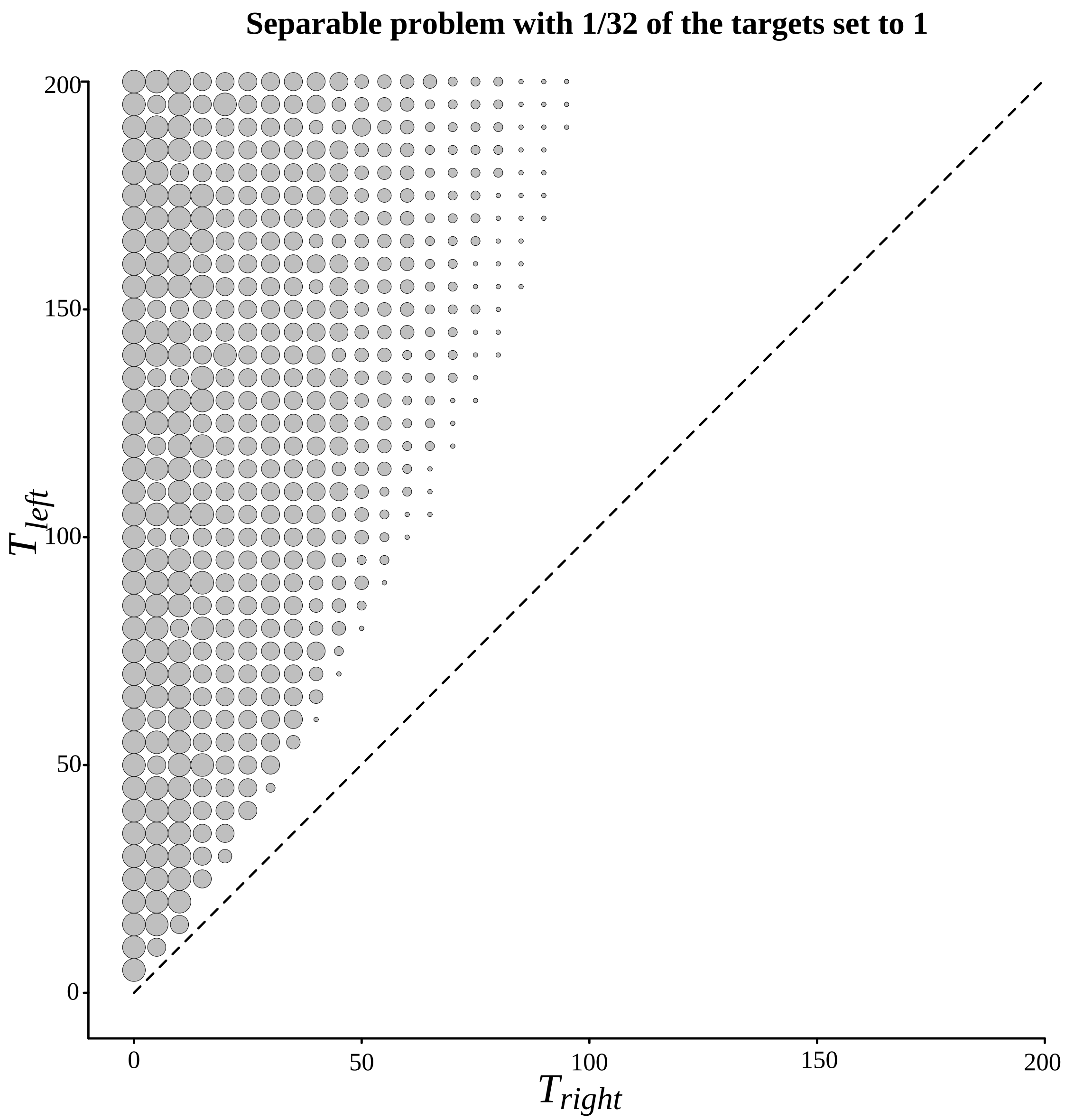

Figure 3.17: Performance on a linearly separable problem with 1/32 of the targets set to 1. The range denoted by circles is from 96.875% to 100% of the training set correct after 100 epochs of training. OFFSET = 0, MAX = 1.

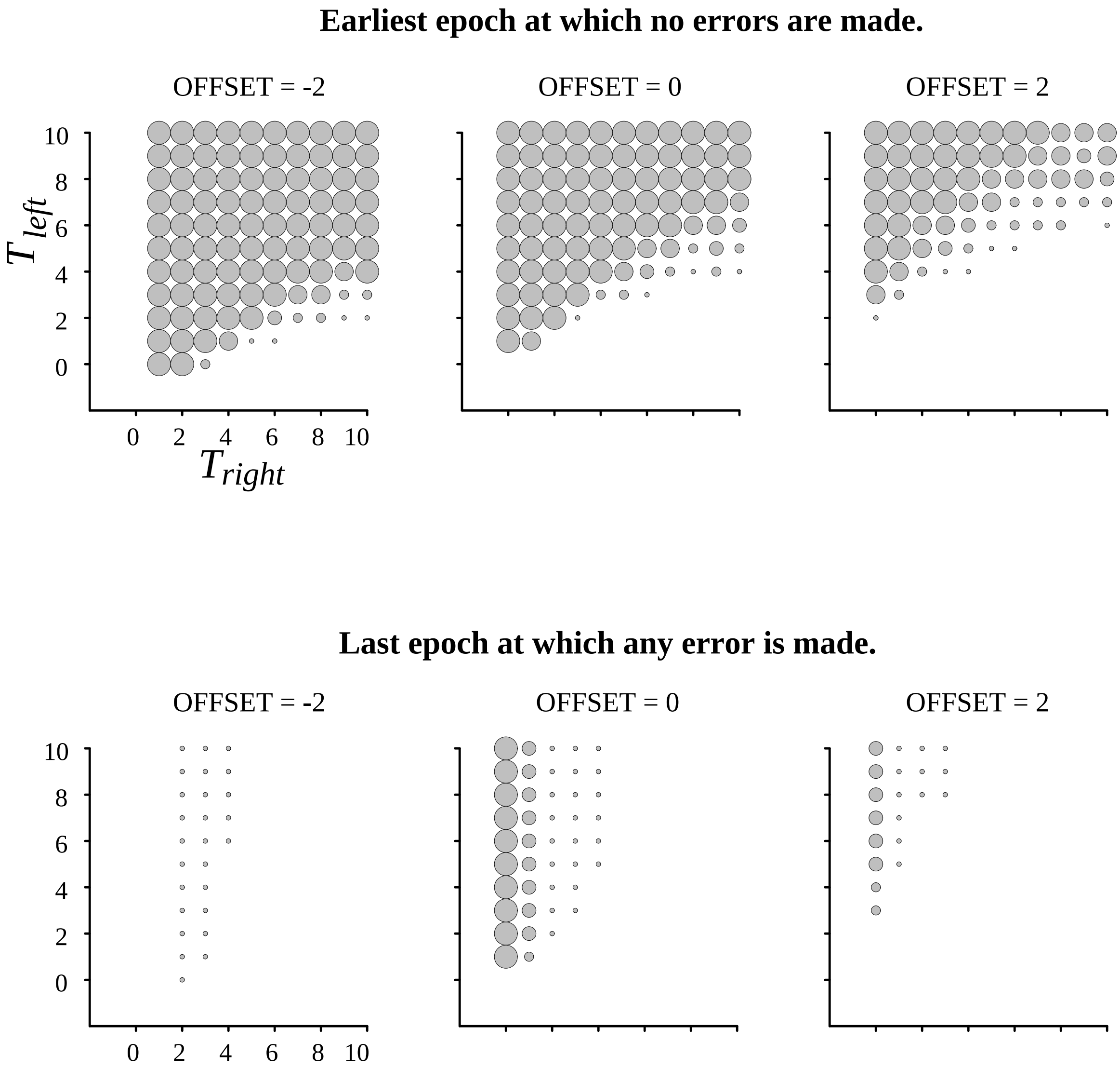

Figure 3.18: The earliest epoch at which no errors are made, compared to the last epoch at which any errors are made. The range is from 0 epochs (large circles) to 1000 (no circle) epochs. MAX is 1.

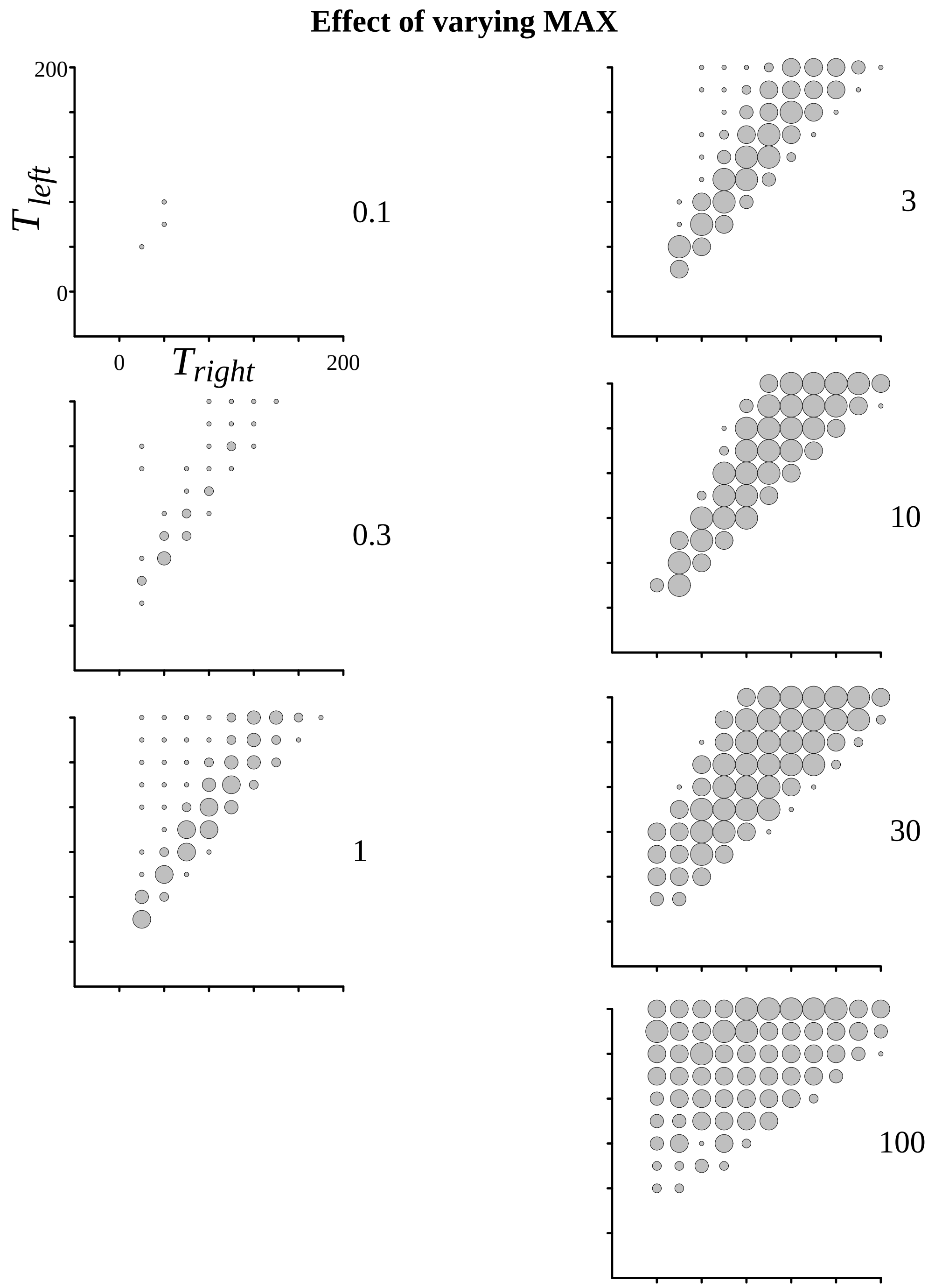

Figure 3.19: Effect of varying the value of MAX, over T from 0 to 200. Performance is shown for the random target problem using 256 patterns across 8 inputs with 25% of the targets set to 1. The range denoted by circles is from 75% to 79.3%. OFFSET is 0.

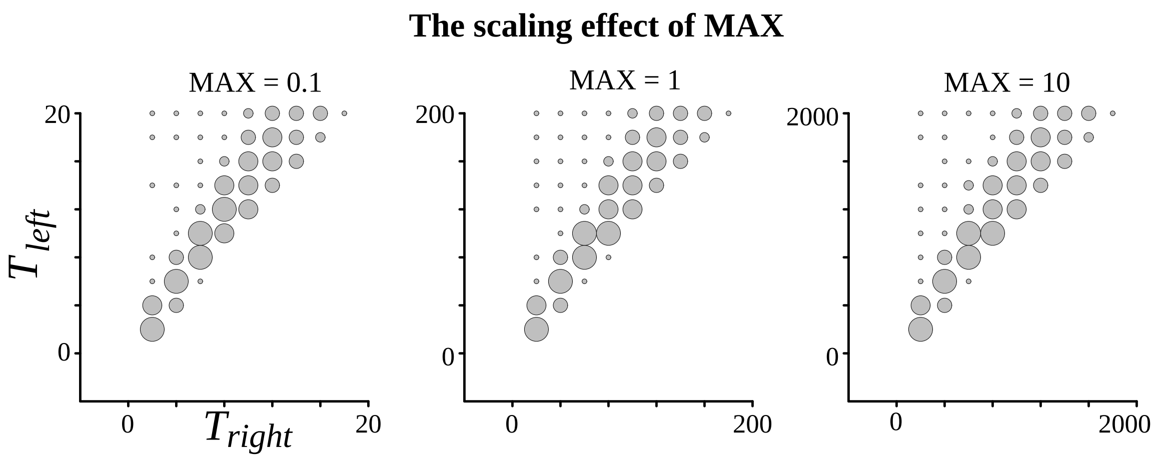

Figure 3.20: Effect of varying the value of MAX: the axes are scaled in each case in proportion to the value of MAX. Because all three pictures are similar, the effect is simply one of rescaling. The range denoted by circles is from 75% to 78.4%, and OFFSET is 0.

3.6.6 The effect of OFFSET.

Figure 3.21 shows the effect of varying the value of OFFSET. The same diagonal feature appears shifted, with its intersection with the line $T_{left} = 0$ moving by approximately the amount of the offset. Suppose that the curves used were actually "flat" out to a threshold at $\phi = (\text{OFFSET} - T_{left})$ and $(\text{OFFSET} + T_{right})$ respectively. If the OFFSET is altered by

$$\text{OFFSET} \quad \rightarrow \text{OFFSET} + \Delta$$

then changing

$$\begin{aligned} T_{left} &\rightarrow T_{left} - \Delta \\ T_{right} &\rightarrow T_{right} + \Delta \end{aligned}$$

reproduces the original curve. This corresponds to a shift of the line seen in the figures up and to the left by amounts equal to the increase in OFFSET. This is the apparent direction of the shift (however the actual movement is somewhat less than this). Hence a possible explanation for the observed shift is that an important feature determining performance is the simply the area under the curve to either side of the origin.

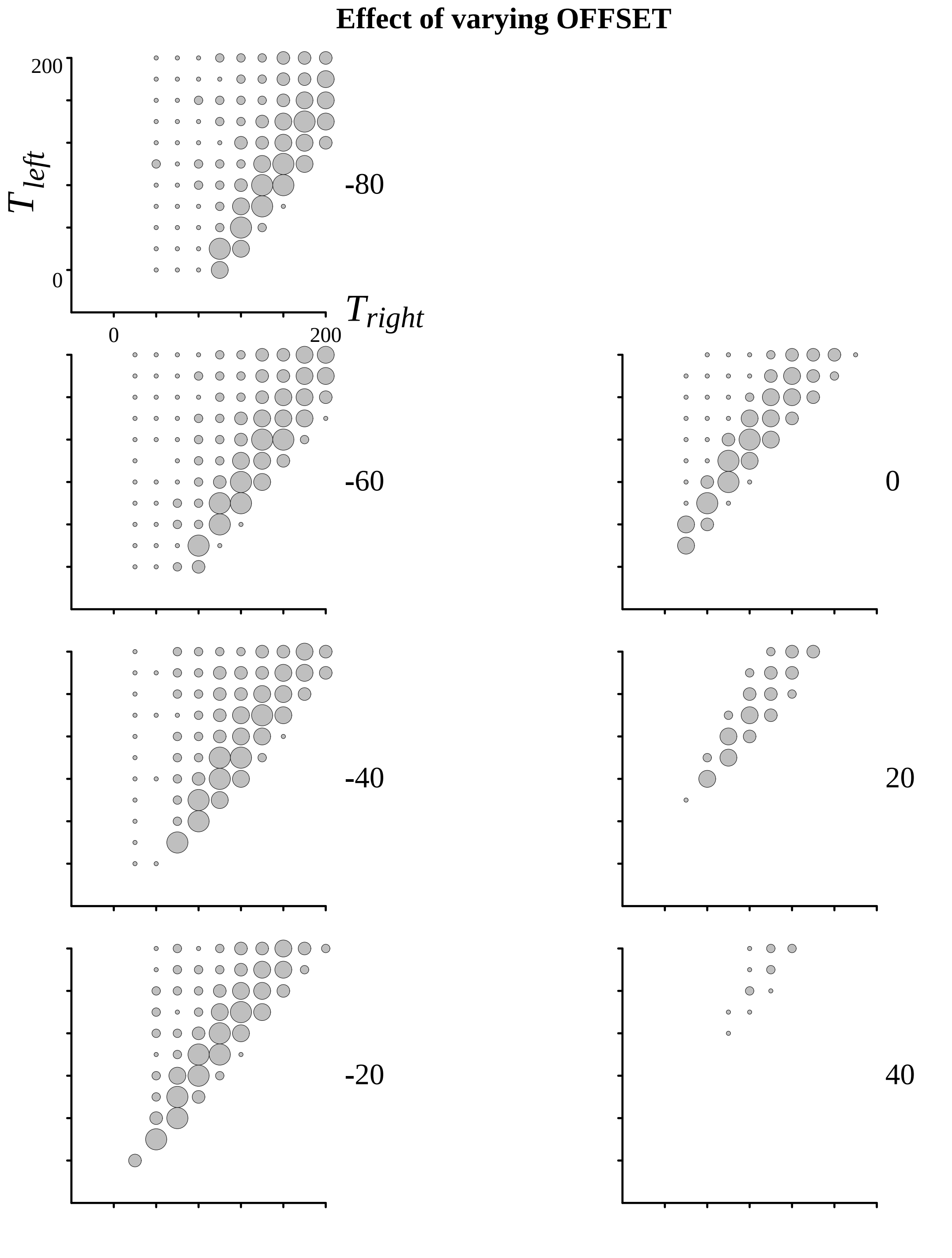

Figure 3.21: Effect of varying the OFFSET. Performance on the random target problem using 256 patterns across 8 inputs with 25% of the targets set to 1. The range denoted by circles is from 75% to 78.4%. MAX is 1.

3.7 Conclusions

It is easy to see why the Pocket algorithm is so slow in the highly non-separable case by looking at the mean errors made by the PLR perceptron: the "good" weights configurations are *very* infrequently visited by any rule with large T_{left} and small T_{right}.

It may seem surprising then that the Thermal rule works as well as it does, given that it has $T_{right} = 0$. However, it was seen that the performance without any annealing depends on the value of T, which is best set at close to unity for the problem used for figure 3.1, and this is also true for the $N = 8$ cases investigated here. T_{left} and the parameter T used in Thermal are roughly comparable, being the value of $|\phi|$ at which the weight change is $1/2$ and $1/e$ respectively of the maximum. Therefore the Thermal rule works best at approximately $T_{left} = 1$, $T_{right} = 0$, which is in the region of "reasonably good" rules as shown in figures 3.11 and 3.12. The annealing corresponds to moving slowly down the $T_{right} = 0$ line through the region where good weights are visited. Performance is less sensitive to the initial temperature in the annealed case because the "good" region is so close to $T_{left} = T_{right} = 0$, which is why the lower curve in figure 3.3 rises only at the very end of the annealing schedule. It would be interesting to investigate the effects of annealing in the two gaussian widths, as similar benefits would be expected to those seen for Thermal.

In summary, rules where $T_{left}/T_{right} \sim 4/3$ are best for the non-separable problems investigated here, whereas T_{right} should be close to zero for fast and stable convergence on separable problems. However there appears to be an intersection or "middle ground" in which performance on either type of problem is good. The parameter OFFSET can be varied over a large range but is as well set at zero as anywhere. The learning rate MAX is linearly related to the width of the curve and must be set accordingly to maximise performance.

3.8 Further work: Constructive methods revisited.

Chapter 2 showed the Upstart algorithm has improved performance over the algorithms discussed previously, and in the present chapter the Thermal perceptron rule has been shown to generate better weights than the Pocket algorithm for difficult problems. Hence it is not yet clear which of the constructive algorithms is really the best and under what conditions. This question requires a careful theoretical and practical evaluation of all the various approaches. A complicating factor is that the two issues of how best to train the weights and how best to construct the network may be coupled. That is, a particular weights algorithm may well favour a particular constructive approach and *vice versa.*

This work remains to be done. Preliminary results of applying the Thermal rule to other algorithms are given here, purely as a pointer to future work. The learning task is the "two or more clumps" problem discussed in Chapter 2. The Tiling algorithm is not included since it may be viewed as a combination of the Splitting and Tower methods. Tower and Splitting can be implemented using the Thermal rule with no other changes. An adaptation is required for use in Whittling; this method requires that weights be learned which make only "wrongly OFF" errors, so a hidden unit which makes a single "wrongly ON" error is unusable. The modification (which replaces a complicated method based on the Pocket algorithm) is simply to apply the original PLR to patterns whose target is 0 and the Thermal rule to those whose target is 1.

The "clumps" problem used here is exactly the same as that used to demonstrate the Upstart algorithm. The results are shown in figure 3.22, where the Upstart results are included for comparison. Somewhat surprisingly, the Whittling algorithm produces the smallest networks of all, and shows the best generalisation performance. This might be expected if the problem consisted of an obvious conjunction of several separable features, but this is not the case with the clumps problem. Also noteworthy is the fact that although the Tower algorithm consistently builds smaller networks than either Split or Upstart, its generalisation performance is slightly *worse.* Further simulations were done confirming that this is not improved by allowing more connections between Tower units than one to each unit's immediate predecessor.

3.9 Summary

Constructive algorithms constitute an important addition to the field of neural networks. They are powerful methods for producing networks to perform classification tasks, and do this by learning in a way which is quite different from the majority of network methods presently in use. An understanding of learning rules for individual perceptrons is an enterprise of comparable importance, given the striking extent to which the performance of these constructive methods is dominated by the choice of such a rule.

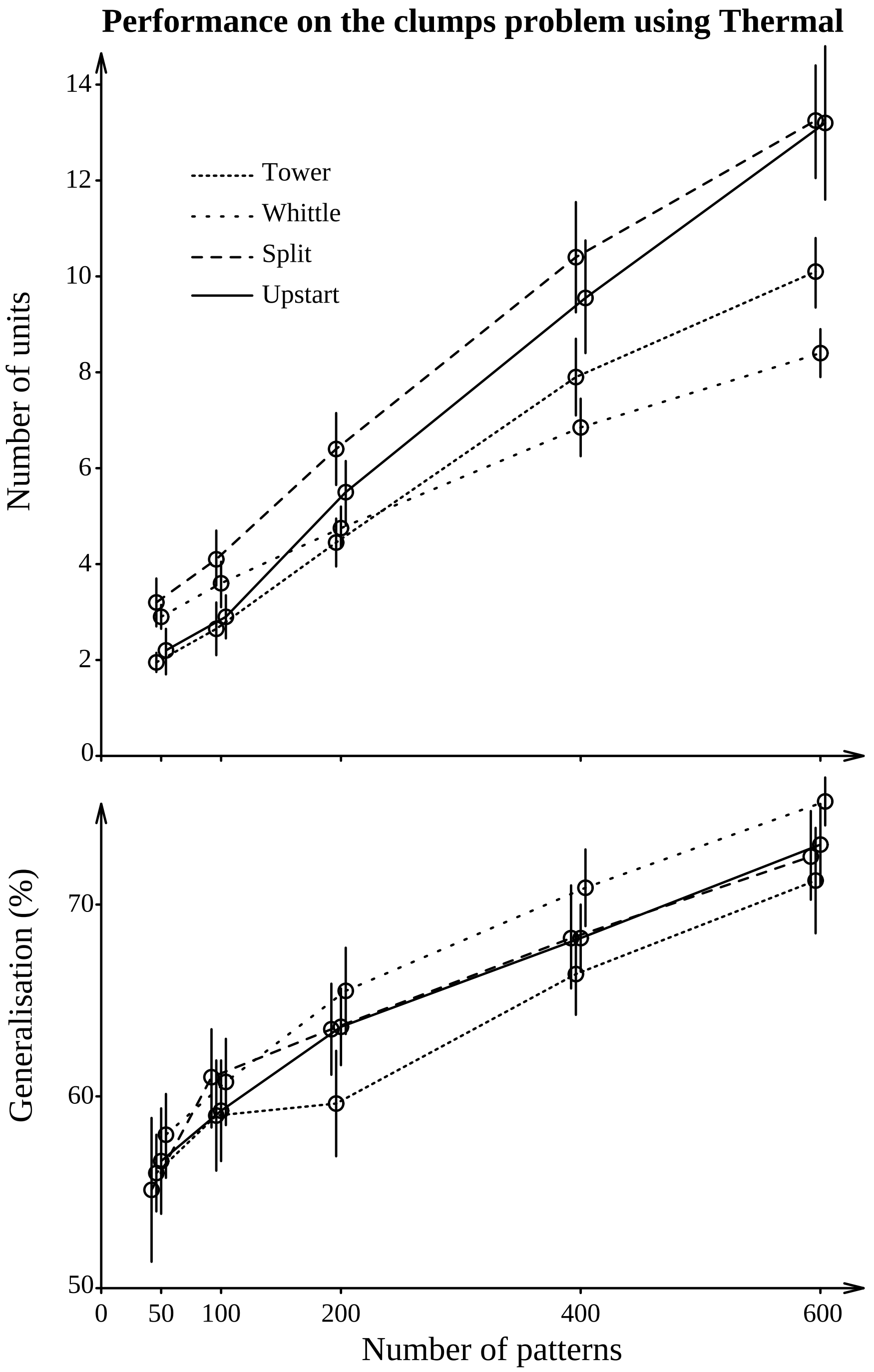

Figure 3.22: Comparison of the constructive algorithms incorporating the Thermal rule on the "two or more clumps" problem. The graphs show the size of the networks and their generalisation performance, *vs* the size of the training set. The circles and vertical bars show the mean and standard deviation respectively over 25 trials. $T_0 = 1.5$ for all methods except Whittle, where it is 2.5

Part II

Topographic mappings and the Travelling Salesman Problem

Chapter 4

Introduction to the TSP

4.1 Overview

The following two chapters are concerned with the Travelling Salesman Problem (TSP), a well known and widely studied problem in combinatorial optimisation. Finding the optimal solution is a computationally intractable task, hence heuristics for finding good solutions in reasonable time are of more practical interest.

In this chapter, the TSP and the conventional methods for solving at are briefly reviewed, and the key concepts introduced by Hopfield and Tank relating neural networks to the problem are pointed out.

Chapter 5 is motivated by the success of the topographic mapping approach as embodied in the Elastic Net algorithm (Durbin & Willshaw 1987). A proposal is put forward to overcome a restriction inherent in the Elastic Net approach, which leads to two novel algorithms for the TSP.

4.2 Outline of the problem.

'Given a set of cities and the distances between them, construct a closed

tour of minimal length which visits each city exactly once.'

This is the usual statement of the Travelling Salesman Problem. In other words, given N elements (for example cities, processes, or states) and the set of N^2 scalars relating one to another (for example distances, costs or differences) construct a closed cycle including each element exactly once such that the sum of the scalars is as small as possible. The TSP can also be stated as the decision problem: "Given a set of cities and the distances between them, does there exist a tour of length less than L?"

Only in certain cases can the TSP be formulated in terms of the actual positions of cities. An N-by-N matrix specifying all the distances between the N cities is sufficient to specify an instance of the problem. The most general case is called the Asymmetric TSP (see figure 4.1). For this, the matrix elements are arbitrary positive quantities, and the distance from A to B need not necessarily equal that from B to A. This could be visualised in terms of real cities, connected by one-way roads (*ie.* the lengths aren't symmetric) which are not straight. Important subclasses are problems for which the distances are symmetric (two-way roads), and those where distances between any three cities obey the triangle inequality (*ie.* the distance between two cities must be less than the sum of the distances from any third city to each of them), and also problems for which both conditions are true. A further constraint may be that the distance matrix correspond to straight-line distances between cities lying in some Euclidean hyperspace. The TSP in the plane is the simplest and most often quoted example of these, corresponding to perfectly straight, two-way roads. At first sight it might appear that the non-Euclidean problems could be Euclidean but in a space with a higher number of dimensions, provided the triangle rule is not violated. However, as the example in figure 4.2 shows, this is not true in general: even slight perturbations of the distances in a Euclidean instance of the TSP turn it into a non-Euclidean problem. This has important consequences for the algorithms put forward in the next chapter.

4.2.1 Why is the TSP interesting?

The TSP combines simplicity of statement with difficulty of solution. Even for small numbers of cities a complete search is out of the question. The number of possible tours for N cities is (N-1)!/2. For only 50 cities there are of the order of 10^{65} valid tours, which would require billions of years of computing time for an exhaustive search. More importantly no algorithm has been found which can produce the shortest tour in a time which scales reasonably with the number of cities. For comparison, consider the Minimal Spanning Tree problem (MST): "Given a set of N points and the distances between them, construct the minimal

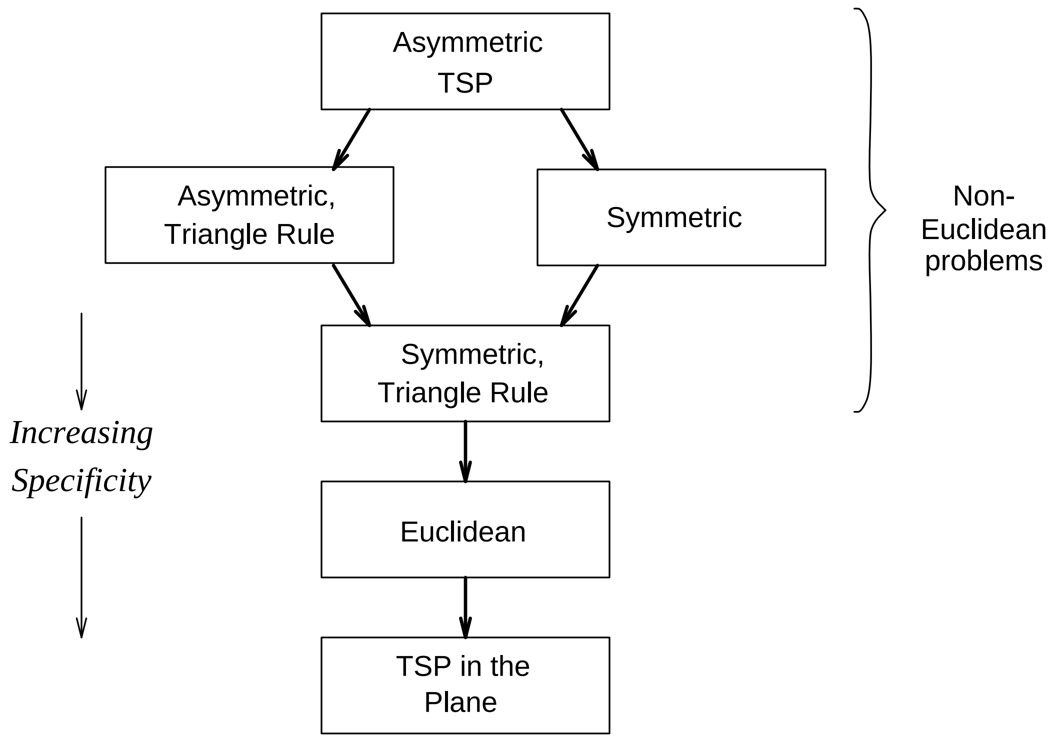

Figure 4.1: The Hierarchy of TSP's.

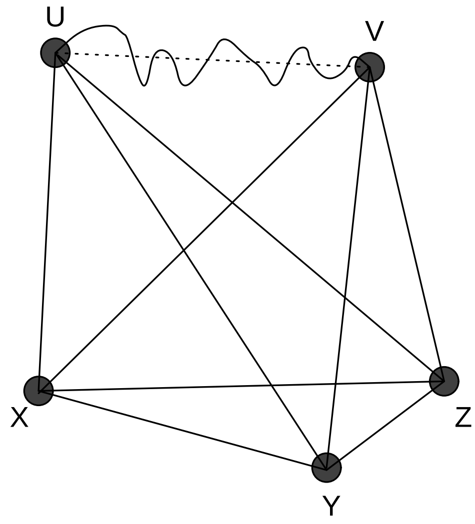

Figure 4.2: Problems that obey the triangle rule are not necessarily Euclidean. For example, suppose we have five cities in the *2D* plane as shown. Now consider altering only one distance, that from U to V, by putting "bends in the road". This cannot be recast in terms of straight-line distances in a higher dimensional Euclidean space, because the (unchanged) distances from X, Y and Z specify that both U and V must remain in exactly the same *2D* plane as these cities.

length graph spanning the points". The TSP is a restriction of this problem in which this graph must be a cycle. Equivalently a search of all N^{N-2} possible trees quickly becomes unfeasible, but unlike the TSP there does exist an algorithm which finds the minimal spanning graph in time proportional to N^2. The differences in algorithmic scaling between these two problems are captured by the notion of the class *NP*-Complete, of which the TSP is a member but the MST problem is not. The class *P* is the set of problems, such as the MST, for which algorithms exist which complete in polynomial time (that is, a number of steps which is some polynomial function of the problem size). Now consider a decision problem, and an algorithm which constructs a solution in a non-deterministic way and tests the decision condition. If such an algorithm can form a solution with some (perhaps negligible) probability of satisfying the condition and test it in polynomial time, then the problem is in *NP*. Note that *P* itself is part of *NP*. The so-called SAT problem (standing for 'satisfiability') is the 'hardest' problem in *NP*, since every other problem in *NP* can be transformed to it in polynomial time. Further, it is possible to show whether a given problem in *NP* is *as hard as* SAT, which then defines the class *NP*-Complete: the hardest problems in *NP*.

Since the advent of this idea, a large number of problems long suspected of being intractable have been shown to be members of this class. Examples of other important problems in *NP*-Complete are:

- Multiprocessor Scheduling Problem. Given tasks to be performed and times for each, minimise the total time involved for a given number of sequential processors operating concurrently.
- Hamiltonian Circuit Problem. Given a partially connected graph, is there a closed circuit visiting each node exactly once?
- Map Colouring Problem. Is it possible to colour a given map using 3 colours so no border has the same colour on both sides?

Since all problems in *NP*-Complete are equally hard, and all are reducible to SAT, in effect all are *mutually transformable*. This means that *if* a polynomial-time algorithm exists for any one of them, then there exists such an algorithm for every one of them. Although this doesn't preclude the possibility that such an algorithm does exist, it is generally agreed to be extremely unlikely.

The TSP itself is an important practical problem since it has applications in many different fields, from silicon chip layout design to commercial transportation routing and airline personnel timetabling. Also, given the close formal relationship between all members of *NP*-Complete, it is hoped that successful approaches to TSP may carry over to these other problems of similar complexity.

Finally, the TSP has become a standard by which various general approaches to combinatorial optimisation problems are measured. When a new approach is proposed to such problems, it is common for its potential to be evaluated by attempting this problem.

4.2.2 Conventional methods

These can be divided into heuristics for finding good tours and exact methods for finding the optimal tour.

Since guaranteed optimal solutions to *NP*-Complete problems cannot be found in polynomial time at present (and there are strong doubts they can be in principle) there inevitably arises a trade-off between the quality of solutions and the speed with which they can be found. A lot of interest naturally focuses on heuristics for finding good solutions quickly. An enormous amount of effort has been expended on such suboptimal methods for the TSP in particular. Some examples of such heuristics are:

Tour Construction procedures. Examples are:

- **Nearest neighbour or 'greedy' algorithm.** Starting from an arbitrary city, simply connect to the nearest cities not already included until a tour is formed with the last link. This is the most naive way of constructing a tour and, not surprisingly, generates very bad tours.
- **Minimal Spanning Tree (MST) methods.** These work by constructing the MST and converting it into a tour. The best way of doing this is known as Christofides Algorithm, which doubles every edge of the MST to produce an Eulerian graph (*ie.* one in which each vertex has an even number of edges attached to it), finds a tour for this graph, and converts it to a Travelling Salesman tour by using shortcuts. For any TSP instance where the city distances obey the Triangle Inequality, this guarantees a tour of not more than $\frac{3}{2}$ of the optimal. This is the best such 'worst case' condition known for such a heuristic.

Tour Improvement procedures. In general these work by repeating the following two steps:

- **Step 1.** Modify an existing tour Q of length L slightly, producing Q_{new}
- **Step 2.** Compare Q with Q_{new} and adopt Q_{new} under some condition involving the difference between the tour lengths.

Step 1 is often done using a procedure called *r-opt* (Lin 1965) which involves making r cuts in the tour and reassembling the pieces in all the various combinations. In that case the condition in Step 2 is simply that the new tour be shorter. An 'r-opt' tour is the shortest tour found by trying all possible combinations of r cuts. Empirically it is found that the probability of a 3-opt tour being optimal is about 5% for 50 cities (Lin 1965). Genetic algorithms (Holland 1975) have recently been applied to perform Step 1 by dealing with mutations and combinations within large populations of tours (Mühlenbein *et al.* 1988). In the method of Simulated Annealing (Kirkpatrick *et al.* 1983) the condition in Step 2 is evaluated nondeterministically, so there is some probability of longer tours being accepted. If the new tour length L_{new} is shorter than L it is always accepted, but if it is longer then it is only accepted with probability $e^{-(L_{new}-L)/T}$. In the exponential, T plays the role of a "simulated temperature". This value may be "annealed" very slowly from a high temperature where virtually every change is accepted, to a low one where only strict improvements are accepted.

Composite procedures. The best known of these is due to Lin and Kernighan (Lin & Kernighan 1973). The method is fairly complex, with the core idea being the dynamical setting of r, the number of cuts made when using r-opt.

For exact solutions the problem is treated very differently by so-called polyhedral methods typified by the *branch and bound* and *cutting plane* procedures. In these a geometric space is examined in which the tours are seen as points, and an initially all-enclosing bounded region of this space is considered. The idea is progressively to shrink the volume of this region by rejecting more and more potential tours, making sure an optimal tour is never rejected. In this way we can know that the last tour left in the region is the optimum. A dominant characteristic of these methods is their high degree of complexity. As well as being very specialised and complex, several techniques usually need to be combined into hybrids to make them work effectively. [Lawler *et al.* 1985] describes the polyhedral methods in detail.

All of the algorithms mentioned above are concerned with manipulating links between cities in a 'binary' fashion: cities are either linked in the tour or not. In a sense then they conduct a search in the binary hyperspace of possible links. The current tour is representable as a point at one of the vertices of this hypercube, with the algorithm jumping from vertex to vertex in the heuristics, or working to enclose as few as possible vertices in the polyhedral methods.

4.3 Neural networks: the method of Hopfield and Tank.

Hopfield and Tank proposed a new method for finding short tours in the TSP (Hopfield & Tank 1985), using a neural network approach. Their method introduced two important new approaches to solving optimisation problems using networks: searching a continuous state space for solutions to a discrete problem, and the use of energy functions.

Assume a given problem can be stated in terms of some finite number of variables bounded between zero and one. If these state variables are represented by mutually orthogonal vectors then we have a Cartesian or configuration space which is a hypercube. Hopfield and Tank introduced the idea that searching a continuous state space can have advantages even when the valid solutions exist only at the vertices of the hypercube. In a sense this is a more general approach than methods which deal only with operations concerning corners of the hypercube. In effect, additional degrees of freedom have been introduced, which the algorithm must ultimately remove for the system to converge to a valid solution.

Hopfield and Tank's technique was first to express the quantity to be optimised, and the various constraints of the problem, as minimisation problems over continuous variables. That is, they found a scalar function for which the optimal solution (*ie.* the shortest tour length) of the given problem corresponds to a minimum, and similarly they found functions at whose minima the constraints of the problem are satisfied (*ie.* valid tours). They called the sum of these functions an *energy function*, since the optimal solution (which should be stable) is attained if and only if the function's value reaches its global minimum. This is analogous to the ground state energy in physics. The problem is then to move through the space of possible states from the starting point towards the energy minimum.

If the free variables are denoted by s_i with E some function of all s, and the dynamics of a system are such that

$$\begin{aligned} \frac{ds_i}{dt} &= -\kappa \frac{\partial E}{\partial s_i} \\ \text{with} \quad \kappa &> 0 \end{aligned} \tag{4.1}$$

then changes to the system move it towards states of lower E, provided the "step size" κ, is very small. Dynamics such as this are known as *gradient descent* in E. Identifying E with the energy function described above enables the appropriate equations of motion in s to be deduced. The energy function assigns a scalar value to every point in this space, and equation 4.1 determines the resulting motion. In this way combinatorial optimisation problems can be formulated in terms of the descent of some *energy surface* in a continuous space of possible configurations.

A solution to the TSP is an ordering over cities, and one way of expressing this is by a square matrix whose rows correspond to cities and whose columns denote their ordered position in the tour. A valid tour is then defined if this is a *permutation* matrix wherein exactly one element of every row and every column is 1, and all other elements are 0. Hopfield and Tank associated each matrix element with a unit of a neural network. If there are N cities there are N^2 units, each denoting a particular city (here indexed in upper case) at a given position in the tour (indexed in lower case and assumed modulo N). The presence or absence of city A at position i is represented by the *activation* V_{Ai} of the appropriate unit, which is bounded to between zero (signifying absence) and one (presence) by the sigmoid function:

$$V_{Ai} = \frac{1}{1 + \exp(-\beta\phi_{Ai})}$$

The "gain" parameter β can be varied. The function is approximately linear over a wide range of ϕ if β is low, but turns into a step function as β is increased. The unit's *potential*, denoted ϕ_{Ai} is a weighted sum of all the other units' activations:

$$\phi_{Ai} = \sum_B \sum_j W_{(Ai)(Bj)}\, V_{Bj}$$

This architecture is known as a *recurrent* network due to the bidirectional interactions between all connected units. [Hopfield 1982] and [Hopfield 1984] showed that recurrent networks with symmetric connections have energy functions of a simple form. The object then as far as the TSP is concerned was to make the stable states of such networks correspond firstly to valid tours (permutation matrices) and secondly to short tours. This in turn can be done by setting the interconnections in the right way or equivalently by choosing a particular form for E:

$$E = E_{constraint} + E_{tourlength}$$

Unfortunately the existence of an energy function of this type is by no means a guarantee of success for the algorithm. Most importantly, the two terms are linearly additive even though the first is a constraint and the second is the condition to be optimised *given* that the constraints hold. The relative strengths chosen for the constraint and tour length terms are therefore of critical importance. If they are incorrectly chosen it is possible to trade-off the tour length against the constraints which make the network's configuration represent a valid tour. This problem afflicts the above method, to the extent that it is unusable in practice. [Wilson & Pawley 1988] tried unsuccessfully to reproduce the original results, obtaining valid tours in only 8 % of trials using 10-city Euclidean problems, for example. A search of possible parameter settings produced no combination which gave a valid tour for problems with 16 cities. In addition [Hegde *et al.* 1988] concluded that the space of usable parameters rapidly shrinks as the number of cities is increased. [Peterson & Söderberg 1989] report on a closely related approach called a "Potts neural network", in which the constraint that the sum of the activities of all units in a column be unity is enforced *exactly*, instead of being just one of several terms being minimised, as was true of the original model. Recent results (Peterson 1990) indicate that provided the gain parameter β and its annealing regime are carefully chosen, the method gives tours of the order of 20 % longer than optimal for Euclidean problems of up to 200 cities.

4.4 Advantages of using neural networks.

The use of continuous variables as described is a novel way of solving the TSP which may have advantages over conventional methods. As [Hopfield & Tank 1985] point out, values lying between the extrema at which a tour is defined represent a preference for a *set* of related tours rather than a particular tour. Hence such intermediate states may be interpreted as the simultaneous consideration of many similar possible tours, one of which eventually wins out. Two other important advantages of using neural networks are:

Parallelism.
A potential advantage of using neural networks is that the computations they perform are intrinsically parallel: instead of designing a serial algorithm in the usual way and then attempting to parallelise it, which is neither easy nor necessarily successful, the form of the algorithm is constrained to be parallel from the start. Parallel computers are beginning to revolutionise computation, and methods which can exploit their power are naturally of great interest.

Algorithmic simplicity.
In almost any real application, the exact nature of the trade-off between the quality of a solution and the time required to find it is crucial: we may want good solutions in minutes in one case, and *very* good solutions in hours in another, or as good a tour as possible in say 14 seconds in a third case. As will become apparent the "neurally inspired" methods, particularly those which involve an annealed parameter, are easily adapted to the different cases. This is not so true of the conventional methods (perhaps excepting Simulated Annealing), which tend to be very good at one or other limit of the trade-off.

Chapter 5

Topographic mappings methods for the TSP.

5.1 Overview

In this chapter a new approach to the TSP is examined, based on the formation of a topographic or neighbourhood-preserving mapping by a feed-forward neural network. A topographic mapping between two regions is one in which points which are neighbours in one region project to points which are themselves neighbours in the second region.

The new approach follows on from the Elastic Net (*EN*) method (Durbin & Willshaw 1987), which can produce impressive solutions to the TSP by forming a topographic mapping. However it is limited to the case where the cities lie in a Euclidean space. It is proposed that this restriction is surmountable by applying similar ideas to the development of a mapping between two distinct layers of units in a neural network; this enables the general symmetric TSP to be tackled. Forming the mapping amounts to learning the weights connecting the two layers of units. After showing how such a mapping can be interpreted as a tour of the cities, two new methods for solving the TSP by developing such a neighbourhood-preserving mapping are investigated. The first of these is a direct application of the Elastic Net procedure to the layered architecture, which results in a simple local learning rule for the weights. The second is an adaptation of an early model for the development of topographic mappings in the brain. Finally, the two methods are tested on a number of problems ranging from perfectly Euclidean to highly non-Euclidean.

5.2 The Elastic Net method.

In 1987 Durbin and Willshaw found a new approach applicable to Euclidean TSP's. Their idea hinged on the observation that a valid solution to the TSP is a mapping between cities and positions in a loop, and that short tours correspond to neighbourhood-preserving mappings. This is because in such a mapping nearby cities assume nearby positions on the loop, giving a short tour. The *EN* method has its origins in biologically realistic neural network models for the formation of topographic mappings in the brain; [Willshaw & von der Malsburg 1976] proposed the Neural Activity model (section 5.5), followed by the Tea Trade model (Willshaw & von der Malsburg 1979) which was adapted to become the *EN* method for the TSP. The method heavily exploits the restriction that the city distances correspond to points in some Euclidean space, usually the *2D* plane. The loop is positioned in this space along with the cities, and each loop point is free to move about under the influence of two kinds of force. The first kind pulls it towards the cities. For the j^{th} loop point situated at position $\vec{y}_j$ in the plane this force is:

$$\begin{aligned} \vec{F}_j^{city} &= \sum_i \psi_{ij}(\vec{x}_i - \vec{y}_j) \\ \text{where} \quad \psi_{ij} &= \frac{\Phi(|\vec{x}_i - \vec{y}_j|, K)}{\sum_k \Phi(|\vec{x}_i - \vec{y}_k|, K)} \end{aligned} \tag{5.1}$$

Here $\vec{x}_i$ is the position of the i^{th} city, and $\Phi(d, K)$ (not to be confused with the potential ϕ used elsewhere) is a positive bounded decreasing function of d that approaches zero for $d > K$. An important factor is the normalisation by ψ_{ij} of the total influence of each city. Provided there are a large number of loop points, this guarantees that each city will gain a position on the tour. The second kind of force pulls each loop point towards its immediate neighbours in the loop in proportion to their separation, hence it is analogous to tension:

$$\vec{F}_j^{tension} = K(\vec{y}_{j+1} - 2\vec{y}_j + \vec{y}_{j-1}) \tag{5.2}$$

This force acts to minimise the total length of the loop.

At every iteration each loop point j moves a small distance along the resultant of these two forces:

$$\Delta\vec{y}_j = \alpha\vec{F}_j^{city} + \beta\vec{F}_j^{tension} \tag{5.3}$$

If K is held constant, after many iterations the points would assume a stable positions where the forces balance each other out. Instead however, K is slowly reduced towards zero. This means that initially the tension force dominates but gradually these forces weaken. At the same time the loop-to-city forces become stronger at progressively shorter ranges about each city. Eventually the loop relaxes into a valid tour as the tension and the ranges of the city forces both tend to zero. Further, if $\Phi(d, K)$ is taken as the gaussian function

$$\Phi(d, K) = \exp(-d^2/2K^2) \tag{5.4}$$

an energy function can be defined:

$$E = -\alpha K \sum_i \ln \sum_j \Phi(|\vec{x}_i - \vec{y}_j|, K) \quad + \quad \beta \sum_j |\vec{y}_{j+1} - \vec{y}_j|^2 \tag{5.5}$$

For a given value of K, the motion of the loop points perform gradient descent of E, and because E is bounded below a local minimum will eventually be reached. For large numbers of loop points the first term ensures a valid tour, since it can only remain bounded as K tends to zero if each city i has a loop point j such that $|\vec{x}_i - \vec{y}_j|$ tends to zero. Where such a valid tour is defined, the global minimum of the second term corresponds to the optimum tour.

Several variations of the basic *EN* algorithm have been suggested (Burr 1988; Simic 1990). The method has also been applied to a perplexing biological problem, that of explaining the development of ocular dominance stripes in the vertebrate visual system (Goodhill & Willshaw 1990). [Aua 1990] has extended the method to tackle the minimal length circuit problem (Steiner Problem), which is also *NP*-Complete, with good results. Very similar methods to the *EN* for the TSP which are based instead on what are called "self-organising feature maps" (Kohonen 1982) have also been proposed (Angéniol *et al.* 1988; Fort 1988; Hueter 1988; Ritter & Schulten 1988). In these, typically a city is chosen and the loop point closest to it identified. This point is moved towards the city, and its neighbours do likewise but by an amount which decreases with distance along the loop. These methods tend to be faster to implement than *EN*, but give slightly longer tours and are not as amenable to analysis.

[Durbin *et al.* 1989] gave a probabilistic interpretation of the *EN* algorithm, and found conditions under which each city becomes matched by only one loop point. They also analysed the behaviour of the energy function as K is changed, showing that above a critical value of K a single minimum exists, and finding bifurcation points of the energy function as K is reduced. [Simmen 1990] has qualified their result on the matching of loop points to cities, and shown that as K tends to zero there exist local minima in which some cities are not visited. Constraints were derived on the choice of the ratio β/α which guarantee this will not occur. [Simic 1990] showed that both the *EN* and the method of Hopfield and Tank have an underlying statistical mechanics foundation, and that the difference between them arises because of the more direct way that the *EN* imposes constraints. [Yuille 1990] has also derived both algorithms from a statistical mechanics framework.

In sharp contrast to Hopfield and Tank's method, the *EN* method can always produce valid tours. Further, the tour lengths obtained are very close to those obtained by Simulated Annealing and the best of repeated trials of 3-opt. Being derived from the Tea Trade model for the formation of mappings in the brain, the *EN* method has the same advantages (*ie.* continuous variables, parallelism and simplicity) as neural network methods. However, the requirement that the cities be embedded in a geometric space is a strong one, since a slight change to any of the straight-line inter-city distances destroys this metric, making the *EN* method inapplicable. The removal of this restriction was the major motivation for the approach to be described in the next section.

5.3 The TSP and layered neural networks

An alternative approach is to separate the loop from the cities by representing them as separate sets of units in a neural network, and then attempt to form a neighbourhood-preserving mapping between them through modifiable weights[1]. Besides the advantages of neural network methods in general, this approach has a number of desirable properties:

- There is no external "coordinate system" required, so the mappings need not be between Euclidean vector spaces.
- By varying the number of weights in the network, the total number of degrees of freedom present can be easily manipulated.
- There are algorithms for developing the mapping in a biologically realistic way (for instance the Neural Activity model, discussed in section 5.5).

The proposal is therefore to consider the TSP in terms of developing a topographic mapping between one layer of units which represent the tour positions, and another layer which represent the cities. This architecture and its interpretation as a solution of the TSP are now introduced.

[1] I am grateful to David Willshaw for suggesting this approach.

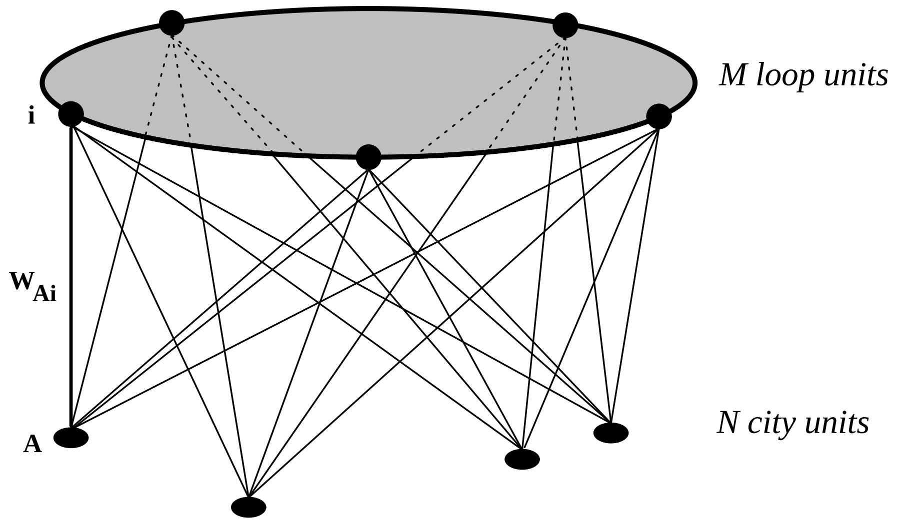

Figure 5.1: Architecture for TSP as a topographic mapping problem between two layers of units. The algorithms work by altering the variable weights shown. The possible fixed feedback connections between cities are not shown.

5.3.1 The network architecture.

The algorithms to be described are methods for altering the weights in the architecture shown in figure 5.1. There are M units in one layer, arranged in a loop. The N units in the other layer represent the cities. The two layers are fully connected, with the initial weights chosen at random from a narrow gaussian distribution centered on $1/M$, with no negative weights. All loop units are indexed *modulo* M and in lower case, with cities in upper case. For instance, the weight between loop unit i and city A is written $W_{A,i}$. It will often be useful to refer to the *selectivity* of a city unit, which is at a maximum when the unit in question is connected only to units in the other layer which themselves have no non-zero weights to other cities, and minimum when its connections are evenly distributed.

5.3.2 Displaying the current state and defining a tour.

A tour is clearly defined when each city has non-zero weights to one group of adjacent loop units only, but the ordering is not so obvious when loop points have weights to more than one city. To define a tour, a representation of the weights matrix is used in which the ordering of cities is apparent. This is shown in figure 5.2, in which *A,B,C,D* represent cities and i represents one of the loop units. For

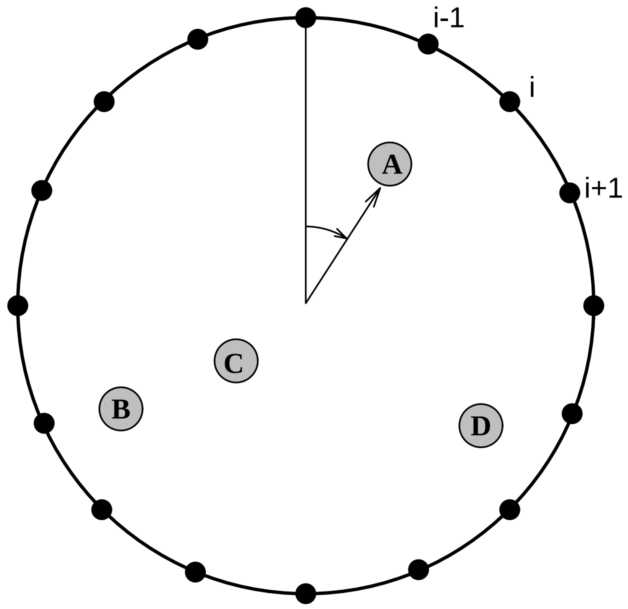

Figure 5.2: Display.

any given city, the strength of the weight to any loop unit is represented by a vector from the center of the circle pointing towards the loop point and of magnitude proportional to the weight. The position of the city is then given by the sum of these vectors taken over all loop points. The scale is such that the radius of the displayed loop corresponds to the maximum possible weight. The distance of A from the center is then a measure of its selectivity for one region of the loop, while its angular coordinate "points to" its position in the tour. Initially all cities appear in the centre. As they become selective they move outwards, eventually touching the loop at maximum selectivity. Although information about the distances between cities is not indicated explicitly, this representation has the appeal that the ordering is purely dependent on the angular coordinate while selectivity appears as the radial coordinate. Hence a tour is definable at any stage but its significance is clear from the radial coordinates of the cities.

Where the distance matrix corresponds to cities lying in a plane their actual positions can be shown, along with a projection of the loop onto the plane formed similarly as a summation of *2D* vectors. That is, if city A is at position $\vec{x}_A$ in the plane, the j^{th} loop point is shown positioned at

$$\vec{v}_j = \frac{\sum_A W_{Aj} \;\; \vec{x}_A}{\sum_B W_{Bj}}$$

This can be deceptive, since for instance a city unit with equal weights to only two cities will be displayed half way between them, and may therefore appear very close to a third city even though its weight to this city is negligible. However at maximum selectivity the loop so displayed follows the tour as defined above, and well before this stage the broad form of the tour is clear. This makes it useful as a development tool.

This representation of the weights is now used in conjunction with two algorithms which develop the mapping appropriate to the TSP.

5.4 A Layered Elastic Net method

The *EN* method is a successful approach to solving the TSP in the plane which is based on the idea of forming a topographic mapping. The following is an attempt to apply a similar method to learn the weights in a layered neural network architecture as described above. First, quantities are defined which are analogous to the distances between loop points and cities, and the distances between adjacent loop points of the *EN*. These are used to motivate a learning rule for forming a neighbourhood-preserving mapping between the layers. Some approximations are necessarily made to keep the method simple, but it is found that the *EN* approach can be applied to a layer of weights with success.

5.4.1 Pseudo-distance measures

In the elastic net, loop points are influenced by two types of attractive force, the first between loop and cities, the second between adjacent loop points. To apply the *EN* idea to a layer of weights we require measures which I call *pseudo-distances* between

- loop point j and city X, called S_{Xj}
- j and $j+1$, called $L_{j,j+1}$

for all X, j in terms of

- weights W

- city-to-city distances D

Suppose we take the weight W_{Xj} to represent the proximity of j to X. An essential property is that it should be impossible for j to be in two places at once, or more generally, moving j towards X must have direct consequences on j's proximity to other cities. This will be enforced by the constraint

$$\sum_Y W_{Yj} = 1$$

for all loop points j. A weight of $W_{Xj} = 1$ may then be interpreted as meaning point j is exactly *at* city X. The normalisation means that j must always move away from some cities in order to get closer to others. Now consider

$$S_{Xj} = \sum_Y W_{Yj} D_{XY} \tag{5.6}$$

as the pseudo-distance between j and X. The final state of the network should be one in which each loop point is either at a city or directly between two cities. If j is at X, its pseudo-distance to other cities should be just the actual inter-city distance. This is true of S as defined above, *ie.*

$$S_{Aj} \to D_{XA} \quad \text{as} \quad W_{Xj} \to 1$$

Alternatively if j has weights to only two cities A and B then

$$\begin{aligned} S_{Aj} + S_{Bj} &= D_{AB}(W_{Aj} + W_{Bj}) \\ &= D_{AB} \end{aligned}$$

so j is indeed directly between the two cities. Notice that increasing a single weight, say W_{Cj}, means we think of j as moving closer to C, but does not on its own alter S_{Cj}. Normalisation then is crucial. It is effectively *competition* between the S's as to which ones can decrease. Consider three cities A, B and C. In the non-Euclidean case there is no direct sense in which moving from B towards C necessarily implies moving towards or away from A, for instance. Recall that in the *EN* method this problem doesn't arise, since a loop point's movement in the plane *implicitly* changes all its distances to cities (that is, positions are fundamental, while the distances are inferred). Instead, here there are no positions; the competition replaces the distance changes implicit in the Euclidean problem.

Using S as defined above, a "force" on loop units can be defined in analogy with that of the *EN*. The component of this force on loop unit j, pulling it towards city X is then

$$\begin{aligned} F_{Xj} &= \Psi_{Xj} S_{Xj} \\ \text{where} \quad \Psi_{Xj} &= \frac{\Phi_{Xj}}{\sum_k \Phi_{Xk}} \end{aligned}$$

with $\Phi_{Xj} = \Phi(S_{Xj}, K)$, a monotonically decreasing function of pseudo-distance S_{Xj}. In this case, as in the *EN* method, the chosen function is the gaussian:

$$\Phi(S, K) = \exp(-S^2/2K^2)$$

Some measure of pseudo-distance between points on the loop is also needed. Hopfield and Tank used the following 'tour length' term in their energy function, here rewritten in terms of dynamical variables W:

$$\begin{aligned} E_{tourlength} &= \frac{\beta}{2}\sum_i \sum_{X,Y} W_{Xi}(W_{Y,i+1} + W_{Y,i-1})D_{XY} \\ &= \beta \sum_i \sum_{X,Y} W_{X,i}W_{Y,i+1}D_{XY} \end{aligned}$$

In the desired limit situation wherein W forms a permutation matrix this term is just β times the length of the tour, so it can be written as

$$\begin{aligned} & E_{tourlength} && = \beta \sum_i L_{i,i+1} \\ \text{with} \quad & L_{i,j} && = \sum_{X,Y} W_{Xi}W_{Yj}D_{XY} \end{aligned}$$

For this reason it is tempting to identify $L_{i,j}$ with the pseudo-distance between the two points since if $W_{Xi} = 1$ and $W_{Yj} = 1$ then $L_{i,j} = D_{XY}$, which is sensible given that such weights are taken to mean that i is at X and j is at Y. However it should also be true that

$$L_{i,i} = 0$$

which is never obeyed by the above definition of $L_{i,j}$ except in the limit. If instead we define

$$L_{i,j} = \frac{1}{2}\sum_{X,Y}(W_{Xi} - W_{Xj})(W_{Yj} - W_{Yi})D_{XY} \tag{5.7}$$

then the limit distance is unchanged but $L_{i,i}$ is always zero, so this is now adopted as the pseudo-distance between i and j.

5.4.2 Derivation of a rule for changing the weights.

In the *EN* method there are two terms, one which enforces the constraints of the problem and one which favours a short loop. One way to produce a weight change rule is simply to differentiate the *EN* energy function using the pseudo-distances in place of the original Euclidean distances. This yields the following rule:

$$\Delta W_{Xj} = -\alpha \sum_Y \Psi_{Yj} S_{Yj} D_{XY} \quad - \quad K\beta[L_{j,j+1}(S_{X,j+1} - S_{X,j}) + L_{j,j-1}(S_{X,j-1} - S_{X,j})]$$

This rule has no intuitive appeal, and also takes no account of the crucial normalisation. Instead, in the following I derive a simple rule that retains the basic character of the *EN*.

The constraint term.
Suppose that the unnormalised weight change caused by the "force" F_{Xj} is simply a positive change (say δ) in the weight W_{Xj}. The normalisation reduces the weights between j and all other cities $Y \neq X$:

$$\begin{aligned} W_{Yj}^{new} &= \frac{W_{Yj}}{1+\delta} \\ \text{or} \quad \Delta W_{Yj} &= \frac{-\delta\, W_{Yj}}{1+\delta} \end{aligned}$$

The resulting change to pseudo-distance S_{Xj} is

$$\begin{aligned} \Delta S_{Xj} &= \sum_{Y} \Delta W_{Yj} D_{XY} \\ &= \frac{-\delta}{1+\delta} S_{Xj} \end{aligned}$$

In the *EN* method, the loop point j moves under the force F_{Xj} such that, were there no other forces present, the distance between j and X decreases in proportion to the magnitude of the force. Therefore a desirable property of the weight change is that it causes S to change by

$$\Delta S_{Xj} = -\alpha F_{Xj}$$

Equating the two expressions for ΔS_{Xj} readily gives the appropriate weight change as

$$\Delta W_{Xj} \quad = \quad \delta \quad = \quad \frac{1}{\frac{1}{\alpha \Psi_{Xj}} - 1}$$

Since both α and Ψ are much less than 1 this may be approximated by

$$\Delta W_{Xj} = \alpha \Psi_{Xj}$$

Does this number Ψ_{Xj} bear any relationship to the usual quantities calculated by neural networks? Suppose that all of the cities interact via feedback connections, with the connection between cities X and Y having an associated weight ω_{XY}, of magnitude equal to their separation D_{XY}. In that case, if the j^{th} loop unit is activated at a given time (and then turned off), then S_{Xj} is the sum of the input to city X (*ie.* its potential) one time step later. Φ_{Xj} is a monotonically decreasing function of this potential (in this case it is a gaussian). Ψ_{Xj} is simply the ratio of this to the sum of Φ_{Xi} over *all* inputs, i, or equivalently it is Ψ_{Xj} relative to (M times) the mean of Ψ_X over all inputs. Hence the weight change ΔW_{Xj} is a simple function of the input to X due to j.

The tour length term.
Unlike the method of Hopfield and Tank, the *EN* energy contains a sum of *squared* inter-loop distances, *ie:*

$$E_{tourlength} = \frac{\beta}{2}\sum_{j} L^2_{j,j+1}$$

Each term is the energy of a spring of extension $L_{j,j+1}$ and spring constant β, hence the associated force is a tension. The tension force means that (in the absence of any other forces), the distance $L_{j,j+1}$ should decrease in proportion to itself, or

$$\Delta L_{j,j+1} \propto -L_{j,j+1}$$

Suppose the two weights vectors $\vec{W_j}$ and $\vec{W_{j+1}}$ are simply moved towards one another by an amount proportional to *their* difference, *ie.*

$$\begin{aligned} \Delta W_{X,j} &= \tfrac{\beta}{2}(W_{X,j+1} - W_{X,j}) \\ \text{and} \quad \Delta W_{X,j+1} &= \tfrac{\beta}{2}(W_{X,j} - W_{X,j+1}) \end{aligned}$$

for all X. The resulting change in $L_{j,j+1}$ is found to be

$$\begin{aligned} \Delta L_{j,j+1} &= -\beta(1-\beta)L_{j,j+1} \\ &= -\beta' L_{j,j+1} \end{aligned}$$

So this simple way of altering weights does result in a tension-like effect on $L_{j,j+1}$. Since loop unit j is pulled towards both $j-1$ and $j+1$, the net change is actually

$$\Delta W_{Xj} = \frac{\beta}{2}(W_{X,j-1} + W_{X,j+1} - 2W_{X,j})$$

5.4.3 The LEN algorithm.

The layered elastic net (*LEN*) develops a mapping which is a solution to the TSP by changing its weights in the following way.

For each iteration,

1. The weights are altered for every city X and loop point j by

$$\Delta W_{Xj} = \alpha\Psi_{Xj}(K) \;\; + \;\; \frac{\beta K}{2}(W_{X,j-1} + W_{X,j+1} - 2W_{X,j}) \qquad (5.8)$$

2. The sum of all j's weights is restored to unity by rescaling:

$$\begin{aligned} \vec{W}_j^{\,scaled} &= \lambda_j \vec{W}_j \\ \text{where} \qquad \lambda_j &= \frac{1}{1+\sum_X \Delta W_{Xj}} \end{aligned} \tag{5.9}$$

The parameter K is gradually reduced to zero over many iterations, as in *EN*.

5.4.4 An example.

The example shown here is one of the 50 city problems tested by [Durbin & Willshaw 1987] (their city set 'b'). There are 125 units in the loop layer. The parameters used were $\alpha = 0.15$ and $\beta = 2.0$. K starts at 0.2, remaining at this value for 1000 iterations. K is then reduced linearly to 0.002 over a further 1000 iterations. Figure 5.3 shows the projection onto the loop of this mapping after 500, 1000, 1500 and 2000 iterations. The solid line indicates the ordering of the cities given by best of 100 trials of 3-opt. Hence large differences between 3-opt and the tour derived from the mapping after 2000 iterations would appear as lines cutting across the circle. In this case a tour is found which is only 1 % longer than the best 3-opt tour, and the two tours are very similar in form. This is confirmed by projecting the loop points back onto the plane in which the cities lie, as shown in figure 5.4.

5.5 An adaptation of the Neural Activity model to the TSP.

5.5.1 Background: the modelling of biological mappings.

Topographic mappings between groups of nerve cells occur frequently in the nervous systems of both invertebrates and vertebrates; indeed they may be the most striking accessible evidence of order in the structure of these systems. The connections from cochlear membrane to auditory cortex in mammals form a one dimensional mapping, but most models have concentrated on the two dimensional

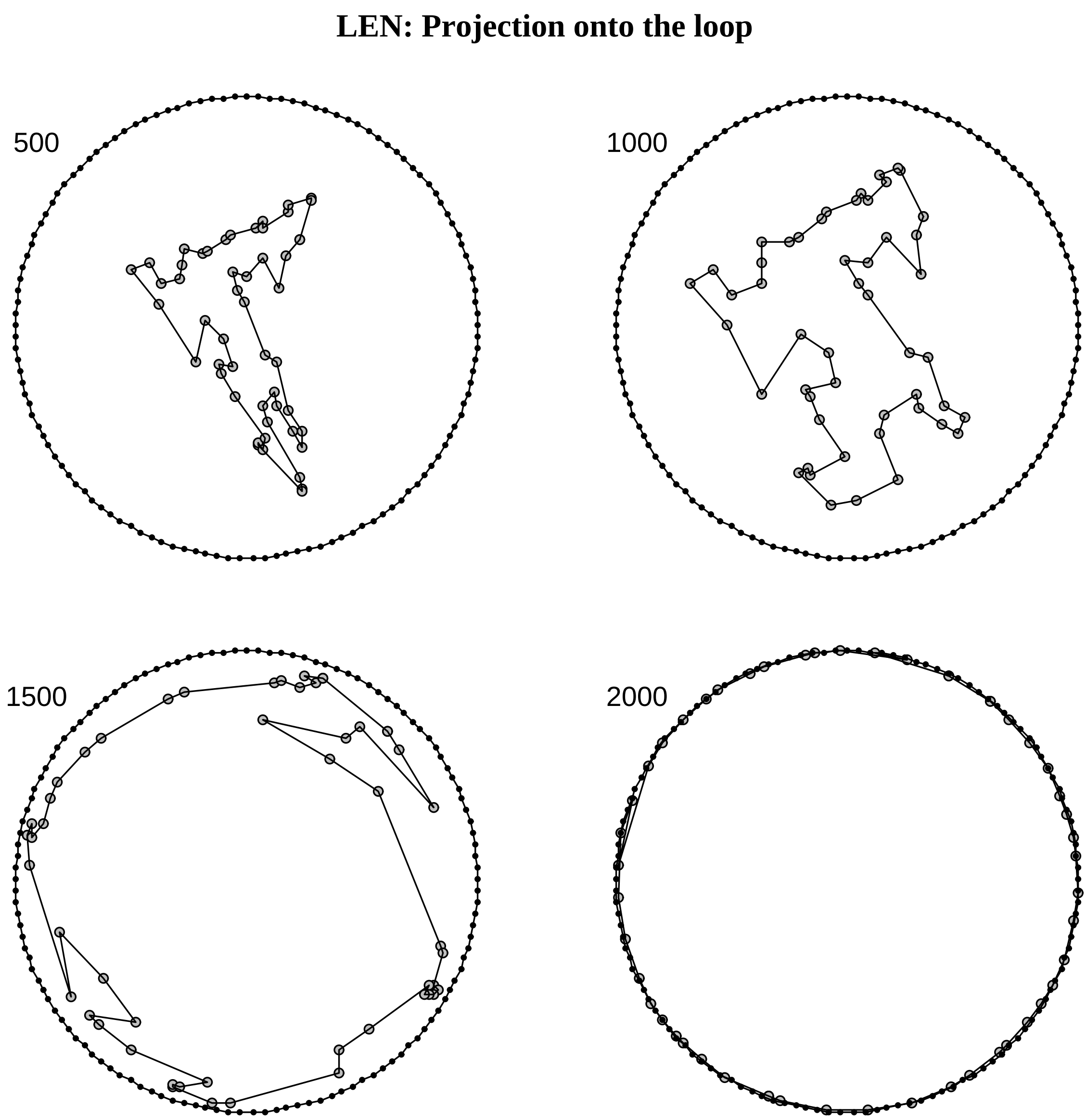

Figure 5.3: Example of *LEN* on a 50 city Euclidean problem, shown after 500, 1000, 1500 and 2000 iterations. The small circles dispersed inside each picture are the cities projected as described earlier. Their movement outwards indicates that each is increasing its selectivity for a particular portion of the loop, and their angular coordinate determines their ordering in the tour. For the purposes of comparison, the straight lines joining them denote the ordering of cities in the best tour obtained by 3-opt. The degree of folding in this line indicates how different the two tours are.

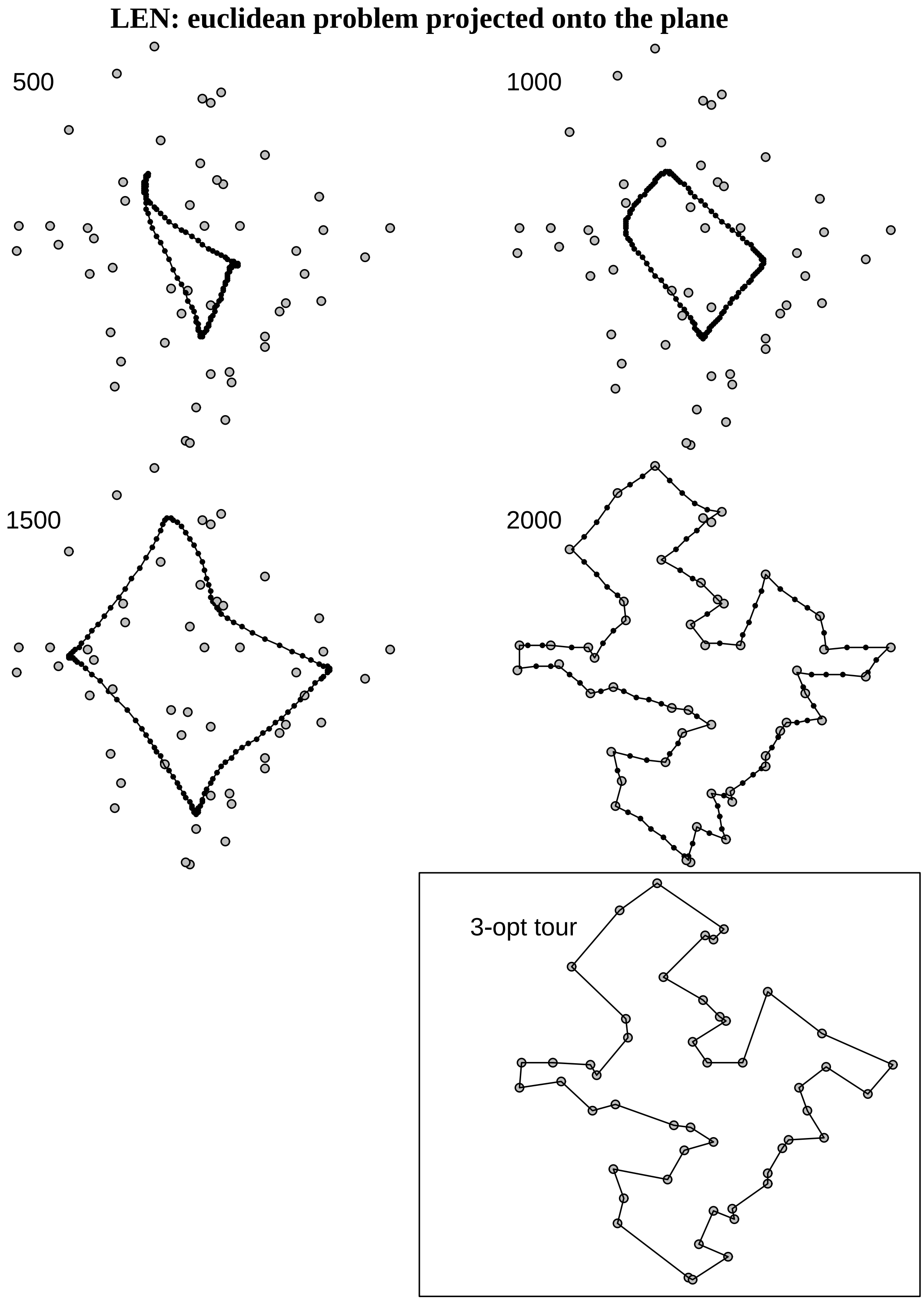

Figure 5.4: Example of *LEN* on a Euclidean problem. The loop is shown projected into the plane in which the cities lie. For comparison, the best 3-opt tour obtained is also shown. It is interesting to note the similarity between the shape of the final tour and the projection onto the loop after 1000 iterations in the previous figure.

mapping from retina to optic tectum in vertebrates such as *Xenopus* (Gaze 1970), or the connections from the retina to the lateral geniculate nucleus and thence to striate cortex in higher animals, although these are not so accessible to experiment and are more complex. These are shown schematically in figure 5.5. The mappings are demonstrable by naturally or artificially exciting a small region of the retina, and then measuring the pattern of neural activity in the tectum. There is a wealth of literature on the nature of these connections, their formation in developing animals under various regimes, and their reorganisation following surgical interference. The basic experimental finding is that the retinotectal connections form a topographic mapping; any pattern presented to the retina essentially reappears as a pattern of activation across the tectum. The two regions are well separated from one another, and the fibres connecting them are not necessarily topographically ordered *en route*. In fact the fibres seem to re-organise themselves into their correct positions where they arrive at the tectum. Substantial research effort is expended on discovering the rules they use to do this during development.

5.5.2 The Neural Activity model.

[Willshaw & von der Malsburg 1975] propose a model to account for the formation of such topographic mappings. Each axon of a pre-synaptic cell conducts a search in the post-synaptic region for the cells which its neighbours project to, and so the appropriate mapping develops. In the Neural Activity (*NA*) model this search is conducted purely on the basis of correlated neural activity.

It is obvious that the notion of a neighbourhood is logically dependent on there being some metric present in the layer in which it resides. In the model this is provided by "on-centre-off-surround" lateral feedback. Feedback of this type is common in many regions of the brain, including the retina. Suppose that the two layers are fully connected to one another by synapses with initially random but positive synaptic efficacy (*ie.* weight). A tectal cell's *potential* (or membrane depolarisation) is taken to be the sum of all its inputs (including those from other tectal cells), each weighted by its synaptic efficacy. Willshaw and von der Malsburg assumed that the cell's actual *activity* increased linearly along with its potential, provided this was above some threshold. Below the threshold there would be no activity.

Suppose some random activity arises in the retina. The effect of the feedback in the retina itself is to accentuate simultaneous activity in nearby cells. Therefore neighbouring cells will tend to have correlated activity, cells at intermediate distances will tend to have anticorrelated activities, and distant cells will remain uncorrelated. The overall effect is that retinal activity tends to become concen-

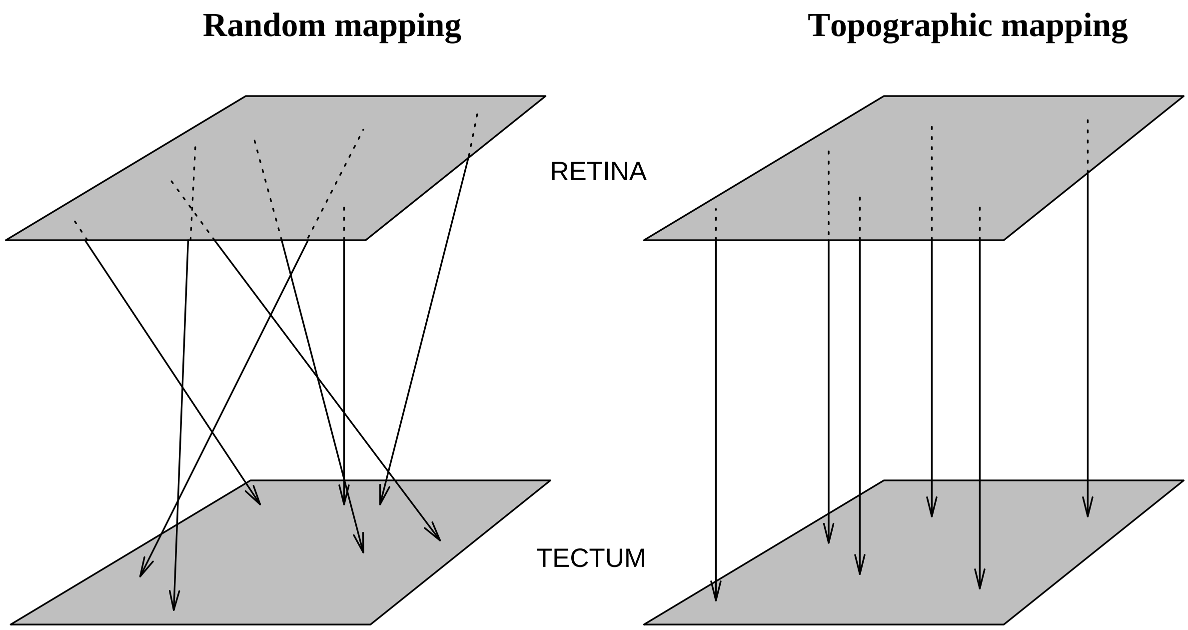

Figure 5.5: Retino-tectal projection.

trated in patches. Given initially random connections, in the absence of feedback within the tectum the resultant activity amongst tectal cells would be expected to be randomly distributed. However the effect of feedback in the tectum this time is to accentuate activity only when the active cells are neighbours, so tectal activity also tends to become concentrated in patches.

Altogether then, there is a tendency for activity to occur in small patches of both the retina and tectum, and this is precisely the pattern of connection which needs to be enhanced in order to produce a topographic mapping. The obvious way to increase this link is the simple Hebbian rule derived from the postulate (Hebb 1949) that simultaneous activation in both the pre-synaptic and post-synaptic cells should lead to an increase in synaptic efficacy.

Clearly in this case the weights will always increase. To keep the weights from increasing without bound while avoiding simply setting a ceiling level to which all weights will saturate, there must be some means of reducing weights as well. This introduces the requirement for some kind of *competition* among weights, determining which of them might increase, if we are to retain the Hebbian principle. Willshaw and von der Malsburg achieved this by postulating that each tectal cell attempts to keep the total of its incoming weights at a constant value; in that case a Hebbian increase to the weight from an active retinal cell implies a decrease of the weights from inactive cells.

5.5.3 Orientation and part-maps

There are two problems with the proposal as it stands.

Firstly, there is nothing so far to give the mapping its overall orientation. Since *NA* was put forward as the possible principle by which actual mappings arose, and those mappings do have a definite orientation, it was important that this come out of the model in a plausible way. Secondly, there is the problem that several clumps of activity in the tectum may well occur at one time, even if there were only a single retinal region active. This would give rise to several regions in the mapping which are correctly ordered in themselves, but which don't fit together properly. Willshaw and von der Malsburg overcame these by

- postulating a small number of 'marker cells' in either layer whose position was genetically predetermined. Each retinal marker cell had a specific tectal counterpart with some means of recognition (for example a chemical affinity) such that they could form a strong synaptic connection prior to development of the full mapping.
- introducing a threshold into the weight change, similar to that applied to potentials. Below this threshold which there is no Hebbian weight change.

The markers give the mapping its orientation, and the threshold is set so that isolated activity elsewhere is unlikely to alter the weights. In combination, these produce a region of nucleation around the markers since initially the activity is not high enough to cause change elsewhere. Development then expands over the entire surface as one consistent map.

This model produces topographically correct maps. However on its own it does not account for all the experimental findings concerning retinotectal mappings. This led Willshaw and von der Malsburg to propose a different method which they called the Tea Trade Model (von der Malsburg & Willshaw 1977; Willshaw & von der Malsburg 1979) which accorded better with the experimental data. In this proposal, a small number of molecule types diffuse throughout each layer. These in effect can act as coordinates: a cell 'knows where it is' by the concentrations of these molecules in its vicinity. The Tea Trade Model later formed the basis of the Elastic Net method for the TSP.

5.5.4 Details of the original implementation.

In the NA model the variable weights (W) from retina to tectum are initially random, apart from a small number of large weights serving as markers. The fixed weights (ω) between tectal cells are positive between cells which are immediate neighbours, negative between cells in the medium range, and zero between distant cells. An input (*ie.* retinal) pattern $\vec{\xi}$ consisting of a pair of adjacent units is applied, and each output (*ie.* tectal) unit's activation $\phi^\star_A$ is evaluated by numerically solving N coupled equations (one for each tectal cell A):

$$\begin{aligned} \frac{d}{dt}\phi_A(t) &= \sum_i W_{Ai}\xi_i + \sum_B \omega_{AB}\phi^\star_B(t) - \alpha\phi_A(t) \\ \text{where} \qquad \phi^\star &= \begin{cases} \phi - \theta & \text{if } \phi > \theta \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{5.10}$$

These equations are iterated until some criterion of stability is met. The first term is the direct effect of the input activation and the second is the effect of feedback from other output units.

The Hebbian weight change to be made is

$$\Delta W_{Ai} = \eta\phi^\star_A\xi_i$$

which is itself thresholded so that only the largest changes take effect:

$$\Delta W'_{Ai} = \begin{cases} \Delta W_{Ai} - \epsilon & \text{if } \Delta W_{Ai} > \epsilon \\ 0 & \text{otherwise} \end{cases}$$

Finally the condition

$$\sum_i W_{Ai} = S$$

for each output unit A is restored by rescaling all its weights by the appropriate amount. These steps are followed for many presentations of many input patterns, and a topographic mapping gradually forms.

5.5.5 Relationship to competitive learning.

The Neural Activity model is a direct descendant of the self-organising model due to [von der Malsburg 1973], and was itself a precursor of a currently popular unsupervised learning paradigm known as *competitive learning* used in many models of self-organising systems (see for example [Rumelhart & Zipser 1986] and [Kohonen 1982]). Suppose that the weight changes for a particular output unit are ΔW_j for all inputs j prior to normalising. All the weights are then multiplied by a single factor to keep their sum constant, giving

$$W_j^{new} = \frac{W_j + \Delta W_j}{1 + \sum_k \Delta W_k}$$

Provided the changes are small at each step ($\eta \ll 1$),

$$W_j^{new} \simeq (W_j + \Delta W_j)\left(1 - \sum_k \Delta W_k\right)$$

and keeping linear terms in ΔW, the overall change denoted by $\tilde{\Delta} W$ is

$$\tilde{\Delta} W_j = \Delta W_j - W_j \sum_k \Delta W_k$$

This obeys the sum rule exactly. If the normalisation occurs after each input pattern as in the original *NA* model, then ΔW_j is just $\eta \phi^\star \xi_j$, so the net change is

$$\tilde{\Delta} W_j = \eta\,(\xi_j - m W_j)\,\phi^\star$$

This specifies a movement towards the input pattern by an amount proportional to the unit's activation $\phi^\star$. There are two senses in which this is "competitive" learning. The first is that the weights compete with one another to increase, because the normalisation drives weights from inactive inputs down. The second is the competition between output cells themselves. Often in these models this competition takes the form of resetting some fixed number of the largest potentials to 1 and all others to zero (referred to as "k-winners take all" networks), without any topological information (as for example in [Rumelhart & Zipser 1986]). This is akin to having inhibitory feedback connections of uniform strength between all units in the output layer. In other models, the largest potential is reset to 1 together with some of its neighbours in the output layer, and all others are set to 0 (Kohonen 1982). This is an idealisation of the effect of short-range lateral feedback in the *NA* model.

5.5.6 Adaptation to the TSP.

There are two ways to implement the Neural Activity model as a TSP network: the loop can be treated as either the input (retinal) or the output (tectal) layer.

Firstly, all the cities should be able to influence one another to a degree dependent on their distance, whereas the loop cells need only interact with those adjacent to them. Secondly in the *EN* method it was seen that a normalisation giving each city equal "influence" is important. In the *NA* model the local feedback between retinal cells was not modelled explicitly, but instead each pattern of activity consisted simply of two adjacent retinal cells having activity 1 and all others activity 0. Also each tectal cell has weights which are normalised to the same amount, making all of them equally "strong". Together these make the presynaptic layer the most natural choice for the loop. Thus the loop topology enters via presentation of input patterns consisting of contiguous sections of the loop, and inter-city distances are encoded as feedback between the city units in the output layer.

Tours have also been obtained by the other option, in which the cities act as input. This is done by normalising the weights from each city unit and varying the magnitude of the input activation of two cities as a function of their separation. The tours are unimpressive and the method shows no particular advantages, and so will not be considered further.

5.5.7 Simplifying the model.

This model can be considerably simplified for the purposes of the TSP.

Firstly, it is not essential to solve equation 5.10 iteratively to stability. Instead, a single iteration of feedback amongst the cities suffices. This does not adversely affect the tour lengths produced, and is both faster to implement and easier to analyse.

Secondly the threshold θ ensures that the activation of cities, and hence the Hebbian weight change before normalisation, is never negative. This can be set to its minimum possible value of zero. Likewise, for now we assume that ϵ can also be set to zero, in which case it is made redundant by θ.

Thirdly the particular form of the feedback chosen in the original model was influenced by the known biological fact of short-range excitation and medium-range inhibition. This is not appropriate for the TSP, since it is likely to be important for cities at large separations to influence one another directly. Instead, in lieu of a better motivation, the simplest function relating influence to distance is just linear inhibition:

$$\omega_{AB} = -\lambda D_{AB}$$

where D_{AB} is the known distance between cities A and B.

Finally the input patterns need not consist of activity in only two adjacent units, and it is not necessary that the presentation of patterns be in a random sequence with normalisation of the weights after every pattern. Instead all the patterns containing some number (say m) of adjacent units can be presented and the weight changes accumulated. These changes can then be made and the weights renormalised once all the M such patterns have been presented. Moreover this increases the potential parallelism of the method.

The above points lead to a simplified algorithm. Assume the loop cells are labelled in a cyclic fashion (0 to $M-1$) so that k maps to k modulo M (if $k \geq M$). The m cells that are 'on' (starting at cell i) have activity $\xi = 1$ and all others are 'off' ($\xi = 0$). The algorithm for using the NA model to solve the TSP is as follows:

For each iteration:

1. Start with all weight changes ΔW at zero.
2. Complete the following steps for each of the M input patterns of width m:

   Step (a): Calculate the sum of the weighted inputs into every city X
   $$\phi_X^{raw} = \sum_j W_{Xj}\xi_j$$

   Step (b): Find the potential of every city X by including feedback from the other cities
   $$\phi_X = \phi_X^{raw} + \sum_Y \omega_{XY}\phi_Y^{raw}$$

   Step (c): Eliminate negative potentials to give the "activity" of each city X
   $$\phi_X^\star = \begin{cases} \phi_X & \text{if } \phi_X > 0 \\ 0 & \text{otherwise} \end{cases}$$

   Step (d): Accumulate the weight change for each X, j
   $$\Delta W_{Xj} = \Delta W_{Xj} + \eta\phi_X^\star\xi_j$$

3. Add the change ΔW_{Xj} into each weight W_{Xj}
4. Rescale the weights into each city X by the amount
   $$\frac{1}{1 + \sum_i \Delta W_{Xi}}$$
   to restore the condition $\sum_j W_{Xj} = 1$.

Together, these steps constitute one iteration.

λ, the constant of proportionality of the the linear inhibition, is set such that the lowest potential that could be generated by uniform weights is small but positive. In that case, $\phi^{raw} = \frac{m}{M}$, so $\phi_X = \frac{m}{M}(1 + \sum_Y \omega_{XY})$ for each city X. Therefore for positive potentials,

$$-1 < \sum_Y \omega_{XY} \leq 0 \tag{5.11}$$

Thus we require $0 \leq \lambda \sum_Y D_{XY} \leq 1$, so λ is set slightly less than the maximum over all X of $1/\sum_Y D_{XY}$

5.5.8 Two problems.

Simulations using the algorithm were carried out using $M = N$ and $m = 2$, as in the original model. Two problems become apparent, which are essentially the appearance of part-maps and the 'fuzziness' in the mappings.

The problem of part-maps.
In the *NA* model the first few units to develop selectivity determine the broad form of the mapping, because increased selectivity itself encourages self-organisation around these units. However this means that several small but locally ordered regions of the mapping form and are stable, even though they don't fit together with other such regions. Because the initial conditions are random, this leads to major errors at this early stage which cannot be corrected later. In the application to retinotectal mappings, these result in misaligned part-maps. In the TSP, part-maps are manifested as segments of tours which are short in their own right but which cannot be connected together to form a tour without using long paths. In the case of cities in the plane, a crossed tour is a good example of a part-map: it is known that a tour which crosses itself can always be shortened by uncrossing the offending links, and this operation amounts to merely reversing the order of a large segment of the tour.

The use of a 'marker' (*ie.* a bias strongly associating one loop point with a particular city), as in the original Neural Activity model, might seem a good idea since it removes the $2N$-fold degeneracy of possible tours (N start points and 2 directions). This degeneracy was noted by [Wilson & Pawley 1988] as being problematic for algorithms of this type. However in this case it is inappropriate because it gives a certain (presumably arbitrary) city a special status; development of the mapping would spread from the 'marked' city, making the method akin to the nearest-neighbour algorithm, which is known to be very ineffective (see Lawler *et al.* 1985).

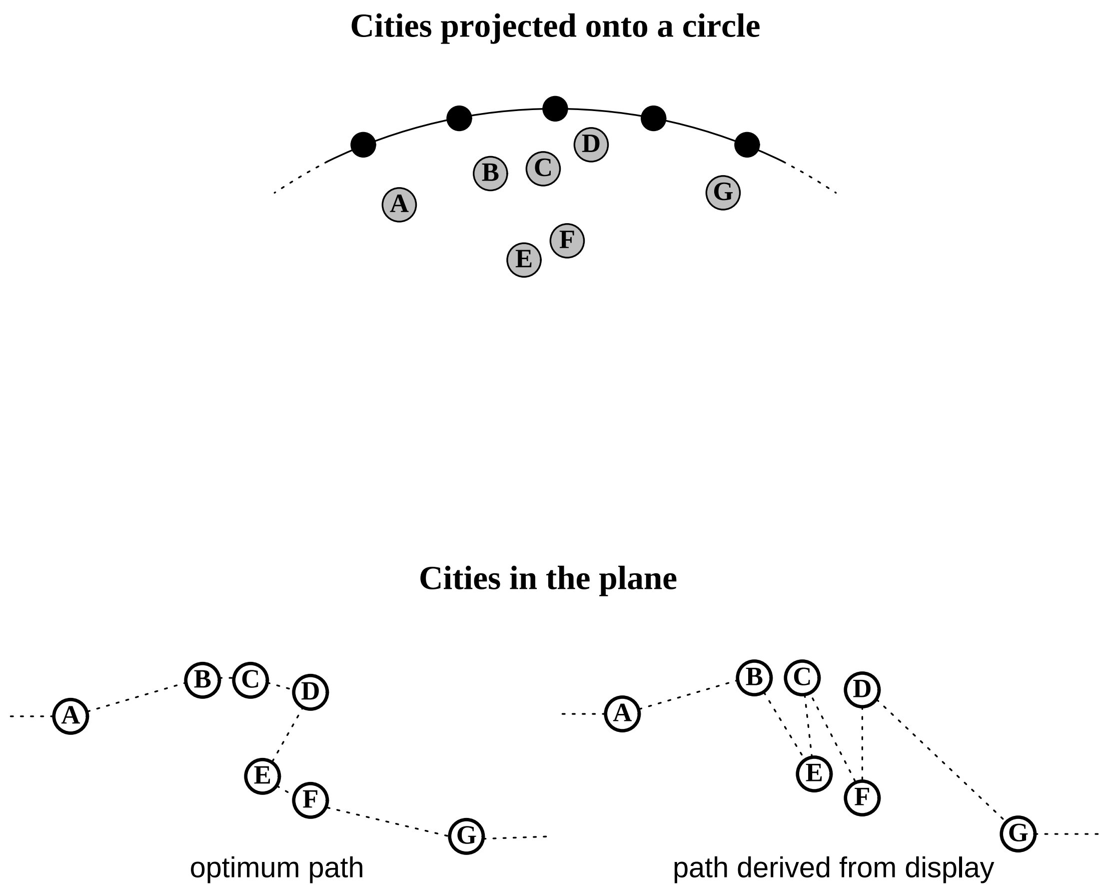

Figure 5.6: Part of a bad tour for a Euclidean problem. The upper picture shows the cities as they appear in the display from which the tour ordering is taken. The lower pictures show the actual positions of the cities in the plane.

Problem of defining the final tour.
After weight changes have stabilised, even if the broad form of the tour is good, it remains unfinished on small scales. Consider the situation shown in figure 5.6 for a Euclidean TSP. In this case cities B, C and D have become more selective than E and F, as is shown by their positions close to the loop in the display. E and F are in more central positions because their weights are distributed over a wider range of loop units. This creates a problem because if the tour is evaluated at this stage it alternates between the two groups as shown in the figure, and this is clearly not the optimal ordering.

Both these problems can be seen as being related to the degree of selectivity attained by cities. The first problem arises essentially because some cities become selective too quickly. The second problem arises because cities don't become selective enough to guarantee a sensible tour. Hence one approach to both problems is to attempt to control the selectivity directly, through the learning rule. Two possibilities for this are:

- Define a modification threshold ϵ as was done in the original model, so the Hebbian rule becomes

  $$\Delta W_{Aj} = \begin{cases} \eta \phi^\star_A \xi_j & \text{if } \phi^\star_A > \epsilon \\ 0 & \text{otherwise} \end{cases}$$

  Recall that at present $\epsilon = 0$ to prevent negative weights occurring. As ϵ is raised the number of patterns able to cause (positive) weight changes decreases.

- Use a non-linear Hebb rule, for example

  $$\Delta W_{Aj} = \eta (\phi^\star_A)^p \;\; \xi_j$$

  At present $p = 1$. As p is raised, the weight changes for patterns causing large potentials become much larger.

In both these rules, as the relevant parameter is raised, higher potentials become progressively more accentuated compared with lower ones. These methods do improve the tours, mainly by alleviating the second problem. They also raise a new difficulty however. Single loop units tend to become mapped to several cities which themselves have negligible weights from other loop units; that is, all the cities become connected to only a few of the loop units. When this happens there is nothing to tell these cities their overall orientation in the tour as a whole. Hence crucial contacts relating groups of cities together can be lost. For example with the cities shown in figure 5.6, if A has no connections to loop units that are common to groups (B,C,D) or (E,F), it no longer influences their orientation, which could lead to the ordering A,D,C,B,F,E,G. Broadly speaking the tour information becomes encoded onto too few loop points, and the multiple cities connected to a single loop point may lose their overall ordering with respect to the rest of the cities, leading to poor as well as poorly defined tours.

One way to prevent this occurring is to ensure an even distribution of weights by including a normalisation of the weights originating from each loop unit. However, this is not compatible with the existing normalisation of weights into each city. As might be expected, this second normalisation seems to interfere too strongly with the first, resulting in poor tours. Another method is to vary the 'competitiveness' of loop units, thereby ensuring that each unit gains its share of the weights to cities. This can be done by making the activity ξ_j a decreasing function of either the mean weight associated with j, or the mean potential in the output layer when j is active, making 'greedy' cells less competitive. These methods are only partially successful, and in many cases the original problems remain.

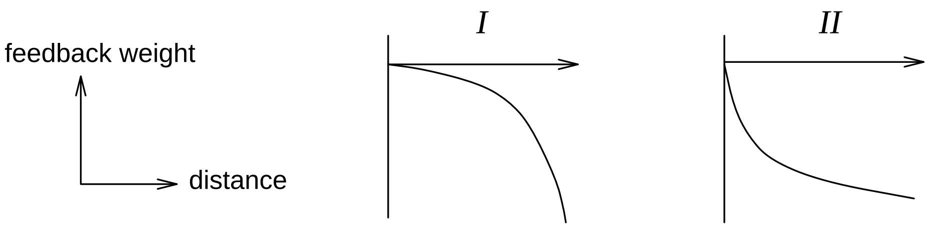

Figure 5.7: Varying the shape of the feedback function.

Another approach is to regard both the problems as being caused by interactions between units occurring at inappropriate ranges. Thus part-maps arise because units interact on too local a scale and hence are not bound to adopt a global ordering. Likewise the difficulty in defining a tour at the end is caused by the city units not competing vigorously enough for the available loop units, which is a result of the range of interaction between units in one or both layers being too broad. In this view, the solution is to vary the range of interaction in one or both layers, beginning with long range interactions which establish the broad form of the mapping, and proceeding to short range where each city has a strong association with one part of the loop, but the regions of association still overlap. Furthermore, this is akin to what happens with the *EN* method: as the annealed parameter K is reduced, tension forces (*ie.* the degree of interaction between loop units) decrease, while at the same time the range of the city forces becomes progressively shorter.

The range of feedback between cities.

In the non-Euclidean case the "range" of the city forces cannot be given explicitly due to the absence of a metric, but instead is encoded in the form of interactions between units representing cities. This in turn is controlled by the magnitude and form of the curve relating feedback between two cities to the given distance between them. The shape of the curve determines at which range the inter-city distances are most influential. For example in figure 5.7, feedback curve *(I)* accentuates competition between *distant* cities whereas in *(II)* it is nearby cities that are in competition the most. The tours may be improved in some cases by beginning with feedback of the form of *(I)* and gradually altering it to *(II)*. However which feedback function is optimal appears to be rather dependent on the particular TSP instance, and in many cases no improvement is seen.

Controlling the range of interaction in the loop.

The relevant quantity determining the range of interaction in the loop is m/M. To avoid part-maps it is best to begin with $m/M = 1/3$, intuitively because four objects can be wrongly ordered but three cannot. To ensure each city gains a valid place on the tour, m/M is reduced. However there is no point in decreasing m below 2 (since at $m = 1$ there are no longer any interactions to give the units the topology of a loop) and with $M = N$ the second problem remains. Therefore M needs to be greater than the number of cities to allow the ratio m/M to decrease further.

This method of controlling development (by varying m) was adopted in preference to those discussed earlier. It is simple and reliable, and deals successfully with both of the problems. In practice if $M = 2.5N$, then by reducing m linearly $M/3$ to 2 over 1000 iterations and then continuing with $m = 2$ for a further 1000 iterations, part maps (at least for Euclidean problems) are almost eliminated, and all cities gain an unambiguous position in the tour.

5.5.9 An example.

The cities used here are the same as those in the example of the *LEN* method. The input width, m, was varied as described above. The rate of change, η, is set at $1/m$, which counters the decrease in accumulated Hebbian changes as m is reduced. Figure 5.8 shows the projection of the cities onto the loop after 500, 1000 and 2000 iterations. By this stage the tour defined by the mapping is 8 % longer than the best tour obtained by 3-opt. As in the *LEN* example, the solid lines shown the ordering of cities in the 3-opt tour, and the small number of lines crossing the large circle indicates that most of the tour defined by the mapping agrees with that of 3-opt.

Figure 5.9 shows the development of the same mapping, projected back into the plane of the cities. Note that the mapping after 2000 iterations appears unfinished since it does not conform exactly to the city positions. However, the loop representation of the same weights matrix (Figure 5.8) shows that interpretation of the tour is clear, since all the cities are positioned unambiguously on the outer circle.

5.5.10 Analysis of the linear case.

A problem with any analysis of the algorithm's behaviour is the difficulty of dealing with the threshold, which is applied to the potentials. In looking at the condition under which the weights will increase, the analysis is therefore restricted to the case in which all the potentials are positive. Although this is only true at the early

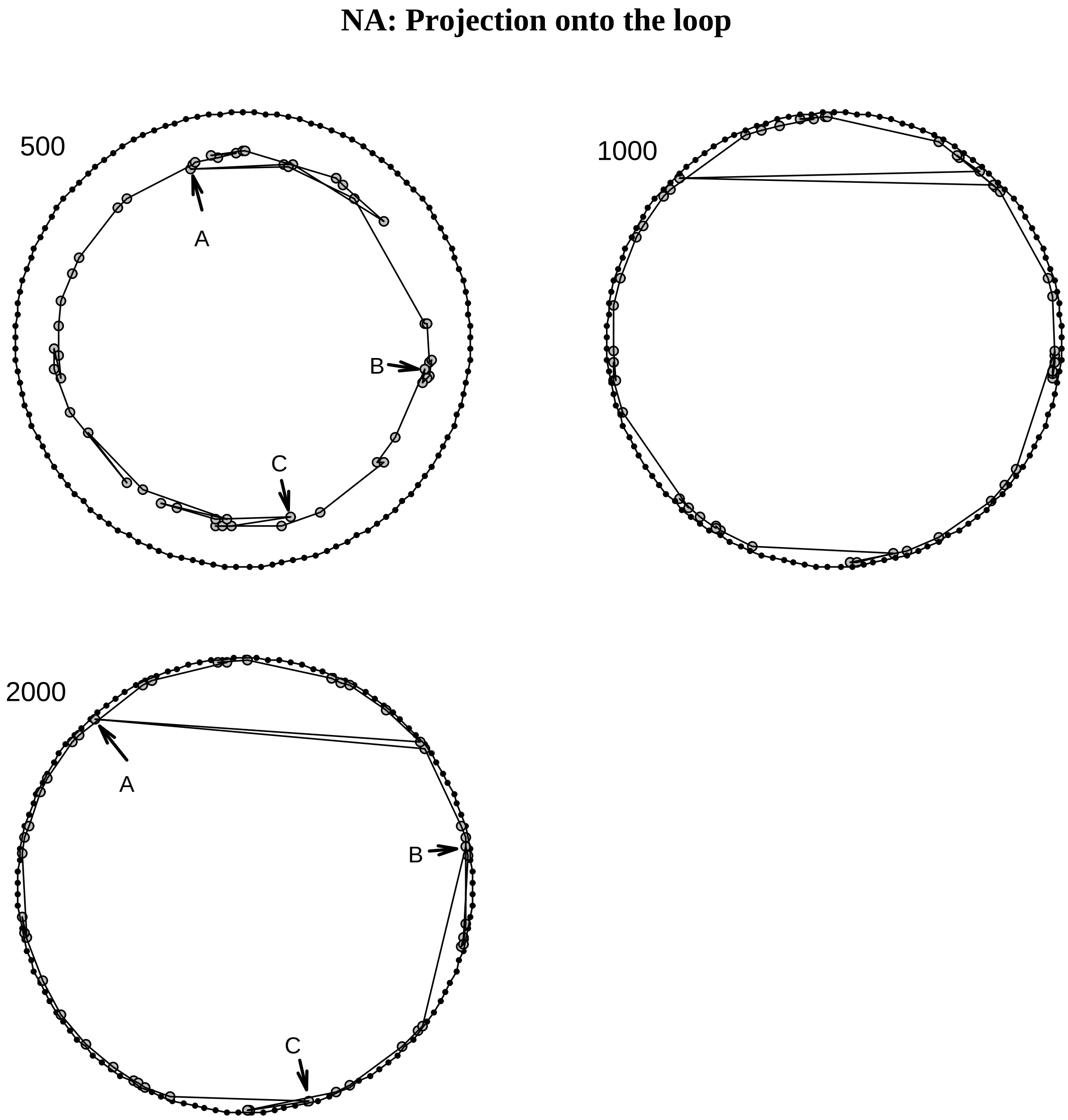

Figure 5.8: Example of NA on a 50 city Euclidean problem, shown after 500, 1000 and 2000 iterations. The dispersed circles in the three pictures denote the cities, which are projected onto a circle representing the loop. For comparison, the straight lines connecting cities indicate their ordering in the best tour obtained by 3-opt. A, B and C are the main points at which the tour defined by the mapping (which is given by ordering the cities according to their angular coordinate) differs from the 3-opt tour.

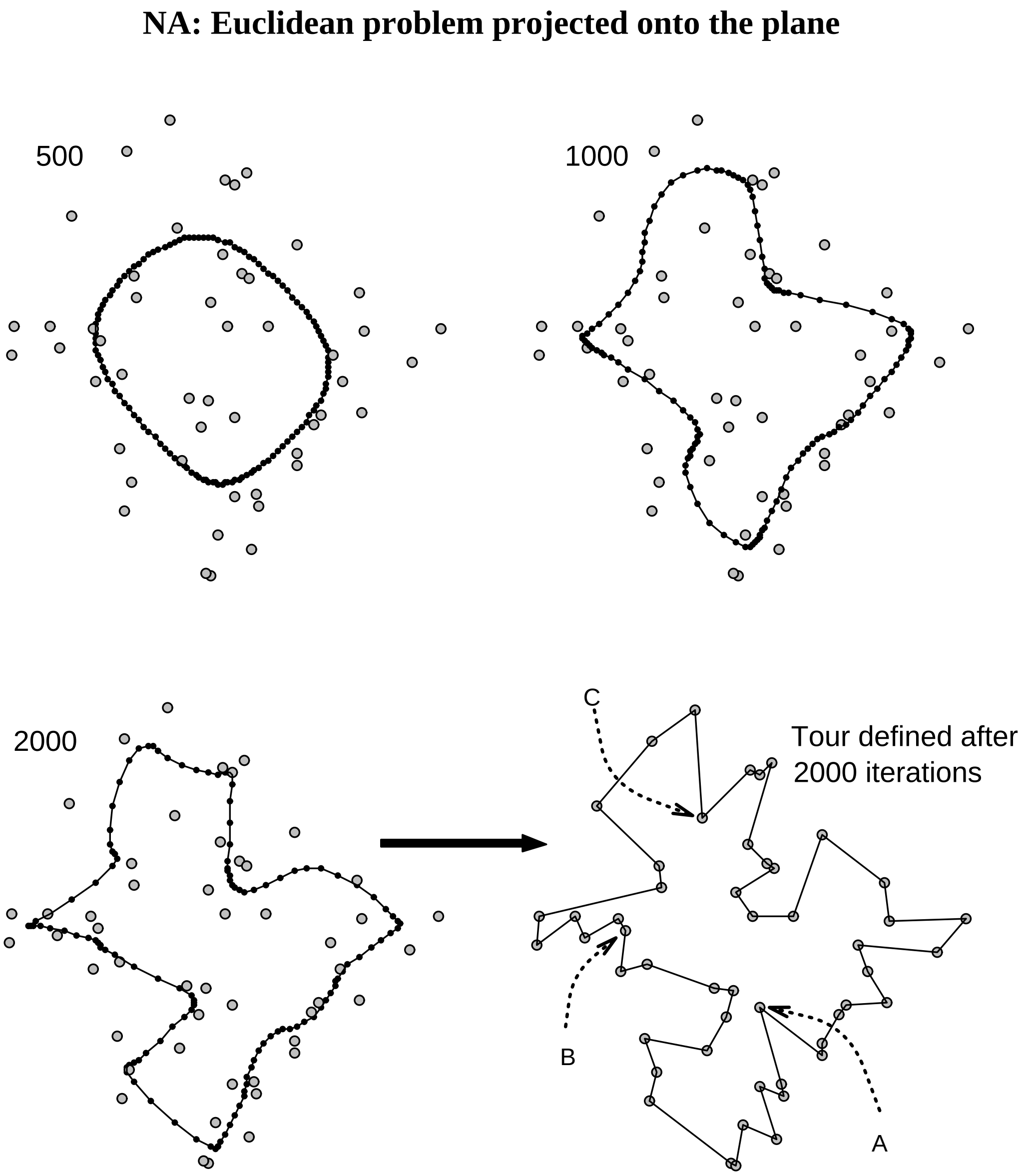

Figure 5.9: Example of NA on a Euclidean problem. The loop is shown projected into the plane in which the cities lie. The lower right picture shows the tour defined by the mapping after 2000 iterations. This may be compared with the 3-opt tour shown in the LEN figure earlier. Cities A, B and C, mentioned in the previous figure are also indicated.

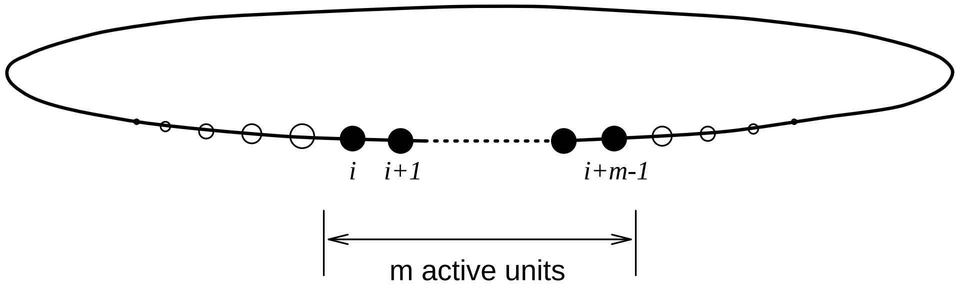

Figure 5.10: The input pattern labelled by (i).

stages in the development of the mapping, it should point to qualitative features of the algorithm's behaviour in general.

An input pattern always consists of m adjacent cells in the loop, as indicated in figure 5.10. The pattern starting at the i^{th} input as shown is to be denoted by a superscripted i on the relevant variable.

The change to the weight from loop unit j to city A after each pattern has been presented once but prior to normalising is

$$\begin{aligned} \Delta W_{Aj} &= \sum_{k=0}^{M-1} \Delta W_{Aj}^{(k)} \qquad \text{with } \Delta W_{Aj}^{(k)} = \eta \xi_j^{(k)} \phi_A^{(k)} \\ &= \eta \sum_{k=j-m+1}^{j} \phi_A^{(k)} \end{aligned}$$

where ϕ can be used instead of $\phi^\star$ since all potentials are assumed positive. From steps 1 and 2 of the simplified algorithm,

$$\begin{aligned} \phi_A^{(k)} &= \sum_{l=k}^{k+m-1} W_{Al} \;+\; \sum_B \omega_{AB} \sum_{l'=k}^{k+m-1} W_{Bl'} \\ \text{or} \quad \phi_A^{(k)} &= \sum_B \omega_{AB}^\star \sum_{l=k}^{k+m-1} W_{Bl} \end{aligned}$$

where $\omega_{AB}^\star = \omega_{AB} + \delta_{(A=B)}$. This gives the weight change before normalisation as

$$\Delta W_{Aj} = \eta \sum_B \omega_{AB}^\star \sum_{k=j-m+1}^{j} \sum_{l=k}^{k+m-1} W_{Bl}$$

which is the same as

$$\Delta W_{Aj} = \sum_B \omega_{AB}^\star \sum_{q=-(m-1)}^{(m-1)} (m - |q|) \;\; W_{B,j+q} \tag{5.12}$$

Now suppose that all the cities apart from A have uniform weights $W = 1/M$, and that the i^{th} weight to A is slightly different from the others, that is:

$$\begin{aligned} W_{Ai} &= \frac{1}{M} + \gamma \\ \text{with} \quad W_{A,j\neq i} &= \frac{1}{M} - \frac{\gamma}{M-1} \end{aligned}$$

to keep the sum of weights equal to one. It is interesting to consider whether this fluctuation will be increased after one iteration, or whether it will fade back towards uniformity.

For city A, after making the Hebbian weight changes, normalisation rescales all its weights:

$$W_{Ai}^{new} = \frac{W_{Ai} + \Delta W_{Ai}}{1 + \sum_l \Delta W_{Al}}$$

The overall weight change is therefore

$$\begin{aligned} \tilde{\Delta} W_{Ai} &= W_{Ai}^{new} - W_{Ai} \\ &= \frac{\Delta W_{Ai} - W_{Ai} \sum_k \Delta W_{Ak}}{1 + \sum_l \Delta W_{Al}} \end{aligned} \tag{5.13}$$

The denominator is positive. The sign of the numerator indicates what happens to the perturbed weight.

Noting that

$$\sum_{q=-(m-1)}^{(m-1)} (m - |q|) = m^2$$

the following are easily obtained from equation 5.12:

$$\begin{aligned} \Delta W_{Ai} &= \frac{\eta m^2}{M}(1 + \sum_B \omega_{AB}) + \eta m \gamma (1 - \tfrac{m+1}{M-1}) \\ \sum_j \Delta W_{Aj} &= \eta m^2 (1 + \sum_B \omega_{AB}) \end{aligned}$$

By substituting these values into equation 5.13, the numerator is found to be

$$\begin{aligned} \text{numerator} &= \gamma \eta m \{ \tfrac{M-m-2}{M-1} - m\Omega_A \} \\ \text{where} \quad \Omega_A &= 1 + \sum_B \omega_{AB} \end{aligned} \tag{5.14}$$

Notice that γ is just a multiplicative factor. From equation 5.11, Ω_A lies between zero and one. If A is an isolated city, $\Omega_A \sim 0$ and the bracketed term in equation

5.14 is positive. In that case any initial perturbation will be accentuated and A's selectivity increased: if γ is positive the city becomes more selective for loop unit i, and if γ is negative the normalisation ensures that it selects for the rest of the loop instead. Conversely if A is part of a close grouping of cities, $\Omega_A \sim 1$ so the bracketed term is negative. Hence the weight change for $\gamma > 0$ is negative, and that for $\gamma < 0$ is positive: the perturbation is damped out. The increase in selectivity of a given cell therefore depends on its activity in a somewhat counterintuitive way, in that the post-synaptic cells which are least active are most selective.

Given an initial random distribution of weights, the isolated cities are the first to select a place on the tour. As the weights to an isolated city change, the response of the city to inputs at that part of the loop grows, eventually inhibiting other cities from selecting that region, causing grouped cities to select other parts of the loop *en masse*. These cities in turn become able to selectively inhibit one another, breaking the group up further. Broadly speaking the algorithm can be said to organise the distant cities first followed by successively smaller groupings of cities. That is, the development of the mapping proceeds from large scale to small scale.

Of course this only applies where all the potentials are positive, which is only true in the initial stages. What then is the effect of resetting negative potentials to zero? As a given city becomes more selective, its potential for rejected parts of the loop becomes negative. This makes the purely Hebbian change zero, so in effect the unit begins to ignore this part of the loop. This makes sense: we do not want effects at distant parts of the tour to continue to interfere with development elsewhere once the broad form of the mapping has been established.

5.6 Results.

Both algorithms were tested on a number of Euclidean and non-Euclidean problems.

The figures quoted are the percentage increase of the tour length L over the presumed optimal tour length $L^\star$:

$$100 \times \frac{L^\star - L}{L^\star}$$

$L^\star$ is the length of the shortest tour found in 100 independent runs of 3-opt, starting from random tour configurations.

In both methods the inter-city distances are first rescaled to make the maximum distance unity, and the number of loop cells $M = 2.5N$. In simulations of the *LEN* method, $\alpha = 0.15$, $\beta = 2.0$. The annealed parameter K begins at 0.15 for the 30 city problem and 0.2 for the 50 city problems, unless noted otherwise. K is held constant at this value for the first 1000 iterations and is then reduced linearly to 0.002 over a further 1000 iterations, at which time the tour is evaluated. For the *NA* algorithm, the number of inputs active at any one time (m) begins at $M/3$ and decreases linearly over 1000 iterations to 2, remaining at 2 for a further 1000 iterations, after which the tour is evaluated. Both algorithms were run five times on each problem.

5.6.1 Euclidean problems.

Hopfield and Tank quote a *best* tour for a set of 30 cities of 19% longer than optimum. Using the same cities, both *NA* and *LEN* produce tours only 2-5% longer than optimum (taken from 10 runs of each algorithm).

The algorithms were tested on the set of five 50-city problems used by [Durbin & Willshaw 1987] to evaluate the *EN*. Each of these consists of cities distributed at random in the unit square. The table below shows L for these problems, with the *EN* results (means over 5 trials, from [Durbin & Willshaw 1987]) for comparison.

Cities in the plane					
Cities	Elastic Net	*LEN*		*NA*	
		min	max	min	max
a	2.3	4.2	10.1	3.3	4.7
b	0.7	0.7	0.7	7.8	7.8
c	3.1	3.9	3.9	5.2	8.2
d	3.5	7.1	8.8	4.9	6.3
e	5.2	3.4	7.3	11.7	15.1

5.6.2 Non-Euclidean problems.

In order to evaluate the methods on non-Euclidean problems, Euclidean city sets were first generated by placing cities at random positions in the unit square and forming the matrix of inter-city distances. Each of these distances was then increased by a random number within some percentage of the original distance. The reverse distance was altered by the same amount to keep the matrix symmetric. This is the same as taking a map consisting of roads that are straight lines between cities and putting bends in some of the roads. Hence these problems could be described as "approximately Euclidean". The results for perturbations of up to 1%, 10%, 100% and 1000% are shown below:

Perturbed Euclidean matrices.					
Perturbation	Cities	*LEN*		*NA*	
		min	max	min	max
1 %	a	4.2	7.1	5.8	6.4
	b	8.1	8.1	15.2	21.9
	c	7.4	7.4	5.3	7.4
	d	3.2	3.9	3.2	3.2
	e	2.3	16.0	14.1	14.3
10 %	a	10.9	20.1	6.0	6.4
	b	4.4	5.3	8.9	8.9
	c	2.4	13.7	8.6	8.7
	d	6.6	18.1	13.2	13.2
	e	4.1	4.4	3.2	3.5
100 %	a	4.4	5.9	11.2	13.8
	b	6.6	7.1	13.0	13.5
	c	3.6	9.4	7.1	11.7
	d	34.0	59.1	27.8	27.8
	e	6.1	38.9	12.5	15.0
1000 %	a	83.3	119.4	57.2	57.2
	b	56.9	73.4	88.5	97.7
	c	27.6	28.9	74.1	74.2
	d	56.7	149.2	33.3	34.9
	e	31.1	92.1	62.5	66.2

Another method of generating a non-Euclidean symmetric problem is simply to choose the inter-city distances at random (uniformly) within some range while keeping the matrix symmetric. The algorithms were tested on five problems generated in this way, with the following results:

Random symmetric distance matrices.				
Cities	*LEN*		*NA*	
	min	max	min	max
a	126.9	147.8	137.7	251.1
b	145.2	174.3	173.1	194.7
c	164.5	185.1	187.4	194.4
d	197.1	224.0	186.4	234.1
e	189.1	233.8	259.0	283.2

The average percentage increases, taken over 5 runs on each of the five city sets are:

Average performance over all runs.				
Cities	*LEN*		*NA*	
Euclidean	7.1	$\pm$ 6.6	7.4	$\pm$ 3.3
Perturbed by 1 %	6.3	$\pm$ 3.9	8.3	$\pm$ 5.6
10 %	10.4	$\pm$ 9.2	7.8	$\pm$ 3.6
100 %	15.6	$\pm$ 12.8	14.8	$\pm$ 6.3
1000 %	70.3	$\pm$ 36.7	57.2	$\pm$ 19.8
Non-Euclidean	166.7	$\pm$ 31.7	209.6	$\pm$ 39.9

5.7 Conclusions

For Euclidean problems both *LEN* and *NA* find tours with lengths which are reasonably close to those of the *EN* method. This is very encouraging, as neither method is able to exploit the Euclidean nature of the problem in the way that *EN* does.

The performance of both algorithms degrades considerably for large distortions of the inter-city distances, which suggests that the general approach is inappropriate for dealing with highly non-Euclidean instances of the TSP. In such problems the Triangle Inequality is frequently broken. This rule is a bound on what might be called "common sense" spatial relationships; it is broken if the shortest path between A and B is not the direct one, but rather through (say) C. Since the topographic mapping approach is intrinsically spatial, it is not entirely surprising that it should break down in these cases.

However there is a middle ground, where the distance matrix is "nearly Euclidean". Even for small perturbations of this type the *EN* method can no longer be used, but *LEN* and *NA* still find short tours in these cases. This would appear to be the domain in which algorithms based on the formation of a topographic mapping can be used for non-Euclidean problems.

Epilogue

This thesis is about learning algorithms which attempt to maximise performance on pattern classification and combinatorial optimisation tasks. These are distinct problems and the neural network methods employed to solve them are very different. However, they derive their power from the same source - radically local computation. Indeed, it is the essential idea of neural networks, borrowed from real nervous systems, that complex calculations might be performed more efficiently by large numbers of simple processors acting in parallel and interacting locally, rather than a single processor performing extended serial calculations.

Understanding this requires understanding the multiple levels of computational analysis. The same computation can be implemented by different algorithms, each of which in turn might be implemented either serially or in parallel. The field of neural networks is not merely about parallel mechanisms for the lowest level, it is rather the search for wholly different classes of algorithm.

Associated with these multiple levels of computation are multiple levels of locality. Some purely low-level local mechanisms (*eg* tension in a rubber band) implement higher-level global goals (*eg* minisiming its length). The methods in this thesis are local. Even the global measure to be optimised only figures implicitly, never explicitly, in the operation of the algorithm.

Consider the networks in Part I. Generic backpropagation solves the problem of minimising performance error subject to a particular architecture, and using a non-local error measure during learning. This is not necessarily the task of interest, given that units and connections are taken to be "cheap"; more often it is poor network performance that is costly. The networks in Part I solve the dual of this; they minimise the size of the network subject to a particular criterion of error, and use only local measures. Optimisation of the whole network follows directly from the local optimisation carried out by each unit. Not only is the added locality advantageous in itself for implementation, but also the problem being solved is more relevant.

Locality is equally significant for the networks in Part II. Conventional methods for the TSP involve non-local computation, because entire tours are rejected or entire sections are rearranged at every step. Here, though, minimisation of the tour length is achieved by the formation of a topographic mapping, which itself is the result of purely local and thus highly parallelisable computations. The global constraint that the tour be valid is merely implicit in the running of the networks.

The algorithms and results presented here show how radically local computation can be applied succesfully to difficult problems. Impressive global solutions emerge.

Bibliography

Bibliography

Ash,T. 1989. Dynamic node creation in backpropagation networks. *ICS Report 8901, Institute for Cognitive Science, Univ. of California, San Diego.*

Angéniol,B., de La Croix Vaubois,G., & Le Texier,J.-Y. 1988. Self organising feature maps and the travelling salesman problem. *Neural Networks* **1:** p289-293.

Aue,A. 1990. A new class of algorithms for the Steiner problem. *MSc. Thesis, Department of Computer Science, University of Edinburgh.*

Baum,E. & Haussler,D. 1989. What size net gives valid generalization? *Neural Computation* **1:** p151-160

Binder,K. 1979. Monte Carlo Methods in Statistical Physics. *Topics in Current Physics,* **7** *(Berlin:Springer)*

Bliss,T.V.P. & and Lomo,T. 1973. Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetized rabbit following stimulation of the perforant path. *J. Physiology* **232:** p331-356.

Block,H.D. 1962. The perceptron: a model for brain functioning. *Reviews of Modern Physics,* **34:** 1, p123-135.

Blum,A. & Rivest,R.L. 1988 *Proceedings of the First Workshop on Computational Learning Theory (Morgan Kaufmann),* 9.

Brady,M., Raghavan,R. & Slawny,J. 1988. Gradient descent fails to separate. *Proc. IEEE International Conference on Neural Networks, San Diego,* **I:** p649-656.

Breiman,L., Friedman,J., Olshen,R. & Stone,C.J. 1984. Classification and Regression Trees. *Wadsworth Belmont, California (1984).*

Burr,D.J. 1988. An Improved Elastic Net Method for the Travelling Salesman Problem. *Proc. IEEE International Conf. on Neural Networks.,* **I**: p69-76.

Chauvin,Y. 1989. A Back-propagation algorithm with optimal use of hidden units. *Advances in Neural Information Processing Systems (NIPS), Touretsky,D.S. (ed) San Mateo : Morgan Kaufmann.* **1:** p519-526.

Cover,T.M. 1965. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Trans. Electron. Comput.* **14:** p326-334.

Cowan,J.D. & Sharp,D.H. 1987. Neural Nets. *LA-UR-87-4098, Los Alamos.*

Denker,J., Schwartz,D., Wittner,B., Solla,S., Howard,R., Jackel,L. and Hopfield,J. 1987. Large Automatic Learning, Rule Extraction and Generalization, *Complex Systems* **I**: p877-922

Duda,R.O. & Hart,P.E. 1973. Pattern Recognition and Scene Analysis, *John Wiley and Sons, New York.*

Durbin,R., Szeliski,R. & Yuille,A. 1989. An analysis of the elastic net approach to the travelling salesman problem. *Neural Computation* **1:** p348-358.

Durbin,R. & Willshaw,D.J. 1987. An analogue approach to the travelling salesman problem using an elastic net method. *Nature, (April 16,1987)* **326:** 6114, p689-691.

Fahlman,S. & Lebiere,C. 1990. The Cascade-Correlation Learning Architecture. *In Advances in Neural Information Processing Systems (NIPS), Touretsky,D.S. (ed) San Mateo : Morgan Kaufmann.* **2:** p524-532.

Fort,J.C. 1988. Solving a combinatorial problem via self-organising process: an application of the Kohonen algorithm to the travelling salesman problem. *Biological Cybernetics.* **59:** p33-40.

Frean,M.R. 1989. Internal Report. *Department of Physics, University of Edinburgh.*

Frean,M.R. 1990. The Upstart Algorithm: A Method for Constructing and Training Feedforward Neural Networks. *Neural Computation* **2:** 2, p198-209.

Gallant,S.I. 1986a. Optimal Linear Discriminants. *IEEE Proc. 8th Conf. on Pattern Recognition, Paris.*

Gallant,S.I. 1986b. Three Constructive Algorithms for Network Learning. *Proc. 8th Annual Conf. of Cognitive Science Soc.* p652-660.

Gallant,S.I. 1989. Perceptron-based learning algorithms. *Technical Report NU-CCS-89-21, College of Computer Science, Northeastern University, Boston,MA.*

Gaze,R.M. 1970. The formation of nerve connections. *London, Academic Press.*

Golea,M. & Marchand,M. 1990. A growth algorithm for neural network decision trees. *Europhysics Lett.,* **12:** 3, p205-210.

Goodhill,G.J. & Willshaw,D.J. 1990. Application of the elastic net algorithm to the formation of ocular dominance stripes. *Network* **1:** 1, p41-59.

Grossman,T., Meir,R. & Domany,E. 1989. Learning by choice of internal representations. *Complex Systems* **2**, 555.

Hanson,S.J. & Pratt,L.J. 1989. Some comparisons of constraints for minimal network construction with back-propagation. *Advances in Neural Information Processing Systems (NIPS), Touretsky,D.S. (ed) San Mateo : Morgan Kaufmann.* **1:** p177-185.

Hao,J., Shaohua,T. & Vandewalle,J. 1990. A geometric approach to the structural synthesis of multilayer perceptron neural networks. *in Proc. International Neural Networks Conf., Paris, 1990.* p881-885.

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

Hegde,S.U., Sweet,J.L. & Levy,W.B. 1988. Determination of Parameters in a Hopfield /Tank Computational Network. *Proc. IEEE International Conf. on Neural Networks.,* **II:** p291-298

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

Holland,J. 1975. Adaptation in Natural and Artificial Systems. *Ann arbor: University of Michigan Press.*

Honavar,V. & Uhr,L. 1988. A network of neuron-like units that learns to perceive by generation as well as reweighting of its links. *in Proc. of the 1988 Connectionist Models Summer School, Touretsky,D., Hinton,G. & Sejnowski,T. (eds). Morgan Kaufmann, San Mateo.* p472-484.

Hopfield,J.J. 1982. Neural networks and physical systems with emergent collective computational abilities. *Proc. Natl. Acad. Sci. USA* **79:** p2554-2558

Hopfield,J.J. 1984. Neurons with graded response have collective computational properties like those of two-state neurons. *Proc. Natl. Acad. Sci. USA* **81:** p3088-3092

Hopfield,J.J. & Tank,D.W. 1985. Neural computation of decisions in optimization problems. *Biol.Cyber.,* **52:** p141-152

Hueter,G. 1988. Solution of the Travelling Salesman Problem with an Adaptive Ring. *Proc. IEEE International Conf. on Neural Networks.,* **I**: p85-92.

Judd,S. 1987. Learning in networks is hard. *Proc. IEEE First Conference on Neural Networks, San Diego 1987.* (IEEE Cat. No. 87TH0191-7), **II**, p685-692.

Kirkpatrick,S., Gelatt,Jr.,C.D. & Vecchi,M.P. 1983. Optimization by simulated annealing. *Science* **220:** p671-680.

Kirkpatrick,S. 1984. *Journal of Statistical Physics,* **34:** p975.

Kohonen,T. 1977. Associative memory - a system-theoretical approach. *Berlin, Heidelberg, NewYork: Springer.*

Kohonen,T. 1982. Self-organized formation of topographically correct feature maps. *Biological Cybernetics* **43:** p59-69.

Krogh,A., Thorbergsson,G.I., & Hertz,J.A. 1990. *Advances in Neural Information Processing Systems (NIPS), Touretsky,D.S. (ed) San Mateo : Morgan Kaufmann.* **2**.

Kruschke,J.K. 1988. Creating Local and distributed bottlenecks in hidden layers of back-propagation networks. *in Proc. of the 1988 Connectionist Models Summer School, Touretsky,D., Hinton,G. & Sejnowski,T. (eds). Morgan Kaufmann, San Mateo.* p120-126.

Lawler, Lenstra, Rinooy Khan & Shmoys (eds). 1985. The Travelling Salesman Problem. *Wiley, New York.*

Le Cun,Y. 1985. *Proc. Cognitiva* **85:** 593.

Le Cun,Y. 1989 Generalization and network design strategies. *In Pfeifer,R., Schreter,Z., Fogelman,F. and Steels,L., editors, "Connectionism in Perspective", Zurich, Switzerland. Elsevier.*

Le Cun,Y., Denker,J.S., Solla,S.A. 1990. Optimal Brain Damage. *In Advances in Neural Information Processing Systems (NIPS), Touretsky,D.S. (ed) San Mateo : Morgan Kaufmann.* **2**.

Lewis,P.M. & Coates,C.L. 1967. Threshold Logic. *Wiley, New York.*

Lin,S. 1965. Computer solutions of the travelling salesman problem. *Bell Syst. Tech. J.,* **44:** p2245.

Lin,S. & Kernighan,B.W. 1973. An effective heuristic algorithm for the travelling salesman problem. *Oper. Res.* **21:** p498-516.

Lippmann,R.P. 1987. An Introduction to Computing with Neural Nets. *IEEE ASSP Magazine, April 1987*, p4-22.

Mèzard,M. and Nadal,J-P. 1989. Learning in Feedforward Layered Networks : the Tiling Algorithm, *J.Physics A*, **22:** 12, p2191-2203

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

Morgan,N. & Bourlard,H. 1990. Generalization and Parameter Estimation in Feed-forward nets: some experiments. *In Advances in Neural Information Processing Systems (NIPS), Touretsky,D.S. (ed) San Mateo : Morgan Kaufmann.* **2:** p630-637.

Moser,M. & Smolensky,P. 1989. Skeletonization : A technique for trimming the fat from a network via relevence assessment. *Advances in Neural Information Processing Systems (NIPS), Touretsky,D.S. (ed) San Mateo : Morgan Kaufmann.* **1:** p107-115.

Mühlenbein,H., Georges-Schleuter,M. & Krämer,O. 1988. Evolution algorithms in combinatorial optimization. *Parallel Computing,* **7:** p65.

Nabutovsky,D., Grossman,T. & Domany,E. 1990. Learning by CHIR without storing internal representations. *Submitted to Complex Systems.*

Nadal,J-P. 1989. Study of a Growth Algorithm for Neural Networks *International J. of Neural Systems,***1:** 1, p55-59

Nilsson,N.J. 1965. Learning Machines. *McGraw-Hill, New York.*

Pearlmutter,B.A. & Hinton,G.E. 1986. G-Maximization: an Unsupervised Learning Procedure for Discovering Regularities. In Denker,J.S., editor, *Neural Networks for Computing: American Institute of Physics Conference Proceedings.* **151:** p333-338.

Personnaz, Dreyfus and Knerr. 1990. *Presented at Neural Networks for Computing Conference, Snowbird.*

Peterson,C. & Söderberg,B. 1989. A new method for mapping optimization problems onto neural networks. *International J. of Neural Systems,***1:** 1, p3-22.

Peterson,C. 1990. Parallel Distributed Approaches to Combinatorial Optimization - Benchmark Studies on Travelling Salesman Problem. *Neural Computation.* **2:** 3, p261-269.

Plaut,D.C., Nowlan,S.J. & Hinton,G.E. 1986. *Technical report. CMU-CS-86-126*, Carnegie-Mellon University.

Ramacher,U. & Wesseling,M. 1989. A geometrical approach to neural network design. *Proc. of the First International Joint Conference on Neural Networks, Washington DC.* **II:** p147-153.

Ritter,H. & Schulten,K. 1988. Kohonen's Self-Organising Maps: Exploring their computational capabilities. *Proc. IEEE International Conf. on Neural Networks.,* **I:** p109-116.

Rohwer,R. 1990. The "moving targets" training algorithm. *Advances in Neural Information Processing Systems (NIPS), Touretsky,D.S. (ed) San Mateo : Morgan Kaufmann.* **2:** p558-565.

Rosenblatt,F. 1962. Principles of Neurodynamics, *Spartan Books, New York.*

Rujan,P., & Marchand,M. 1989. A Geometric Approach to Learning in Neural Networks. *in Proc. of the Fisrt International Joint Conference on Neural Networks, Washington DC.* **II:** p105-109.

Rumelhart,D.E., Hinton,G.E., and Williams,R.J. 1985. Learning Internal Representations by error propagation. *ICS Report 8506, Institute for Cognitive Science, UCSD, La Jolla, CA.*

Rumelhart,D.E., McClelland,J.L., and the PDP Research Group. 1986. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume I Foundations,* MIT Press.

Rumelhart,D.E. & Zipser,D. 1986. Feature Discovery by Competitive Learning. *Chapter 5 in (Rumelhart et al. 1986)*

Saffery,J. 1990. A Neural Network Simulator for a Constructive Algorithm. *MSc thesis, Computer Science, University of Edinburgh.*

Simic,P.D. 1990. Statistical mechanics as the underlying theory of "elastic" and "neural" optimisations. *Network* **1:** 1, p89-103.

Simmen,M. 1990. Parameter Sensitivity of the Elastic-Net Approach to the Travelling Salesman Problem. *Preprint 90/495, Department of Physics, University of Edinburgh.*

Sirat,J.A. & Nadal,J-P. 1990. *Presented at Neural Networks for Computing Conference, Snowbird.*

Smith,G. 1990. BackPropagation with Dynamic Topology and Simple Activation Functions. *Tech Report 90-12, Computer Science, Flinders University of South Australia.*

Solla,S.A., Schwartz,D.B., Tishby,N. & Levin,E. 1990. Supervised Learning : A theoretical framework. *Advances in Neural Information Processing Systems (NIPS), Touretsky,D.S.(ed) San Mateo : Morgan Kaufmann.* **2.**

SPSS-X. 1984. SPSS-X User's Guide. *McGraw-Hill Book Company, New York.*

Sontag,E.D. 1988. Some remarks on the Backpropagation algorithm for neural net learning. *Report SYNCON-88-02, Rutgers Center for Systems and Control.*

Sontag,E.D. & Sussmann,H.J. 1988a. Backpropagation separates when perceptrons do. *Report SYNCON-88-12, Rutgers Center for Systems and Control.*

Sontag,E.D. & Sussmann,H.J. 1988b. Backpropagation can give rise to spurious local minima even for networks without hidden layers. *Report SYNCON-88-12, Rutgers Center for Systems and Control.*

Sun,G.Z., Lee,Y.C. & Chen,H.H. 1988. A novel net that learns sequential decision process. *Neural Information Processing Systems (NIPS), Dana Z. Anderson (ed) American Institute of Physics, New York.* p760-766.

Thacker,N.A. & Mayhew,J.E.W. 1990. Designing a layered network for context sensitive classification. *Neural Networks,* **3:** 3, p291-300.

von der Malsburg,C. 1973. Self-organization of orientation sensitive cells in the striate cortex. *Kybernetik* **14:** p85-100.

von der Malsburg,C. & Willshaw,D.J. 1977. How to label nerve cells so that they can interconnect in an ordered fashion. *Proc. Natn. Acad. Sci. U.S.A.* **74:** p5176-5178.

Werbos,P. 1974. Beyond Regression: New Tools for Prediction and Analysis in the Behavioural Sciences. *PhD Thesis, Harvard University Committee on Applied Mathematics, Cambridge, MA.*

Widrow,B. & Hoff,M.E. 1960. Adaptive switching circuits. *IRE WESCON Convention Record, New York : IRE*, p96-104.

Willshaw,D.J. & Dayan,P. 1990. Optimal Plasticity from Matrix Memories: What Goes Up Must Come Down. *Neural Computation* **2:** 1, p85-93.

Willshaw,D.J. & von der Malsburg,C. 1976. How patterned neural connections can be set up by self-organisation. *Proc.R.Soc.Lond.B*, **194:** p431-435

Willshaw,D.J. & von der Malsburg,C. 1979. A marker induction mechanism for the establishment of ordered neural mappings: its application to the retinotectal problem. *Proc.R.Soc.B*, **287:** p203-243.

Wilson,G.V. & Pawley,G.S. 1988. On the stability of the TSP algorithm of Hopfield and Tank. *Biol.Cyber.,* **58:** p63-70.

Wittner,B.S. & Denker,J.S. 1988 *Advances in Neural Information Processing Systems (NIPS), Dana Z. Anderson (ed), American Institute of Physics, New York.*

Yuille,A.L. 1990. Generalized Deformable Models, Statistical Physics, and Matching Problems. *Neural Computation* **2:** 1, p1-24.